A universal storage plugin for Elektra

BACHELOR'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Bachelor of Science

in

Software & Information Engineering

by

Felix Berlakovich
Registration Number 0929233

to the Faculty of Informatics

at the Vienna University of Technology

Advisor: Ao.Univ.Prof. Dipl.-Ing. Dr.techn. Franz Puntigam
Assistance: Univ.Ass. Dipl.-Ing. Markus Raab

Vienna, 1st March, 2016

Felix Berlakovich	Franz Puntigam

Erklärung zur Verfassung der Arbeit

Felix Berlakovich
Edelhofgasse 34/11
1180 Wien

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 1. März 2016

Felix Berlakovich

Abstract

On today's computer systems, configuration files are omnipresent. Administrators, as well as developers, are confronted with an ever growing zoo of different configuration file formats. Elektra, a multi-platform configuration management library, aims to provide a common view of all these configurations. However, to achieve this goal Elektra must be able to read and write the configuration files. At the time of this writing, Elektra supports only 9 different configuration file formats because each configuration file format needs it's own storage plugin. In contrast, the goal of this thesis is to create a single general storage plugin that supports dozens of configuration file formats. We show how Augeas, along with the many configuration file formats it supports, can be integrated in Elektra as a single storage plugin. As a result, Elektra now supports more than 100 different configuration file formats. We compare the new storage plugin to existing storage plugins and discuss important features of Elektra storage plugins in general. Finally, we benchmark the new plugin against an existing Elektra storage plugin. We show that, in contrast to the existing plugin, the runtime of the new plugin quickly increases with larger configuration files. We conclude that the new storage plugin is well suited to integrate new configuration file formats within Elektra, but existing storage plugins often can provide better abstraction and better runtime performance.

Contents

CHAPTER 1

Introduction

1.1 Problem Statement

Most applications need some kind of configuration. That is, a set of user preferences or application parameters that control the applications runtime behaviour. As the number of applications on a system increases so does the work required to manage their configurations. This is a problem because operating systems today host hundreds of different configurations. Many requirements arise, including but not limited to:

- providing a single view to all application configurations on a system
- cascading configurations, i.e. stacking multiple configurations
- reacting to configuration changes, e.g. by logging or notifying other applications
- validating configuration changes, i.e. prevent semantically or syntactically incorrect configurations

Different solutions have been proposed to fulfill these requirements. Most of these solutions provide lots of features, but fail to take existing configurations into account. For them to work an application must be written with the configuration library already in mind. Examples for such solutions are dconf, the Windows® Registry and KDEs kconfig. For example, the Windows® Registry fails to integrate applications that currently use the INI file format. Applications using the INI format cannot be configured via the Windows® Registry.

This is where Elektra steps in. Elektra is a portable library for reading and writing application configurations in a uniform way. Its modular design allows the integration of features like logging of changes or preventing corrupt configurations via validation. Elektra allows to reuse existing configurations by integrating them with a custom plugin for each configuration file format. However, currently there are few plugins for different file formats available. Before this work, plugins for 9 different configuration file formats existed.

In contrast to that, supporting many different configuration file formats is a strength of Augeas. Augeas is also a library for reading and writing configuration files, but unlike Elektra it focuses mainly on parsing. This means that Augeas does not handle any problem apart from parsing configuration files and presenting them as a tree.

As a result we are left with two powerful libraries for dealing with configuration files, but once again none of them is able to solve the full stack of requirements mentioned before. Without support for more configuration file formats neither administrators nor developers can benefit from the many useful features Elektra provides.

1.2 A More Detailed Look

1.2.1 Elektra

Elektra presents all application configurations in a hierarchical key database. The database consists of `Keys` representing configuration directives and `KeySets` grouping those `Keys`. An Elektra `Key` has a unique name and a value. For example, each entry of an INI configuration file could be represented by an Elektra `Key`. All `Keys` occurring in this configuration file are then grouped in a `KeySet`. Each `Key` may have so called `MetaKeys`. `MetaKeys` can hold meta information about the `Keys` they are attached to.

Elektra handles different configuration file formats by using *backends*. A backend is built from a stack of plugins, where different plugins serve different duties. For example, some of them filter or modify the configuration data passing by (e.g. by changing the encoding). Others serve as storage plugins. Storage plugins are responsible for reading from and writing to the actual configuration of the application. Different storage plugins allow Elektra to reuse the configuration file formats that already exist. This is in contrast to introducing yet another format and forcing application developers to adopt this new format.

A common issue when implementing a new storage plugin is the implementation of a parser for the configuration file. While an easy to use system interface exists for some configurations (e.g. `getmntent` for fstab), for most configuration files a parser has to be written by hand. Sometimes it might be possible to reuse existing parsing code of the application. However, most of the time the parsing code is strongly coupled to the application and therefore has to be reimplemented in the corresponding storage plugin.

As a result the parsing logic exists at least twice – once in the application itself and once in Elektra.

Apart from reading the configuration file, it must be written back after modifications have been made. For complex configurations this poses another challenge for the plugin developer. Great care should be taken, not to mangle parts usually ignored by a configuration file parser (e.g. comments or formatting). Otherwise information valuable to the user may be lost. In addition, both translation directions must be coordinated with each other. For example, data that has not been read in the first place cannot be written back properly. This is where Augeas comes into play.

1.2.2 Augeas

Augeas aims to tackle the problem of providing an editable view with a concept called lenses. Lenses are specifically designed to solve the issue of creating a view that can also be modified. Augeas already ships with more than 150 lenses for different configuration file formats. However, Augeas is not intended to solve any problem apart from configuration file manipulation. As the author of Augeas states in „Augeas - a configuration API“ [6]:

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

Unfortunately this means that many other issues related to configuration management remain unsolved, when solely Augeas is used. For example, the previously mentioned requirement of *configuration change notifications* remains unfulfilled. In particular features like cascading need a higher abstraction over the underlying configuration file than Augeas provides. Low abstraction means that details like platform specifics or the location of the configuration file leak through to the application using the configuration API. Most of the time these specifics are not useful or even hindering for applications that just aim to read or manipulate their configuration. A detailed description of these problems can be found in [16] and [18].

1.3 Goal of this Thesis

The goal of this thesis is to combine the benefits of Elektra and Augeas. While Elektra fulfills the previously introduced requirements, Augeas ships with a large number of lenses and therefore supports many different configuration file formats. The integration will happen via a newly developed Elektra storage plugin. The plugin will make use of the Augeas library and will be called the AUGEAS PLUGIN. Based on the plugin implementation the following questions will be answered:

RQ1 Can the existing lenses of Augeas be reused unchanged with the augeas plugin?

RQ2 Can the augeas plugin provide the same level of abstraction over the configuration as existing Elektra plugins?

RQ3 How resource intensive is the use of the augeas plugin compared to existing Elektra plugins?

CHAPTER 2

Theoretical Background

2.1 Introduction

The storage plugin implemented in the course of this work differs from existing storage plugins mainly in the handling of the view-update problem. While existing implementations provide independent view and update procedures, the augeas plugin makes use of a concept called bidirectional programming. In short, bidirectional programming provides a way to describe the transformation of a concrete source to an abstract view, and from a modified view back to the concrete source, in a single specification. Because the way of dealing with this problem is a key element of storage plugins in Elektra, a short introduction to the problem itself and possible solutions is given here. The view-update problem states the following:

Definition (view-update problem)**.** *Suppose that s is some concrete source of data, v is an abstract view of this data and q is a query from s to v such that v = q(s). Given an update u on v that transforms v to v', what is a possible translation of u that transform s to s' such that v' = q(s')? [14]*

A visualisation of Definition 2.1 can be seen in Figure 2.1.

This problem originated in the database community and was studied intensively in the context of relational databases, for example in [5, 7, 15]. Nonetheless, it reappears in many other contexts such as data synchronisation [10], XML transformations [4, 19] and configuration APIs. In the particular case of Augeas and Elektra the concrete source is a configuration file and the abstract view is a tree[1]. After the tree was modified it must

[1]Strictly speaking the abstract view of Elektra is a `KeySet` consisting of `Keys`, but conceptually the `Keys` represent a tree.

$$\begin{tikzcd}
s \arrow[r, "q"] \arrow[d, dashed, "t"] & v \arrow[d, "u"] \\
s' \arrow[r, "q"] & v'
\end{tikzcd}$$

Figure 2.1: Visualisation of the view update problem, based on the figure in [14]

be converted back to a configuration file. The difference compared to databases is that databases use the same format for original data source and views (i.e. relations in both cases).

The translation of the update u can happen in two different ways. The first way is to translate the update operations done on v to corresponding update operations on s. For example, an insertion of a new Elektra `Key` could be translated to the insertion of a new line in the corresponding configuration file.

The second possibility is to translate complete states. This means that the updated source s' is completely rebuilt on the basis of v'. All technologies presented in this chapter apply the second approach. First, implementing state translation usually is simpler than translating updates. Translating updates requires a formal description of update operations on both sides and their correlation. As an example consider an Elektra `KeySet` with newly inserted and deleted `Keys`. Simply writing a new file containing the contents of the `KeySet` is simpler than calculating all the updates that must be done on the old file. Calculating such a changeset is especially complex because updates to an Elektra backend are represented by a new `KeySet` instead of a series of operations to be performed. Interestingly this was not always the case. Version 0.7 of Elektra used a combination of the first and the second approach. While most modifications were handled by translating the complete state, deletions were handled differently. Deleted `Keys` were still present in the updated `KeySet`, but marked for deletion. Storage plugins had to make sure that the delete operation is correctly applied to the affected parts in the configuration file.

2.2 Solving the View-update Problem

Solutions to the view-update problem can be categorised by the design of the transformation functions. The following sections will explain the different possibilities.

2.2.1 Independent View and Update Procedures

Independent view and update procedures means that the function calculating the view and the function calculating the update on the original source do not have a formal correlation.

This approach is chosen by existing Elektra storage plugins. Transforming a configuration file into an Elektra tree and converting the modified tree back to the file are two independent actions. As an example consider the *ini* storage plugin. The view function q corresponds to the function `elektraIniGet`. Calculating the update that has to be done on the original INI file in order to retrieve s' corresponds to the function `elektraIniSet`. These two functions are independent from each other and are only related by the intuitive expectations of their behaviour. One of these expectations is that updates done on the Elektra tree are correctly propagated to the underlying configuration. Indeed, the plugin makes sure that the file written in the function `elektraIniSet` is a valid INI file and that all transformations done to the Elektra tree are reflected in the modified file. Otherwise the plugin would no longer solve the view-update problem for INI files because $q(s') \neq v'$.

However, sometimes adhering to this specification is not a trivial task. For example, in the case of the INI format the plugin has to restore comments with the correct comment syntax. Some INI variants allow comments starting with `;` as well as `#`. If a comment was started with `;`, the resulting Elektra `Key` (or `MetaKey`) must be written back with `;` in the write direction. However, if the comment was started with `#` instead, the plugin has to remember this fact and use `#` in the write direction. Therefore the plugin has to remember for each comment in the Elektra format which character was initially used in the INI format. Another example is the handling of whitespaces. Even if `elektraIniGet` hides whitespaces from the resulting `KeySet`, `elektraIniSet` should restore them properly.

Although these expectations are well known, crafting a storage plugin that adheres to all these requirements is hard. This claim is supported by the fact, that at the time of this writing no storage plugin is able to fully restore all of the initial formatting information such as whitespaces at any places.

The advantage of using independent view and update procedures is high expressiveness. Because no law dictates how exactly the two functions correlate, many implementations for each of the functions are possible. But again, this implementation freedom comes at a cost. Not only must the plugin be written properly in the first place, but changes to one translation function must be reflected in the other one. Both directions must be kept in mind at all times. As an example consider a change to the syntax of sections in `elektraIniGet`. If the syntax of sections was changed from `[section]` to `<section>`, the `elektraIniSet` function would have to be changed as well. Otherwise it would produce files with `[section]` sections that could not be read by `elektraIniGet` anymore.

2.3 Bidirectional Transformations

In contrast to the approach above, bidirectional transformations describe both transformation directions in a single specification. One can think of being able to run one and the same program in forward and in backward direction. When run in forward direction the program transforms the concrete source into an abstract view. When run in backward direction the program transforms an updated view back to the corresponding source representation. Different formalisms exist that aid in the design of such specifications. Constructing these specifications is called bidirectional programming.

This approach was chosen for Augeas. In particular, Augeas uses so called lenses that incorporate this principle of combining both transformations in a single specification. For Augeas to handle configuration files, it is sufficient to provide a single description of the configuration file format. From this description, a function mapping the configuration to a tree, and a function mapping the modified tree back to a modified configuration file, is constructed automatically.

2.4 Bidirectional Programming with Lenses

Foster, Greenwald, Moore, Pierce and Schmitt initially proposed lenses in [13] to tackle the view-update problem for tree structured data. However, the concept can be used to transform between completely different data types as well. A lens is just the packaging of everything that is needed to do a complete round trip: from a source representation to the abstract view and from the updated view back to the modified source. Formally a lens can be described as follows [14]:

Definition. *Let $\mathcal{U}$ be a universe of objects, $S \subseteq \mathcal{U}$ be a set of concrete sources and $V \subseteq \mathcal{U}$ be a set of views. Then a basic lens consists of three functions, namely* get*,* put *and* create*:*

$$\begin{aligned} l.get &\in S \rightarrow V \\ l.put &\in V \times S \rightarrow S \\ l.create &\in V \rightarrow S \end{aligned}$$

Put and *Create* are very similar, except that *Create* constructs a completely new source from the given view, instead of updating an existing source. This additional function is needed, if no concrete source exists yet that can be updated (e.g. if a new element is created in the abstract view representation). The additional source argument in the put function makes it possible for the get function to remove details from the source while building the view. These details can be restored in the put function only with the help of this additional argument. For example, this allows lenses to remove whitespaces in the abstract view, but properly restore them when transforming the view back to the concrete source.

In order to formalise the intuitive expectations of transformation behaviour, three rules suffice that correlate the lens functions [14]:

Definition. *Let $s \in S$ and $v \in V$, then the lens functions must fulfil the following laws*

$$l.put\,(l.get\,s)\,s = s \qquad \text{GetPut}$$
$$l.get\,(l.put\,v\,s) = v \qquad \text{PutGet}$$
$$l.get\,(l.create\,v) = v \qquad \text{CreateGet}$$

Simply put, the PutGet as well as the CreateGet laws guarantee that all information available in the view is actually propagated to the source. This means that no modifications done to the view are lost when the updated source is generated. The GetPut law makes sure that the details abstracted away by the get function are correctly restored by the put function.

Lenses can be classified into two different categories. Lens primitives are the basic building blocks of a bidirectional program and do the actual transformation work. Lens combinators combine other lenses (i.e. lens primitives or lenses resulting from other combinators) to new lenses. As shown above, lenses have to specify what happens in both transformation directions so there is nothing gained yet. But once a hand full of lens primitives and lens combinators have been established, simple lenses can be combined to form complex transformations without caring about their implementation details. Especially a user of lenses does not have to know the details of both transformation directions anymore. The lens laws guarantee that if a transformation can be described with a lens (or combinations thereof) both transformation directions are valid. Due to the clean formalisation of lenses, the adherence to the lens laws can be proved for the initial lenses. The result is a framework which allows the user to describe one of the two directions (usually the get direction) and get the other direction for free.

CHAPTER 3

The Augeas Storage Backend

3.1 The Augeas Library

Augeas is a library for reading and manipulating configuration files in a uniform way. The idea behind Augeas is similar to the idea of Elektra, but Augeas' focus is mainly on reading and manipulating files instead of the whole stack of problems related to dealing with application configurations. Augeas makes use of lenses to solve the problems explained in Section 2.2. The simplicity of parsing files with the help of lenses is reflected by the number of available lenses for Augeas. At the time of this writing Augeas is bundled with 180 lenses. This seems to make it a perfect fit to be integrated in a storage plugin for Elektra.

3.1.1 Lenses in Augeas

Augeas reads the lenses from so called schema files. These are specifications written in a *DSL (= Domain Specific Language)* that is a subset of ML [17]. Apart from defining lenses, this DSL allows the use of variables, functions and some other useful programming constructs such as unit tests. The specified lenses are able to convert between strings as concrete sources and trees as views. In order to take advantage of all the existing lenses, the Augeas DSL was not modified in any way. Instead, the implemented storage plugin uses the Augeas API to parse files with the existing lenses.

Each node consists of a label, a value and a list of subtrees. The list of subtrees is ordered. Although the tree labels do not have to be unique, the API appends a unique number in order to allow easier access.

There are several atomic lenses built into Augeas. The atomic lenses are parameterised via a regular language (i.e. a regular expression). They take a regular expression which they use to match parts of the configuration file. Atomic lenses can only be used to gather different information pieces of a tree. Atomic lenses do not actually create the tree itself. The tree creation is handled by the so called *subtree* combinator. The subtree combinator combines all the information gathered by its contained atomic lenses and builds a new subtree out of the matched information. The subtree combinator is a special form of a lens combinator.

Combinators are used to combine lenses to more complex lenses. For example, the concatenation combinator can be used to build a single lens that represents the concatenation of two smaller lenses. Another example is the repetition combinator that allows to repeat a lens several times. The whole idea of using atomic building blocks and combining them with combinators is based on the idea of constructing parsers in a modular way, as shown in [8, 12].

Consider the following example:[1]

Listing 3.1: Augeas example schema

```
module Example =
  let key_lens = key /[A-Za-z0-9]+/
  let value_lens = store /[A-Za-z0-9]+/
  let equal_lens = del "=" "="
  let entry_lens = [ key_lens . equal_lens . value_lens ]
  let lines_lens = ( entry_lens . del "\n" "\n" )*
  let xfm = transform lines_lens (incl "/etc/example")
```

The variable `key_lens` holds a lens that makes the matched text available as the label of the subtree it will be used with. The `value_lens` does the same for the tree value.

The variable `equal_lens` contains a primitive lens that hides a found equation sign from the tree, but restores it as soon as the tree is converted back to a configuration file. If a tree is newly created, the `equal_lens` also uses an equation sign as the default character. This fact is represented by the second argument which resembles the parameter for the lens `Create` function. Note that the default character must be matched by the regular expression `"="` used as the first argument. Otherwise the lens would violate the *CreateGet* law. For example, `del "=" ":"` would not be a valid lens.

The variable `entry_lens` holds a subtree combinator (indicated by the square brackets). It takes another lens as a parameter and is responsible for creating a tree. The tree properties, i.e. label, value and subtrees, are the ones provided by the inner lens (for example by the key primitive lens). In the example the combinator builds a new tree from the concatenation of `key_lens`, `equal_lens` and `value_lens`. Keep in mind

[1]A schema may consist of other parts such as unit tests too. They were omitted for simplicity.

that the concatenation is again a single combinator lens (i.e. results in a single new lens) and is therefore a valid parameter for the subtree lens.

The variable `lines_lens` holds a lens that is able to match several lines by repeating the concatenation of two smaller lenses zero or more times. The concatenation concatenates the `entry_lens` and a lens that matches and hides newlines and therefore is able to match multiple whole lines containing an entry each.

Applied on the string `k1=v1` the `entry_lens` would yield a tree with the key `k1` and the value `v1`. If the tree was modified to contain the value `v2` and converted back to a string, it would yield `k1=v2`.

Finally, the `transform` function tells Augeas to use the lens `lines_lens` to transform the file `/etc/example`. Therefore the order of the lens definitions in the schema is irrelevant. Only the lens supplied to the `transform` function is used to parse the file. All other lenses are used to construct this final lens.

3.1.2 Aligning Pieces

The alignment problem is about matching parts of the abstract view with corresponding parts of the concrete source. This is important if details from the source are hidden during the conversion to the view. When the view is converted back to the source, these details must be reinserted at the correct place. Otherwise the view-update problem had not been solved correctly. As an example, consider the following snippet of an OpenSSH server configuration. The alignment problem occurs when comment keys need to be related to configuration directives.

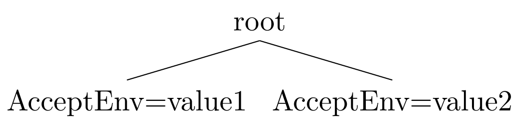
```
AcceptEnv value1 # line comment 1
AcceptEnv value2 # line comment 2
```

The same key (`AcceptEnv`) is allowed to reoccur multiple times. Suppose that the line comments should be abstracted away and therefore are not visible in the abstract view. Different options exist on how a lens achieving this goal could look like. A straight forward lens definition could be

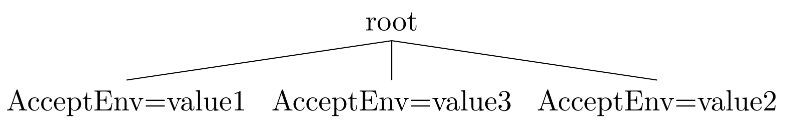
Listing 3.2: Naive lens

```
let del_comment = del /([ \t]+#.*|[ \t]*)\n/ "\n"
let env = store /[A-Za-z0-9]+/
let accept_env = [ key "AcceptEnv" . Util.del_ws_spc . env .
   ↪ del_comment ]
let lines = accept_env*
```

The lens `lines` in Listing 3.2 leads to the following Augeas tree:



What should happen if a node with value `value3` is inserted between the first and the second `AcceptEnv`?



Enumerated in Augeas, the tree with the inserted node would look like the following:

```
/path/to/file/AcceptEnv[1] = "value1"
/path/to/file/AcceptEnv[2] = "value3"
/path/to/file/AcceptEnv[3] = "value2"
```

The lens in Listing 3.2 would produce the following result in the configuration file:

```
AcceptEnv value1 # line comment 1
AcceptEnv value3 # line comment 2
AcceptEnv value2
```

Clearly, a more preferable solution would be

```
AcceptEnv value1 # line comment 1
AcceptEnv value3
AcceptEnv value2 # line comment 2
```

The reason for this behaviour is due to the way Augeas handles the alignment problem. It follows the same approach as Boomerang [1]. During the conversion from abstract view to concrete source, Augeas first builds a skeleton from all the deleted chunks and leaves holes for the content actually present in the abstract view. Afterwards these holes are filled with the corresponding subtrees. Subtrees are matched to holes in two different ways. If the tree node labels are unique, the subtrees can be assigned to the holes by using their labels. If the labels are ambiguous, Augeas falls back to the order of the tree nodes. Conceptually the updated tree produces the following skeleton:

```
<hole for first node with label AcceptEnv> # line comment 1
<hole for second node with label AcceptEnv> # line comment 2
<hole for third node with label AcceptEnv>
```

Due to the ambiguous labels, the order identifies nodes. This means that after the insertion the node with value `value2` has become the third node with label `AcceptEnv`.

To avoid the assignment by order, the `AcceptEnv` token could be deleted and a unique identifier could be used instead. For that reason Augeas provides the `seq` lens, which works like a database sequence.

Listing 3.3: Lens with sequence

```
let del_comment = del /([ \t]+#.*|[ \t]*)\n/ "\n"
let env = store /[A-Za-z0-9]+/
let newidentifier = seq "acceptenvseq" . del "AcceptEnv"
   ↪ "AcceptEnv"
let accept_env = [ newidentifier . Util.del_ws_spc . env .
   ↪ del_comment ]
let lines = accept_env*
```

Now, the nodes resulting from `AcceptEnv` would be assigned unique names generated by the sequence `acceptenvseq`. With the lens in Listing 3.3 the resulting tree would look like the following:

$$\begin{tikzcd}
& \text{root} \arrow[dl, dash] \arrow[dr, dash] & \\
1=\text{value1} & & 2=\text{value2}
\end{tikzcd}$$

If we now insert a node named `3` with value `value3` between the first and the second node

$$\begin{tikzcd}
& \text{root} \arrow[dl, dash] \arrow[d, dash] \arrow[dr, dash] & \\
1=\text{value1} & 3=\text{value3} & 2=\text{value2}
\end{tikzcd}$$

Enumerated in Augeas, the tree with the inserted node would look like the following:

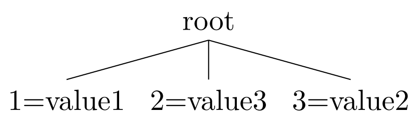
```
/path/to/file/1 = "value1"
/path/to/file/3 = "value3"
/path/to/file/2 = "value2"
```

Augeas would still be able to align the uniquely identified nodes `1` and `2` with their position and the resulting file would look like desired:

```
AcceptEnv value1 # line comment 1
AcceptEnv value3
AcceptEnv value2 # line comment 2
```

Note that, unlike Elektra, Augeas does not sort nodes by their name in the data structure, but by their implicit order. For that reason Augeas allows to explicitly insert nodes before or after another node. Even though the label `3` is alphabetically greater than the label `2` it can be inserted between `1` and `2`.

The numbers generated by the `seq` lens as names are not important because of the resulting order, but because of their ability to uniquely identify nodes. If the generated file is reloaded in Augeas the `seq` lens would assign new numbers to the nodes while still retaining the order of the file:



Enumerated in Augeas, the tree would look like the following:

```
/path/to/file/1 = "value1"
/path/to/file/2 = "value3"
/path/to/file/3 = "value2"
```

3.2 Integrating Augeas with Elektra

If the Augeas API is used, the result of reading configuration files is the Augeas tree structure. Although both of them use a tree as their abstract view, it turned out that many details had to be considered. The Augeas tree structure is similar to the tree built from Elektra `Keys`, but it still has some important differences though.

3.2.1 Recreating the Augeas Tree

When the augeas plugin is asked to write a `KeySet` back to the configuration file it must convert the `KeySet` back to the corresponding Augeas tree. The attentive reader may have noticed that this is another occurrence of the view-update problem. This time the Augeas tree is the concrete source and the Elektra `KeySet` is the abstract view. As we have learned in Section 2.1, the handling of updates can happen either by transforming the update operations or by transforming states. Transforming states would mean to wipe out the Augeas tree and create a new one from the modified Elektra tree. The first version of the augeas plugin used exactly this approach. It turned out, however, that this does not work because of Augeas internals. Some Augeas lenses create tree nodes with `NULL` labels that are hidden from the tree. These are used to tackle different aspects of the alignment problem (see Section 3.1.2 and [6] for more details on why `NULL` nodes are used). Unfortunately, these invisible nodes cannot be accessed via the public Augeas API. Although the hidden nodes are not deleted by public API calls, their position is mangled indirectly as soon as sibling nodes are deleted and recreated elsewhere. As an example consider the hosts lens. The lens uses the following line in order to preserve empty lines and their position:

```
let empty = [ del /[ \t]*#?[ \t]*\n/ "\n" ]
```

Each empty line results in a new tree node (squared brackets), but is completely hidden from the tree (inner `del` lens). The subtree lens contains no `key` or `label` lens and therefore has a `NULL` label. These `NULL` nodes are contained in the tree between other nodes acting as visible nodes. If all visible nodes are deleted, the hidden nodes concentrate at the beginning of their parent node. All newly created nodes are inserted behind them. This leads to a resulting file where all empty lines are moved to the beginning.

For that reason a different approach was needed. The augeas plugin now performs the update of the Augeas tree in two phases. First, the value of all Augeas nodes with a corresponding `Key` in the Elektra `KeySet` is updated. These nodes are known to still exist because otherwise they would have been deleted from the Elektra `KeySet`. Next the plugin cleans up all the Augeas nodes without corresponding Elektra `Keys` by iterating over all the Augeas nodes and deleting them if no corresponding Elektra `Key` is found.

3.2.2 Key Ordering

One of the differences between the Augeas tree and the Elektra tree is that the children of an Augeas tree are implicitly ordered by the data structure holding the child trees. In contrast to that, Elektra `KeySets` are ordered by the `Key` names. This means that the original ordering information of the Augeas tree must be preserved somehow. Otherwise the original ordering would be lost by the abstraction and – even worse – the order of the configuration keys would be changed as soon as the Elektra tree is written back. For most applications the order of their configuration is a formatting issue, but for others changing the order may break application behaviour e.g. access-control-lists. This problem was also encountered during the development of the hosts plugin and is described in detail in [16]. The approach chosen for most of the existing storage plugins is using `MetaKeys`. `MetaKeys` are Elektra `Keys` attached to common `Keys`. They contain meta information about the `Key` they are attached to. Ordering can be one of these meta information. Each `Key` that should have a non default ordering[2] simply has a `MetaKey` describing its order. For example, in the hosts plugin the `Key` for first host entry is given an order `MetaKey` with the value `1`, the `Key` for the second host entry an order `MetaKey` with the value `2` and so on. This approach is reapplied for the augeas plugin. When the Elektra `KeySet` is constructed from Augeas the Elektra `Keys` are numbered consecutively according to their order in the Augeas tree.

[2]Currently, Elektra `Keys` are ordered lexically by their `Key` name.

3.2.3 Key Naming

The path of each Augeas tree node can be mostly reused as the name of the corresponding Elektra `Key`. Elektra `Key` names have to be unique. Although tree nodes in Augeas are not required to have a unique name, their name is made unique by the API. If multiple nodes with the same path exist, the API appends the value of a counter to each node name. However, the path names still had to be adjusted. Augeas makes the filename where the configuration is read from a part of the node path. This part is replaced with the path of the parent `Key` of the Elektra `KeySet`. The parent `Key` is just the uppermost `Key` in the constructed Elektra `KeySet` and given in advance. This way, Elektras ability to abstract over used files could be retained. The write direction does not pose any problem because Elektra prohibits `Keys` with the same name anyway[3].

3.2.4 Treatment of Comments

Comments are represented by Augeas as common tree nodes. This is in contrast to Elektra that handles comments preferably as `MetaKeys`. Converting Augeas comment nodes to Elektra `Keys` would definitely not fit into the Elektra concepts. Fortunately comments have a well known label in the Augeas tree. They are named `#comment` by convention. This gives the augeas plugin a chance to detect these special nodes and convert them to `MetaKeys`.

However, comments may have different semantics, e.g. commented example configurations vs. explanations of configuration keys. Therefore it might not always be useful to convert Augeas comment keys to a fixed named Elektra `MetaKey`. Thus the idea came up to make the conversion of Augeas comment keys configurable.

As it turned out, the need to convert specific `Keys` to `MetaKeys` emerged at different places in Elektra. For that reason a plugin particularly suited for this task, called *keytometa*, was developed. The functionality was split into a new plugin in order to stick with the principle of single responsibility. The KEYTOMETA PLUGIN is just another plugin in the backend of the augeas plugin. It converts `Keys` into `MetaKeys` during the get direction and reverses the conversion during the set direction. The `Keys` to be converted are identified by `MetaKeys` attached to them. These `MetaKeys` control several aspects of the conversion, such as the name of the target `MetaKey`. For example, the GLOB PLUGIN [16] can be used in order to attach this conversion control `MetaKeys` to the target keys.

[3]Actually, the appending of `Keys` with the same name does not cause any error, but the new `Key` simply overwrites the old one.

CHAPTER 4

Evaluation

The following sections deal with the question how the augeas plugin compares to Elektra plugins specialized for one specific file format. In the following text these Elektra plugins will be referred to as CONVENTIONAL ELEKTRA PLUGINS. At first, a feature comparison is given, explaining in detail the strengths and weaknesses of the augeas plugin implementation compared to conventional Elektra plugins. In addition, a benchmark was performed to measure how resource intensive the augeas plugin implementation is.

It should be noted that the following comparisons are not a comparison between Elektra and the Augeas library itself, but a comparison between conventional Elektra plugins and the Elektra plugin that is based on the Augeas library.

4.1 Feature Comparison

The following section takes a closer look at how conventional Elektra plugins and the augeas plugin cope with different kinds of problems occurring while parsing configuration files. It is clear that the comparison heavily depends on the Elektra plugin in question and the Augeas lens used in the augeas plugin respectively. However, some problem areas reoccur in every conventional Elektra plugin as well as in the augeas plugin and can be explained in general. Apart from the comparison with the augeas plugin such an exploration of problem areas also helps to assess and understand the current abilities of different Elektra plugins.

4.1.1 Handling of Comments

In Section 3.2.3 we see that the augeas plugin makes use of the keytometa plugin to convert the created comment `Keys` to `MetaKeys`. Specifically this means that all `Keys` named `#comment` created by the augeas plugin are converted to `comment` `MetaKeys` by the keytometa plugin. However, this strategy causes two different problems:

Statefulness of Keytometa

The current keytometa plugin implementation is stateful. This means that during the `set` direction only those `MetaKeys` are converted back to common `Keys` that were originally created by the keytometa plugin. The reason for that is that the keytometa plugin needs to retain information from the original `Key` that cannot be easily saved to a `MetaKey`. For example, Elektra `MetaKeys` do not support `MetaKeys` themself. This would cause the `MetaKeys` of the original `Key` to be lost because there is no place to save them. As an example consider the following `KeySet` created by the augeas plugin configured with the hosts lens:

Listing 4.1: KeyToMeta comment problem

```
system/augeashosts
system/augeashosts/#comment[1]
system/augeashosts/localhost
system/augeashosts/otherhost
```

Ideally the comment `Key` `#comment[1]` should be converted to a `comment` `MetaKey` linked to the the `Key` `system/augeashosts/localhost`. However, if the comment `Key` has `MetaKeys` itself (e.g. `order`) these `MetaKeys` would be lost during the conversion.

The current solution for this problem is to store the original version of converted `Keys` in the plugin state of the keytometa plugin. The plugin state is a storage area available to each plugin that is managed by Elektra [16]. During the `set` direction of the keytometa plugin the original `Keys` are restored from the plugin state. However, this means that manually created `MetaKeys` would not be converted to common `Keys` because no original version is found in the plugin state. For example, a comment manually added to the key `system/augeashosts/otherhost` would not be converted to a `#comment` `Key`.

Correlating Comment MetaKeys

The second problem is about linking the `MetaKeys` created by the keytometa plugin to the correct `Keys`. The keytometa plugin provides several different options to configure the correlation of `MetaKeys` to their corresponding `Key`. The correlation has to be configured with special conversion `MetaKeys` for each `Key` to be converted. This can be done in bulk by using the glob plugin. The combination of the augeas plugin, the keytometa plugin and the glob plugin will be called the AUGEAS BACKEND. The contract of the augeas plugin already provides a default configuration for the glob plugin:

Listing 4.2: Glob default configuration for the augeas plugin

```
/get/#1" = "*#comment*"
      Metakeys:
      "convert/metaname" = "comment"
      "convert/append" = "next"
      "convert/append/samelevel" = 1

/get/#1/flags" = 0
```

The configuration in Listing 4.2 causes the glob plugin to add three different `MetaKeys` to each `Key` that matches strings containing `#comment`:

- `convert/metaname` specifies that each matching `Key` should be converted to a `MetaKey` named `comment`
- `convert/append` specifies that the resulting `MetaKey` should be appended to the subsequent `Key` in the `KeySet`.
- `convert/samelevel` option guarantees that comment `MetaKeys` stay within their block

Especially the last option might be confusing at first. To better understand this option consider the following hosts configuration file:

Listing 4.3: Simple hosts file with comments

```
# line comment
127.0.0.1 localhost
192.168.0.1 testhost alias1 # inline comment
192.168.0.2 anothertesthost
```

This file would result in the following `KeySet` when read with the augeas plugin configured to use the hosts lens:

Listing 4.4: Simple hosts file with comments in Elektra

```
system/hostssimple
system/hostssimple/#comment
system/hostssimple/1
system/hostssimple/2
system/hostssimple/3
system/hostssimple/1/ipaddr
system/hostssimple/1/canonical
system/hostssimple/2/ipaddr
```

```
system/hostssimple/2/canonical
system/hostssimple/2/alias
system/hostssimple/2/#comment
system/hostssimple/3/ipaddr
system/hostssimple/3/canonical
```

The glob plugin would now add the described conversion `MetaKeys` to `system/hostssimple/#comment` and `system/hostssimple/2/#comment`. The keytometa plugin in turn would convert these two `Keys` to `MetaKeys`.

Without the `samelevel` configuration the `MetaKey` resulting from `system/hostssimple/2/#comment` would simply be added to the next `Key` in the `KeySet`. The next key would be `system/hostssimple/3/ipaddr` which belongs to another host entry.

With the `samelevel` configuration the keytometa plugin tries to find a receiving `Key` on the same hierarchy level. If no such key is found (as is the case in this example) the `MetaKey` is added to the parent `Key` of the converted `Keys` hierarchy. In this case the parent `Key` of the hierarchy is `system/hostssimple/2` which represents the `testhost` entry.

This simple combination of the augeas plugin, the glob plugin and the keytometa plugin works well for many configuration files. However, the problem with this automated approach is that comments not always semantically belong to the subsequent `Key` in the `KeySet`. As an example consider introductory comments explaining the structure of the configuration file. Such comments would semantically belong to the parent `Key` of the `KeySet`. However, the above configuration would cause them to be appended to the first ordinary `Key` succeeding the converted comment `Keys`.

An example of a configuration file where this undesired behaviour can be observed is the Apache configuration as shipped with the Debian distribution[1]. The file contains many explanatory introductory comments. The only convention in place is a blank line between explanatory comments and comments belonging to a directive. The augeas backend causes the comment `MetaKeys` generated from these explanatory comments to be appended to the `Key` representing the first directive in the configuration file.

In the case of the Apache configuration file this is the `Lockfile` directive. This behaviour causes the comment `MetaKey` of the resulting Elektra `Lockfile Key` to become 64 lines long. The result is not very intuitive either as the user would most likely not expect the explanatory comments there.

[1]package version `2.2.22-13+deb7u6`

Currently, the augeas backend cannot deal with conventions consisting of such specific details. Augeas represents each comment line as a `#comment` node, regardless whether the comment is a general explanation or belongs to a specific directive. Handling these details would require a specialized lens for the use in the augeas plugin. The lens would need to incorporate the convention of a separating blank between general comments and comments belonging to one directive. Otherwise the keytometa plugin merges the comment lines like in the case of the `Lockfile` directive.

In contrast to that, such conventions could be handled by fine-grained rules built into conventional Elektra plugins. For example, general comments could be attached to the parent key while specific comments are attached to the `Key` they belong to. However, at the time of this writing no plugin did so.

Different kinds of comments

Some configuration files allow different syntax variants for comments. An example for such a configuration file is the Samba configuration. Comments in the Samba configuration file may use `#` as well as `;` as the start symbol. Conventionally comments starting with `#` are used for explanatory comments while comments starting with `;` are used for disabling directives. In addition, some configuration files allow inline comments. That are comments written in the same line as a configuration directive.

Augeas correctly restores both inline comments and the character (e.g. `#` vs `;`) that was originally used to start a comment. Otherwise, the *GetPut* law would be violated. While comparing this behaviour with existing plugins we noticed that conventional Elektra plugins did not restore comments to their original representation (if they handled comments at all). Especially the comment start character was not handled by any conventional Elektra plugin. At this time only the flat `comment MetaKey` was specified and documented. The `comment MetaKey` contained only the comment itself, but no information about its original representation. Therefore conventional Elektra plugins were not able to restore the original comment representation.

For that reason a new specification for comment `MetaKeys` was created. In addition to restoring the original comment start character this model allows to also capture and restore the exact formatting of the comment. This way the specification tackles two common problems of existing Elektra plugins.

The new specification models comments as `MetaKey` arrays instead of a single `MetaKey`. The first array entry is the inline comment for the `Key` (if any) and the remaining entries contain the comments preceding the configuration directive. If no inline comment is present the first array entry stays empty. Further, each array entry can have two subkeys

- the character used to start the comment, if any (e.g. `#` vs. `;`)

- the number of spaces preceding the start character

The specification was in turn implemented in a small library and used to realize proper comment parsing in the HOSTS PLUGIN. However, while the library significantly eases the correct implementation of the specification, parsing all the required information is still a complex task. It took about 1200 inserted lines of code, 700 deleted lines of code and several hours of work to implement the specification in the hosts plugin. Unfortunately, the augeas backend currently does not support the specification. The augeas backend still produces a single `comment MetaKey`. However, as the augeas backend uses Augeas to create the resulting file, the correct comment start character and formatting is restored anyway.

4.1.2 Preserving File Appearance

Configuration files often contain lots of formatting such as blanks, tabs and newlines. Some of this information is syntactically or semantically required. For example, the Apache configuration file may contain only one directive per line (although one directive may stretch several lines). Each Elektra plugin and each Augeas lens has to cope with this mandatory formatting correctly. This means that the formatting must be correctly restored if it is hidden in the abstract view. Otherwise the implementation can be considered at least partially wrong because it produces invalid configuration files.

However, usually there are also optional formatting details. These details might not be essential for creating correct configuration files, but they most likely are essential for a user editing the configuration file by hand. In this regard it can be considered crucial that Elektra plugins correctly restore the original formatting details present in a configuration file.

Exploiting the Power of Bidirectional Programming

Handling formatting details is a clear strength of the augeas plugin, because Augeas goes to great lengths in order to retain them. Formatting details that should be hidden from the abstract view can simply be deleted with the `del` lens and are automatically restored when the concrete source is rebuilt. Augeas lenses also prevent that the programmer forgets formatting details because each character must be parsed by a lens anyway, otherwise reading a configuration file results in a parsing error. As an example consider the lens that matches records in the hosts file:

Listing 4.5: Hosts file record lens

```
let record = [ indent . store . sep_tab . key word .
                       [ sep_spc . key word ]* ]
```

If the `indent` lens were removed from the lens combinator, the record lens would be unable to parse host records with leading whitespaces. In turn, by adding the `indent` lens the `record` lens not only is able to parse and hide the leading whitespaces, but also to restore them when the abstract view is translated back to the configuration file.

Solving the View-update Problem by Hand

Unfortunately, the handling of formatting details is not so straight forward for conventional Elektra plugins. As described in Section 2.2 conventional Elektra plugins require both translation directions to be specified explicitly by writing code.

We saw in Section 4.1.1 that `MetaKeys` can be used to store information about the original representation comments. The same principle could be used to store information about the original representation of any other configuration directive too. As an example consider the following INI configuration line:

```
␣initkey␣=␣inivalue
```

`MetaKeys` could be used to store that the key `inikey` is preceded by a whitespace and that the = character is surrounded by whitespaces. However, at the time of this writing no Elektra plugin is able to use `MetaKeys` in order to fully restore all hidden formatting details.

4.1.3 Ordering

In Section 3.2.2 we already saw that entries in Augeas have an order and that this order can be represented in Elektra with `MetaKeys`. In the `set` direction the `order` `MetaKey` can be used to recreate the correct order in Augeas.

For many configuration files this imposed order is useful or even necessary. Other configuration files do not require a specific order. For example, in the Samba configuration file changing the order of entries only changes the appearance of the file. Configuration files where the whole file or parts of it require a specific order can be further categorized as follows.

Implicit Order

Configuration files in this category deal with order on a semantic level. As an example, consider the squid configuration file. The access-lists occurring in this configuration file require a specific order. A changed order would not just change the formatting of the configuration file, but would also cause a changed behavior of the application. The syntactic specification of the file does not impose a specific order on the access-lists. It simply describes how an access-list has to look like. A squid configuration file may contain access-lists in any order and still be syntactically valid. However, its semantic changes significantly. Therefore additional information needs to be saved in order to retain the original order. In Augeas this is done with the implicit order of child trees in the Augeas data structure. In Elektra the original order is captured with `order`

MetaKeys. For configuration files with an implicit order it would not be useful to renounce this information. Conventional Elektra plugins, as well as the augeas plugin, require the user to specify the order of keys when creating or modifying the configuration.

Explicit Order

Configuration files in this category have a syntactic specification that imposes a specific order for the whole file or some parts of it. As an example, consider the inittab configuration file used in many Unix variants. According to its specification an inittab entry has the following syntax:

```
id:runlevels:action:process
```

The order of the parts of an entry is fixed. The syntax specifically states that the identifier always has to be the first field, followed by the runlevels and so on. The following entry

```
ca:12345:ctrlaltdel:/sbin/shutdown -t1 -a -r now
```

would result in the following Augeas tree

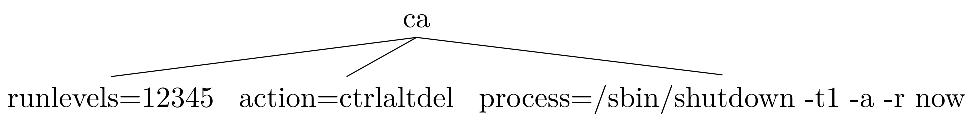


Figure 4.1: Valid inittab Augeas tree

The problem is that, although the subtrees runlevels, action and process could be uniquely assigned to their corresponding position in the configuration file just by their name, Augeas is not able to do so. Instead it still uses the order imposed by the order of the subtree entries. As a result the following Augeas tree could not be successfully converted into an inittab configuration file:

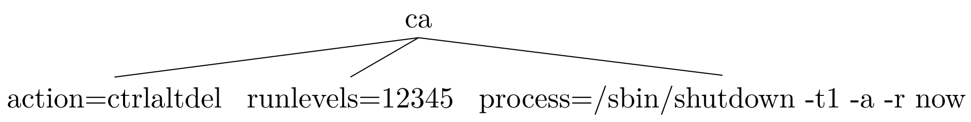


Figure 4.2: Invalid inittab Augeas tree

This problem is especially nasty when a user tries to create a new subtree. The user might know which fields constitute an inittab entry, but might not know in which order they have to appear. As a consequence it might happen that the user creates the entries with an invalid order and Augeas is not able to translate the tree into an inittab entry anymore.

In contrast, a manually crafted plugin could be able to recognize the syntactical correspondence between subtrees and their position in the file. For example, it could detect that the node named `runlevels` resembles the runlevels entry and therefore write its value to the second field regardless of its position in the tree. For example, this automatic ordering is implemented in the `fstab` plugin.

4.1.4 Structural Abstraction

One goal of Elektra is to abstract over the underlying configuration structure. While specialized plugins provide nearly every thinkable abstraction, the abstraction that the augeas plugin can provide is limited by the language used for describing Augeas lenses.

Some restrictions in the language are needed, because Augas uses a typechecker that should be able to statically check lenses for type-correctness. Many of the checks done by the typechecker are only possible on regular languages. For example, the typechecker needs to verify whether two concatenated lenses overlap. In this case the concatenation would be ambiguous. However, determining whether the intersection of two context free languages is empty is undecidable [11]. This means, that the lens language cannot be extended at will, because otherwise the typechecker might not be able to statically guarantee the correctness of lenses anymore.

Backreferencing in the Hosts Lens

One limitation that is encountered when trying to increase the level of abstraction is the lack of backreferencing. The rest of this section will explain an example of missing abstraction caused by the lack of backreferencing.

In the context of regular expressions used in practice, backreferences allow us to define expressions that match only those words that were already matched by an earlier subexpression. Usually the referenced expressions have to be enclosed in parenthesis and are named with numbers starting from left to right. A more thorough definition can be found in [2]. For example, consider the following regular expression containing backreferences: `([ab])c\1`. This regular expression would match the words `aca` and `bcb`, but not `acb`, `bca` or `acd`.

The regular expressions used in Augeas lenses do not support such backreferences. The reason is that the introduction of backreferences causes the described language not to be regular anymore. This in turn would render the typechecker unable to statically guarantee the correctness of lenses making use of backreferences. A detailed description of this problem can be found in [3].

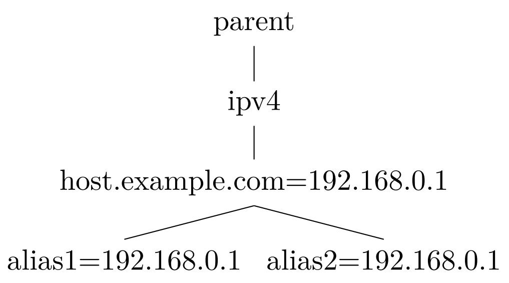
It should be noted that Augeas indeed supports some extensions to strict regular languages. For example, lenses may be used recursively. The resulting expressions are not regular anymore. As a result the typechecker can typecheck these extended expressions only heuristically and issues with invalid lenses may arise during runtime.

Unfortunately, the absence of backreferences makes some desired abstractions impossible. As an example we will consider a hosts file and how it can be represented in Elektra and Augeas.

Listing 4.6: Simple hosts file

```
192.168.0.1 host.example.com alias1 alias2
```

At the time of this writing the Elektra hosts plugin would generate the following Elektra `KeySet` (represented as a tree):



As can be seen, the `alias` nodes also contain the IP address of the canonical entry as value. Although the IP address of an entry exists only once in the hosts file the hosts plugin parser is able to reuse it multiple times. In order to build a comparable tree in Augeas, backreferences would be needed. An example of how the required lens could look like is shown in Listing 4.1.4. However, note that Augeas would not be able to use such a lens because of the backreference.

Listing 4.7: Lens with backreferences

```
let sep_tab = Util.del_ws_tab
let sep_spc = Util.del_ws_spc
let word = /[^# \n\t\/]+/
let record = [ indent . store (word) . sep_tab . key word .
                         [ sep_spc . key word . store \1 ]*
             . comment_or_eol ]
```

The backreference used in the alias subtree resembles the reuse of the already parsed IP address.

4.1.5 Discussion

We saw in the feature comparison that the augeas plugin handles especially formatting related issues very well. The augeas plugin could be used to uncover several shortcomings of conventional Elektra plugins. However, its general nature sometimes causes inconveniences. We also saw that some of the issues of the augeas plugin can only be solved by modifying the underlying Augeas lens, but at least in principle all the existing lenses can be reused. The newly created plugin successfully bridges the gap between Augeas and Elektra.

The augeas plugin together with the existing Augeas lenses is well suited as a proof of concept for new configuration file formats, especially if no specialized Elektra plugin exists. However, if more abstraction or specific features are needed, the default lenses do not suffice anymore. In this case lenses specialized for the use in the augeas plugin are needed. If even more abstraction is required, the development of plugin specialised on a single configuration file should be considered. Therefore RQ1 can be answered positively, but RQ2 must be answered negatively.

4.2 Performance Considerations

In addition to the qualitative comparison, the runtime behaviour of Elektra with different mountpoint configurations was measured. A mountpoint is simply a combination of plugins mounted somewhere in the Elektra tree. A more detailed description of the mounting concept can be found in [16]. Considering the performance of Elektra is relevant because some applications (e.g. KDE) use very large configurations with lots of configuration files.

It should be noted, however, that the following results are neither a fully-fledged benchmark for Elektra and the augeas plugin nor for the Augeas library. The presented results focus solely on the read performance of the plugins as reading configurations is the most performance critical task when dealing with configurations.

4.2.1 Experiment setup

During the experiment three different mountpoint configurations were compared:

- a mountpoint with the specialised hosts plugin
- a mountpoint with the augeas plugin configured to use the Augeas hosts lens
- a mountpoint using the augeas backend, configured to use the Augeas hosts lens. See Section 3.2.3 for why this combination is useful.

The hosts plugin was used for the comparison because its features are most comparable to those of the augeas plugin. The augeas backend was chosen in order to analyze the impact of an additional plugin on the mountpoint.

During the experiment the wall clock time of executing `kdb ls <mountpoint>` was measured with the *time* utility. All measurements were performed by a shell script called `benchmark_augeas_hosts.sh` that executes the `kdb` command 11 times for each mountpoint. The shell script as well as the tested version of the augeas plugin and the augeas backend can be found in commit 937c19636ede8d24d384e89f2e3867a445b81b89 of the Elektra repository[2].

The mountpoints were tested with nine different hosts files. The used hosts files, as well as the benchmark results can be found in separate git repository[3].

hosts file #	lines	line comments	blank lines	host entries
1	14	3	2	9
2	2349	51	3	2295
3	4138	166	3	3969
4	6068	1528	3	4537
5	8254	896	3	7355
6	10543	1124	3	9416
7	12053	1332	3	10718
8	14184	1613	3	12568
9	15571	1748	3	13820

Table 4.1: Different sizes of hosts files used for the benchmark

hosts file `#1` resembles an ordinary hosts file as found on a default installation of the Debian distribution. Hosts file `#2` – `#9` were generated from a hosts file containing blacklist entries for filtering spam mails.

All the results were gathered on a virtual machine with 4 CPU cores and 4096 MB of RAM running in VMware® Workstation 12 Player. The virtual machine was running on a computer with an Intel® Core™ i5 CPU, 16 GB DDR3 RAM and a Samsung® SSD 850 EVO harddisk.

4.2.2 Experiment Results

In order to compensate statistical outliers, the median of the 11 runs was calculated for each configuration. Table 4.2 shows the median runtime for the different mountpoint configurations and the hosts files from Table 4.1.

[2] `http://www.libelektra.org`

[3] https://github.com/fberlakovich/bachelorthesis

hosts file #	augeas (s)	augeas with keytometa (s)	hosts (s)
1	0	0	0
2	0.70	0.80	0.02
3	2.03	2.18	0.04
4	7.89	8.11	0.05
5	8.41	8.93	0.08
6	14.28	15.11	0.10
7	20.84	22.26	0.13
8	30.31	31.91	0.15
9	37.15	39.21	0.16

Table 4.2: Median runtimes in seconds of the different mountpoint configurations

As can be seen in Table 4.2 the runtime for hosts file `#1` is negligible as it is even below the minimum runtime the `time` utility can measure.

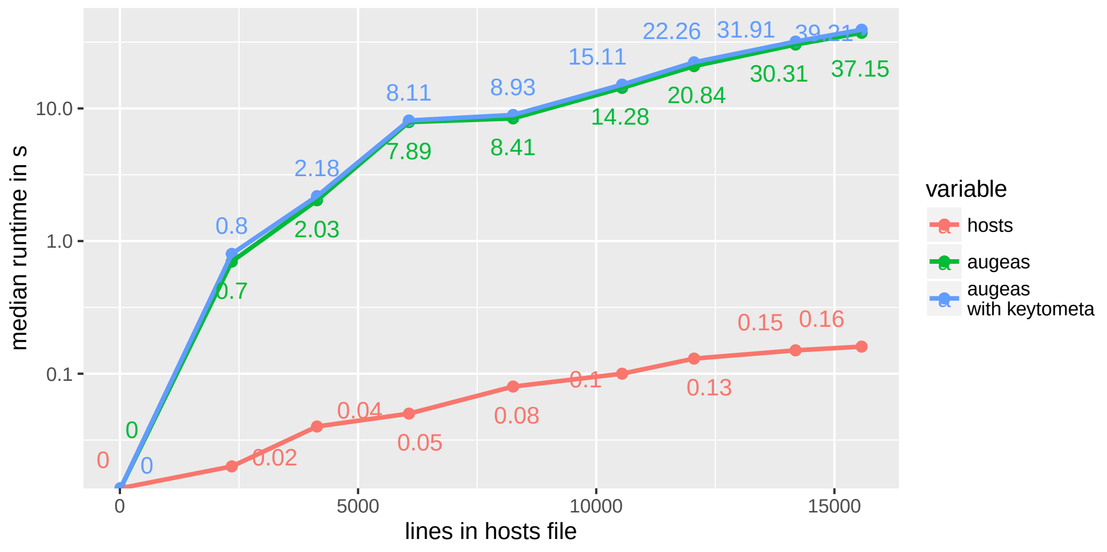


Figure 4.3: Comparison of the mountpoint configurations for differently sized hosts files with logarithmic y-axis

Figure 4.3 shows the runtime difference between the specialized hosts plugin and the augeas plugin. Note that the graph has a logarithmic y-axis because otherwise the runtime increase of the hosts plugin would hardly be perceivable. While the runtime of the hosts plugin increases nearly linearly with the hosts file size it still stays far below half a second. Event for the largest hosts file its median runtime is only 0.16 seconds. In contrast to that, the runtime values of the augeas plugin and the augeas backend suggest that their runtime increases more than linearly with bigger hosts files. While hosts file

#9 is only about 2.5 times as big as hosts file #4, the runtime for the augeas plugin and hosts file #9 is more than 4 times as long as for hosts file #4. Therefore the answer to RQ3 is that the augeas plugin is much more resource intensive than conventional Elektra plugins.

Figure 4.3 also illustrates that the addition of the keytometa plugin indeed increases the runtime in a measurable way. However, the caused runtime increase is small compared to the runtime increase caused by bigger hosts files. Furthermore, it can be avoided if not needed due to the modularity of Elektra.

4.2.3 Callgraph Analysis

In order to narrow down the part of the augeas plugin that causes the massive runtime increase the profiling tool `valgrind` [20] was used. A call to `kdb ls` for the mountpoint with the augeas plugin and hosts file #9 file was profiled and revealed the callgraph shown in Figure 4.4.

The callgraph shows only functions that cause at least 1% of the total runtime and only one level of functions outside of the augeas plugin object file. Each node corresponds to a function and each directed edge corresponds to a function call. The numbers below the function names depict how much of the total runtime was caused by the function. The labels of the directed edges show how often the target was called by the source.

The most expensive function in the augeas plugin is `foreachAugeasNode`. This function is used to construct an Elektra `KeySet` from the nodes found by Augeas. `foreachAugeasNode` first calls `aug_match`, which is used to retrieve a list of all nodes in the Augeas tree. Afterwards `convertToKey` is called for each found node which in turn calls `aug_get`. The callgraph shows that `aug_match` and `aug_get` are responsible for nearly the whole runtime. These two functions are located in the Augeas library itself. This means that the possibilities to optimise the augeas plugin without changing the Augeas library are limited.

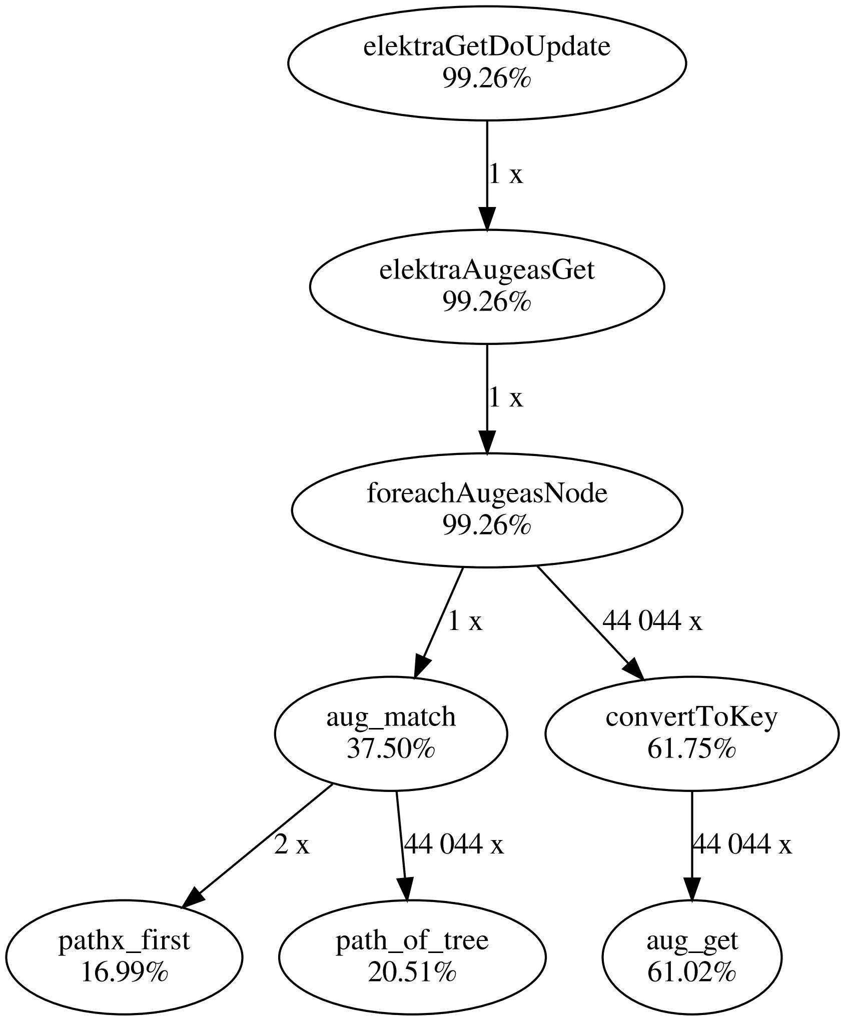


Figure 4.4: Callgraph of the most expensive function calls in the augeas plugin

CHAPTER 5

Conclusion and Future Work

After an introduction describing the current problems with the implementation of Elektra plugins we introduced the concept of bidirectional programming. We learned that a single specification, called lens, can be used to transform a concrete source into an abstract view as well as to transform a modified view back to the source representation.

Next the theoretical aspects of bidirectional programming were introduced. We saw how Augeas uses the concept of lenses to transform between configuration files and tree structures. Furthermore we learned how the Augeas library was used to build an Elektra storage plugin that makes use of all the available Augeas lenses.

Then we analyzed the implementation of the augeas plugin and compared its features to those of conventional Elektra plugins. We saw that Augeas especially excels at handling formatting details. In contrast, existing Elektra plugins do not currently hide and restore formatting details in a uniform way. A new library that restores formatting details with the help of `MetaKeys` was created to solve this problem. We also learned that, due to its generic nature, the augeas plugin sometimes cannot provide such a high level of abstraction as conventional Elektra plugins. However, users of Elektra now have a choice. They may either use the augeas plugin with its excellent handling of formatting details or they may choose a conventional Elektra plugin if a higher level of abstraction is needed.

Many of the useful concepts found in Augeas could also be used in Elektra plugins. For example, the concept of parser combinators allows the composition of complex parsers from small and simple parts as shown in [9]. This could be used to ease the development of future Elektra storage plugins.

At last we took a look at the performance of the augeas plugin. We saw that it is well suited for small configuration files, but that the runtime quickly increases when the configuration file size is increased. Section 4.2.2 revealed that most of the runtime of the augeas plugin is spent to fetch values. However, it is unlikely that all Elektra key values are needed at once. Therefore the required runtime could be delayed until the value is actually needed by supporting lazy loading of Elektra `Key` values.

Bibliography

[1] Aaron Bohannon et al. „Boomerang: Resourceful Lenses for String Data“. In: *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages.* POPL ’08. New York, NY, USA: ACM, 2008, pp. 407–419. ISBN: 978-1-59593-689-9. DOI: 10.1145/1328438.1328487. URL: http://doi.acm.org/10.1145/1328438.1328487 (visited on 03/26/2014).

[2] Alfred V. Aho. „Handbook of Theoretical Computer Science (Vol. A)“. In: ed. by Jan van Leeuwen. Cambridge, MA, USA: MIT Press, 1990, pp. 255–300. ISBN: 978-0-444-88071-0. URL: http://dl.acm.org/citation.cfm?id=114872.114877 (visited on 09/25/2015).

[3] Cezar Câmpeanu, Kai Salomaa, and Sheng Yu. „A formal study of practical regular expressions“. In: *International Journal of Foundations of Computer Science* 14.06 (Dec. 2003), pp. 1007–1018. ISSN: 0129-0541. DOI: 10.1142/S012905410300214X. URL: http://www.worldscientific.com/doi/abs/10.1142/S012905410300214X (visited on 03/06/2016).

[4] Haitao Chen and Husheng Liao. „A Survey to View Update Problem“. In: *International Journal of Computer Theory and Engineering* (2011), pp. 23–31. ISSN: 17938201. DOI: 10.7763/IJCTE.2011.V3.278. URL: http://www.ijcte.org/show-34-684-1.html (visited on 04/17/2014).

[5] Edgar Frank Codd. „Recent Investigations in Relational Data Base Systems“. In: *IBM Research Report* RJ1385 (1974).

[6] David Lutterkort. „Augeas–a configuration API“. In: *Proceedings of the Linux Symposium, Ottawa.* 2008, pp. 47–56. URL: http://www.landley.net/kdocs/ols/2008/ols2008v2-pages-47-56.pdf (visited on 03/26/2014).

[7] Umeshwar Dayal and Philip A. Bernstein. „On the Correct Translation of Update Operations on Relational Views“. In: *ACM Trans. Database Syst.* 7.3 (Sept. 1982), pp. 381–416. ISSN: 0362-5915. DOI: 10.1145/319732.319740. URL: http://doi.acm.org/10.1145/319732.319740 (visited on 04/17/2014).

[8] Jeroen Fokker. „Functional parsers“. In: *Advanced functional programming.* Springer, 1995, pp. 1–23. URL: http://link.springer.com/chapter/10.1007/3-540-59451-5_1 (visited on 08/16/2015).

[9] Richard A. Frost, Rahmatullah Hafiz, and Paul Callaghan. „Parser Combinators for Ambiguous Left-recursive Grammars“. In: *Proceedings of the 10th International Conference on Practical Aspects of Declarative Languages*. PADL'08. Berlin, Heidelberg: Springer-Verlag, 2008, pp. 167–181. ISBN: 3-540-77441-6 978-3-540-77441-9. URL: `http://dl.acm.org/citation.cfm?id=1785754.1785766` (visited on 01/16/2016).

[10] *Harmony Project home page*. URL: `https://alliance.seas.upenn.edu/~harmony/old/` (visited on 04/17/2014).

[11] M. A. Harrison. *Introduction to Formal Language Theory*. 1st. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1978. ISBN: 978-0-201-02955-0.

[12] Steve Hill. „Combinators for parsing expressions“. In: *Journal of Functional Programming* 6.03 (1996), pp. 445–464. ISSN: 1469-7653. DOI: `10.1017/S0956796800001799`. URL: `http://journals.cambridge.org/article_S0956796800001799` (visited on 08/16/2015).

[13] J. Nathan Foster et al. „Combinators for Bidirectional Tree Transformations: A Linguistic Approach to the View-update Problem“. In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 29.3 (2007). ISSN: 0164-0925. DOI: `10.1145/1232420.1232424`. URL: `http://doi.acm.org/10.1145/1232420.1232424` (visited on 03/26/2014).

[14] John Nathan Foster. „Bidirectional Programming Languages“. AAI3405376. PhD thesis. Philadelphia, PA, USA: University of Pennsylvania, 2009.

[15] A. M. Keller. „The Role of Semantics in Translating View Updates“. In: *Computer* 19.1 (1986), pp. 63–73. ISSN: 0018-9162.

[16] Markus Raab. *A Modular Approach to Configuration Storage*. de. Sept. 2010. URL: `http://www.libelektra.org/ftp/elektra/thesis.pdf` (visited on 03/26/2014).

[17] Robin Milner et al. *The Definition of Standard ML - Revised*. English. revised edition edition. Cambridge, Mass: The MIT Press, May 1997. ISBN: 978-0-262-63181-5.

[18] Patrick Sabin and Markus Raab. *Implementation of Multiple Key Databases for Shared Configuration*. 2008. URL: `ftp://www.markus-raab.org/elektra.pdf` (visited on 03/26/2014).

[19] Sławek Staworko, Iovka Boneva, and Benoît Groz. „The view update problem for XML“. In: *Proceedings of the 2010 EDBT/ICDT Workshops*. ACM, 2010, p. 20. URL: `http://dl.acm.org/citation.cfm?id=1754262` (visited on 04/15/2014).

[20] *Valgrind Home*. URL: `http://valgrind.org/` (visited on 02/20/2016).