UNIVERSITY OF TARTU
Faculty of Science and Technology
Institute of Computer Science
Computer Science Curriculum

Aron Sisask

Automatic Error Correction for Elixir

Bachelor's Thesis (9 ECTS)

Supervisor: Vesal Vojdani, PhD

Tartu 2022

Automatic Error Correction for Elixir

Abstract:
Code linting is the practice of automatically checking source code for errors, usually those related to consistency. Most programing languages have linters that come with a set of rules, each one targeting a specific kind of error. To reduce programmer effort, some linters employ autocorrect, meaning that where possible, the linter fixes the mistake for the programmer. Linters are also used in Elixir ecosystem; however, there is no linter for Elixir with autocorrect capabilities. In this thesis, autocorrect is added to one of the most frequently used Elixir linters, Credo. The developed extension can be integrated with Credo seamlessly, and it can automatically correct several types of errors.

Keywords: static analysis, linting, Elixir

CERCS: P175 Informaatika, süsteemiteooria

Automaatne vigadeparandaja Elixirile

Lühikokkuvõte:
Koodi lintimine on programmi lähtekoodist automaatne vigade otsimine eesmärgiga koodi stiili ühtlasena hoidmine. Enamikel programmeerimiskeeltel on linterid, mis realiseerivad mingi hulga reegleid: iga reegel otsib lähtekoodist mingit tüüpi kindlat viga. Programmeerija aja kokkuhoidmiseks toetavad osad linterid automaatset parandamist, mis tähendab, et võimaluse korral parandab linter vead ise ära. Lintereid kasutatakse ka Elixiri ökosüsteemis. Samas pole Elixiri jaoks automaatse parandamise võimalustega linterit. Selles lõputöös lisatakse ühele levinumale Elixiri linterile Credole automaatne parandamine. Väljatöötatud programm integreerub Credoga lihtsalt ja see suudab automaatselt parandada mitmeid tüüpi vigu.

Võtmesõnad: staatiline analüüs, lintimine, Elixir

CERCS: P175 Informatics, systems theory

Contents

1 Introduction

Elixir[1] is a functional programming language that was created by Jose Valim in 2012. The source code of an Elixir program compiles into BEAM bytecode which runs on the BEAM virtual machine, a virtual machine that is also used by Erlang programming language (Juric, 2019, p. 8). Hence, programs written in Elixir can utilize the functions and modules provided by Erlang libraries (Juric, 2019, p. 9). From syntax perspective, Elixir is most similar to Ruby.

As in most programming languages, Elixir also has static analysis tools that can be used to detect mistakes, bad programming practices and inconsistencies in the codebase. One of the most popular static analysis tools is Credo[2]. Credo has a set of checks, each of which will look for one particular problem in the codebase. When Credo is invoked, it runs all of its checks against the codebase and prints out found issues.

One of the drawbacks of Credo is the lack of autocorrection. Many of its reported issues are pretty simple and could be corrected automatically. Yet, Credo has no autocorrection for any of its checks. This leads to reduced developer productivity as software engineers need to spend time to fix the reported issues manually.

The aim of this thesis is to build a library that is capable of automatically correcting some of the issues reported by Credo. The thesis is structured as follows. The second chapter gives an overview of code analysers, Elixir, and Credo. The third chapter describes how autocorrection can be implemented. In the final chapter, the developed autocorrection feature is applied to various Elixir projects for testing purposes. Additionally, an overview of possible future work is given.

[1] https://elixir-lang.org/

[2] https://github.com/rrrene/credo

2 Background

This chapter gives an overview of code analysers in general, the Elixir programming language, and one of the most used Elixir linters Credo.

2.1 Code Analysers

Linter is a static analysis tool that looks for errors in the source code (Tómasdóttir et al., 2020). Most linters parse the source code of a program and form an abstract syntax tree. Manually defined rules are then used to look up violations of linter rules in the abstract syntax tree. Violations can be, for example, declaring an unused variable or the wrong number of rows between functions. Additionally, linters can detect some security vulnerabilities (Rafnsson et al., 2020).

According to a study by Tómasdóttir et al. (2020), linters are mainly used to keep the code style consistent. The same study found that a consistent style also makes the code more readable and understandable. Further, programmers also pointed out that linters avoid disputes over code formatting. According to Tómasdóttir et al. (2020), 62% of GitHub's 300 most popular JavaScript projects use linters.

In addition to error detection, some linters have automatic error correction functionality. For example, a linter could automatically insert the correct number of empty rows between functions in the source code. Johnson et al. (2013) found that the lack of automatic correction capabilities is one of the reasons why software developers do not use linters. From this, it can be concluded that some programmers find that the time it takes to correct the errors reported by the linter outweighs the benefits of keeping the code readable and consistent in style. Without automatic correction, it is challenging to integrate a linter into an existing software project, because in a large codebase, a very large number of errors would have to be corrected manually when adding a linter. Therefore, it is important that the linters can correct the detected errors to some extent themselves.

2.2 Overview of Elixir

The following is based on Juric (2019, p. 3–4, 8–9). Elixir is a functional programming language which was created by Jose Valim in 2012. The language is based on Erlang, a programming language created at Ericsson in the 1980s for building scalable real-time systems. Once compiled, Elixir programs run on BEAM virtual machine — the same virtual machine that is used for running Erlang programs. Some of the features that Elixir provides are:

- **Functional programming.** All variables, regardless of the data type, are immutable in Elixir. This encourages programmers to write pure functions.

- **Concurrency.** Elixir can take advantage of the Erlang/OTP libraries which provide great tooling for writing software where massive concurrency is needed.
- **Processes.** Code in Elixir runs in a process that, when compared to languages like Java and Python, could be thought of as a lightweight thread. Processes exchange information via message passing, which means that no locking is needed for synchronization. This makes multi-threaded applications easier to write, understand and reduces the probability of race conditions and deadlocks.
- **Fault tolerance.** In case one process crashes, it is completely isolated from all the other processes and can be simply restarted.

Thanks to its great concurrency model, Elixir is often used in cases where concurrency and scalability are needed. For instance, Discord uses Elixir for running its whole chat platform with more than 12 million concurrent users (Valim, 2020). Some of the other companies that use Elixir are Heroku, FarmBot, change.org and PepsiCo ("Elixir Use Cases", n.d.). In terms of popularity, Elixir ranked 31st in Stack Overflow Developer Survey of 2021, being roughly as popular as LISP, Clojure and Haskell ("Stack Overflow Developer Survey insights", 2021).

2.3 Elixir Code Examples

```
defmodule Circle.Calculator do
  def calculate_area(diameter) do
    3.14 * r * r
  end
end

defmodule MyApplication do
  def main() do
    area = Circle.Calculator.calculate_area(10)
    IO.puts("Area of the circle is #{area}")
  end
end
```

Figure 1. Code example of Elixir.

Code in Elixir is organized in modules that consist of functions. As Elixir has no concept of classes, the return value of the function is usually dependent solely on the arguments given to the function. In figure 1, there is a code example defining two modules. In the example, function `calculate_area` is defined that calculates the area of a circle, given its diameter. Note that there is no `return` keyword in Elixir, the value of the last expression is returned.

Once a function has been defined, it can be called from the same module or other modules. In the figure 1 there is a module `MyApplication` defined where the `calculate_area` function is called, its return value is assigned to the variable `area` and the result is printed out.

Elixir has the following data types (Juric, 2019, pp. 30–35):

- **String.** An UTF-8 encoded text value.
- **Integer and float.** Numerical values, like in other programming languages.
- **Tuple.** A tuple represents a set of ordered values with a fixed length. Works the same way as tuples in Python.
- **List.** Represents a set of ordered values with dynamic length. Works the same way as lists in Python.
- **Map.** A hash map data structure that contains a set of keys and their corresponding values.
- **Atom.** A literal named constant. Denoted in code by a colon and a text following it. Atoms are often used to return the status of a function call. For instance, many functions return `:ok` to indicate that the function call was successful and `:error` when it was not. Atoms are also often used as keys in maps. Atoms are similar to symbols in Ruby.

In figure 2 there are example values for each of the data types. Note that Elixir is dynamically typed language.

```
example_str = "Any text can be in a string"
example_number = 4
example_float = 3.14
example_atom = :any_text
example_tuple = {3, "text", :asd}
example_list = [5, 12, 3]
example_map = %{7 => "value", "string_key" => 3}
```

Figure 2. Example values for Elixir data types.

2.4 Abstract Syntax Trees in Elixir

We now consider an essential data structure for performing static analysis and transforming code, the abstract syntax tree (AST for short).

Given the source code of an Elixir program in a file, a number of steps are executed in order to run the written program. In each of these steps a certain data structure representing the program is generated based on the output of the previous step. These steps are (Noria, 2017):

1. The source code is converted into tokens by the lexer.
2. Tokens generated by the lexer are parsed and an abstract syntax tree is constructed.
3. The AST created in the previous step is translated into Erlang Abstract Format which is Erlang's AST structure.
4. The AST in the Erlang Abstract Format is compiled by Erlang compiler.
5. BEAM virtual machine runs the compiled code.

Next, the structure of Elixir's AST is described. Using Elixir data types described in 2.3, we can write down abstract syntax tree of any Elixir expression. Elixir also provides a built-in function `Code.string_to_quoted!` which can be used to retrieve syntax tree of any Elixir code. Let's take a look at some Elixir expressions and their respective abstract syntax trees.

The following is based on McCord (2015, pp. 8–10). The expressions are represented by a three-element tuple in which the first element contains the operation to be made (for instance, the name of the function or macro to be called), the second one contains metadata such as line number on which the particular expression is in the source code, and the third element is a list that consists of children of the node. The children can be any other expression.

In figure 3, there is an AST of a function call with two arguments. Notice that in the syntax tree the outermost function call is represented by a three-element tuple. The first element in the AST tuple is an atom with a value of `:a` denoting the function's name to be called, the second one is a list of metadata, and the third one is a list of children. As the first child is a function call of the function `b`, it is represented by a three-element tuple in the AST. The second child is a constant `3` which remains as `3` in the abstract syntax tree.

```
> Code.string_to_quoted!("a(b(), 3)")
{:a, [{:line, 1}], [{:b, [{:line, 1}], []}, 3]}
```

Figure 3. AST of a function call.

In figure 4, there is AST of a more complex expression consisting of function calls and arithmetic operators. Observe that in AST the arithmetic operators and function calls are represented in a similar manner.

```
> Code.string_to_quoted!("fun() + 3 * 9 - 4 * 6")
{
  :-,
  [{:line, 1}],
  [
    {
      :+,
      [{:line, 1}],
      [
        {:fun, [{:line, 1}], []},
        {:*, [{:line, 1}], [3, 9]}
      ]
    },
    {:*, [{:line, 1}], [4, 6]}
  ]
}
```

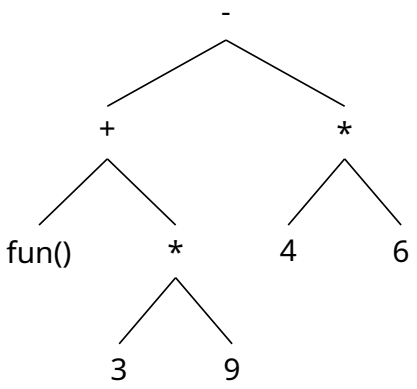

Figure 4. AST of a more complex expression and its visualization.

In figure 5, there is an AST of a function definition. Notice that in the context of AST function definitions are essentially function calls of the function `def` with two arguments:

- The first argument is a function call `fun(param1, param2)` where `fun` is the name of the defined function and `param1` and `param2` are the parameters of the defined function.
- The second argument is a list consisting of one tuple `{:do, body}` where `body` is the body of the defined function and can be any other Elixir expression.

```
> Code.string_to_quoted!("""
  def fun(param1, param2) do
    param1 + param2
  end
""")
{
  :def,
  [{:line, 1}],
  [
    {
      :fun,
      [{:line, 1}],
      [{:param1, [{:line, 1}], nil}, {:param2, [{:line, 1}], nil}]
    },
    [{
      :do,
      {
        :+,
        [{:line, 2}],
        [{:param1, [{:line, 2}], nil}, {:param2, [{:line, 2}], nil}]
      }
    }]
  ]
}
```

Figure 5. AST of a function definition.

Expression `if-else` is also represented as a function call of the function `if` in the syntax tree. Any other Elixir expression can also be expressed similarly using Elixir's own data types.

Elixir provides several built-in functions that can traverse the syntax tree. One of these functions is `Macro.prewalk`, which takes an AST structure and a callback function as a parameter. The provided callback function is called with every node in the syntax tree. The node in the tree is replaced with a value that is returned by the function. In figure 6, there is an example of `Macro.prewalk` which replaces all additions with subtractions. Hence, that function can be useful if one has to make changes to AST while traversing it. In the example, pattern matching was used to match a certain type of nodes.

```
Macro.prewalk(ast, fn
  {:+, meta, children} -> {:-, meta, children}
  other -> other
end)
```

Figure 6. Example of traversing AST and replacing additions with subtractions.

During traversal, it is possible to store state information in an accumulator variable. In figure 7, there is code that counts the number of function definitions in the AST by using an accumulator.

```
Macro.prewalk(ast, 0, fn ast_node, accumulator ->
  case ast_node do
    {:def, _, _} -> {ast_node, accumulator + 1}
    _ -> {ast_node, accumulator}
  end
end)
```

Figure 7. Example of using accumulator in `Macro.prewalk`.

2.5 Credo

Credo is a linter for Elixir that implements rules to discover issues from the source code. Most rules are related to code style. That is, they focus on enforcing a consistent programming style and are not concerned with the semantic meaning of the code. While code style related rules make up most of Credo's rules, there are also some rules that focus on catching functional mistakes. These rules discover common pitfalls in Elixir and code that are forgotten to repositories by mistake, like debugging statements.

To use Credo, it has to be installed as a dependency to an Elixir project. Once installed, the user can optionally configure Credo, more precisely, enable and disable rules. After configuration is complete, Credo can be run using the `mix credo` command. On every run, Credo scans all Elixir files in the current project and checks if any of them contain code that violates some rule. Upon analysing all the files, it prints out all the lines that violate some rule. An example of Credo output is depicted in figure 8.

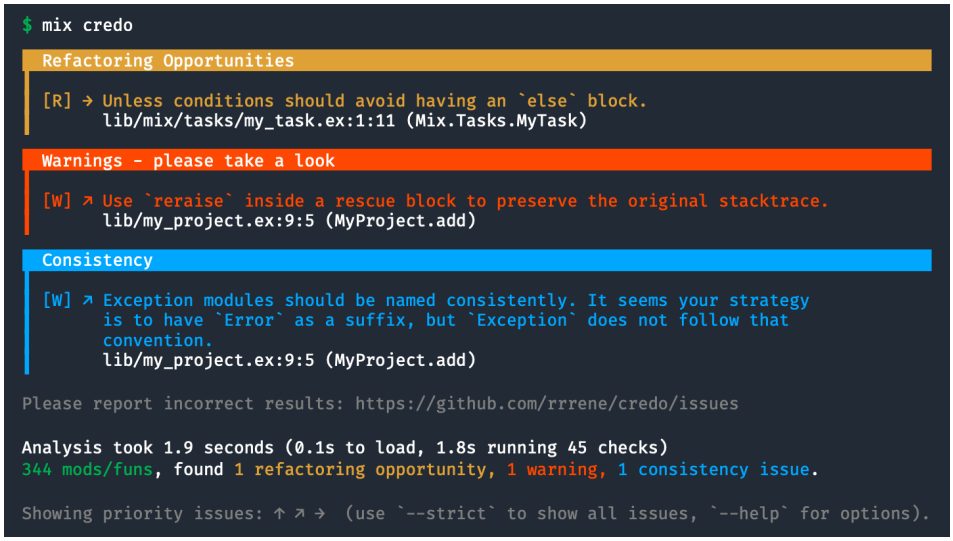

Figure 8. Example Credo output. The image was retrieved from https://github.com/rrrene/credo.

The next paragraphs give examples of different kinds of Credo rules. One example of a rule that enforces code style is `ParenthesesOnZeroArityDefs`. In Elixir, function arguments are usually written inside parentheses, as seen in the first part of figure 9. However, the language allows the programmer to omit these parentheses when the function takes no arguments. The `ParenthesesOnZeroArityDefs` rule enforces that the same convention is used throughout the source code, to wit, that either all zero-argument functions use parentheses in the definition or none of them do. The user can configure which of the two styles should be preferred. The second part of figure 9 shows two functionally equivalent definitions of a zero-argument function.

```elixir
# Defintion of a function that takes two arguments
def average(a, b) do
  (a + b) / 2
end


# Definition of a function that takes no arguments
def epsilon() do
  1e-7
end

# Equivalent definition of the same function
def epsilon do
  1e-7
end
```

Figure 9. Three function definition examples in Elixir.

Another code style related rule is `AliasOrder`. In Elixir, one can create aliases to other modules. In short, to avoid typing a long module name that consists of several components, one can refer to it using only the name of the last component. Figure 10 shows two functionally equivalent implementations of a module. The first example does not use an alias, while the other one does.

```elixir
# Module that uses function from another module
defmodule A do
  def my_func() do
    Something.That.Is.Long.Foo.other_func()
  end
end

# Functionally equivalent module, but now the long module has an alias
defmodule A do
  alias Something.That.Is.Long.Foo

  def my_func() do
    Foo.other_func()
  end
end
```

Figure 10. Example of the use of `alias` keyword in Elixir.

In larger modules, there is often a high number of aliases. To make these aliases more structured and readable, Credo's `AliasOrder` requires aliases to be sorted in alphabetical order. For example, consider two modules `A` and `B` in figure 11. The module `A` does not conform to the order required by `AliasOrder`, while module `B` does.

```elixir
defmodule A do
  alias OtherModule.Apples
  alias OtherModule.Cherries
  alias OtherModule.Bananas
end
```

```elixir
defmodule B do
  alias OtherModule.Apples
  alias OtherModule.Bananas
  alias OtherModule.Cherries
end
```

Figure 11. Two modules that use several aliases. The module on the left does not conform to the order required by `AliasOrder` rule, while the one on the right does.

An example of a rule that looks at program behaviour is `RaiseInsideRescue`. That rule protects against a common pitfall in Elixir related to catching and later rethrowing the same error. In particular, Elixir has a special keyword `reraise` that is meant to be used for throwing a previously caught error. However, a programmer that is not aware of that keyword may use the standard `raise` keyword. The difference between these two keywords is that the first one preserves the stack trace from the original error while the other one does not. This makes it more difficult to understand where an error happened if the standard `raise` keyword is used. Figure 12 shows two pieces of code, an incorrect one on the left that violates the `RaiseInsideRescue` rule, and a correct one which the rule accepts.

```elixir
try do
  # do something
rescue
  error ->
    # do something with the error
    raise error
```

```elixir
try do
  # do something
rescue
  error ->
    # do something with the error
    reraise error, System.stacktrace
```

Figure 12. Example of code that improperly raises an error on the left and preferred way to do it on the right.

Internally, Credo uses the abstract syntax tree of the code to find code that does not comply with rules. Although every rule is implemented separately, all of them share the same working principle — they simply walk the AST and if they observe a pattern that corresponds to an offense, the offense is reported.

3 Development of Autocorrect

In this section, requirements that were set for the autocorrect addon are presented. Also, an overview is given on how the autocorrect functionality was technically implemented.

3.1 Requirements

At the beginning of development, the requirements that the autocorrect addon must meet were set as follows:

- Adding autocorrect addon to an existing Elixir project must be effortless and doable by changing at most a few lines of configuration. The reasoning behind this requirement is that if the addon is challenging to set up, then people would not use it.
- It must be possible to run autocorrect on the whole project with a single CLI command. The goal of the developed tool is to increase developer productivity. Hence running it must be quick and simple.
- Autocorrect must not alter the behaviour of the code. If autocorrect were to change the semantics of the code, then this could lead to bugs in the software.

3.2 Architecture

In terms of software architecture, the autocorrect addon consists of the following parts:

- **Correction modules for each of the rules.** For each rule which is feasible to fix automatically, an Elixir module was developed which can fix violations of that one particular rule. These modules take an abstract syntax tree as an input and output modified syntax tree.
- **Converting modified AST back to source code.** Once each of the correction modules have been run, the new modified syntax trees have to be converted back into the source code.
- **Credo Plugin.** This is needed for providing CLI to the user. The plugin is also responsible for wiring together the other two parts in the architecture.

Each of these parts is described thoroughly in the following sections.

3.3 Correction modules

For each of the Credo checks an Elixir module was developed, which is responsible for fixing violations of that particular rule. This module is called a correction module. How the correction module works is dependent on the specific rule, but broadly speaking, it works by first traversing AST of the source file to detect tree nodes where the rule is violated. One may think that Credo API could be used for finding violations, but the Credo API allows us only to retrieve the files and line numbers where the issues lie. In order to make changes to the syntax tree, it must be known in which AST node the violation is. Still, it was possible to use some of the code from Credo to detect violations. Next, once the violations have been found, changes are made to the AST in such way that there would be no violations of the rule. The rules for which correction module was developed are:

- `AliasOrder`
- `CondStatements`
- `NegatedConditionsInUnless`
- `ParenthesesOnZeroArityDefs`
- `SinglePipe`
- `UnlessWithElse`
- `WithSingleClause`

The following paragraphs provide examples of how the process exactly works for some of the rules. One rule for which a correction module was created is `ParenthesesOnZeroArityDefs`, described in the section 2.5. In figure 13 there are two code examples and their respective syntax trees. Both examples show a function definition, but the one on the left has no parentheses, while the one on the right has parentheses. Regarding AST the difference between the two trees is the node representing function name `Y`. If parentheses are present, then the children of that node are an empty list, but if the parentheses are not present, the children are `nil`.

```
# def Y, do: X
{
  :def,
  [{:line, 1}],
  [
    {:Y, [{:line, 1}], nil},
    [{:do, X}]
  ]
}
```

```
# def Y(), do: X
{
  :def,
  [{:line, 1}],
  [
    {:Y, [{:line, 1}], []},
    [{:do, X}]
  ]
}
```

Figure 13. On the left, there is an AST generated when parentheses are absent. On the right, there is AST when the parentheses are present.

In figure 14, there is a code snippet taken from the correction module of `ParenthesesOnZeroArityDefs`. In this code the AST of a file is traversed, and during the traversal changes are made to the syntax tree to remove the violations. The following logic is applied to every AST node representing a function definition:

- If parentheses are required per Credo configuration, and they are not present, then they are added.
- If parentheses must not be present per Credo configuration, but they are present, then parentheses are removed.
- In other cases, the node does not change.

```
Macro.prewalk(ast, fn
  {def_op, def_meta, [{fun_name, fun_meta, nil} | other]}
  when def_op in @def_ops and is_atom(fun_name) and parens? ->
    {def_op, def_meta, [{fun_name, fun_meta, []} | other]}

  {def_op, def_meta, [{fun_name, fun_meta, []} | other]}
  when def_op in @def_ops and is_atom(fun_name) and not parens? ->
    {def_op, def_meta, [{fun_name, fun_meta, nil} | other]}

  other ->
    other
end)
```

Figure 14. Code snippet from a correction module for `ParenthesesOnZeroArityDefs`.

Autocorrect was also implemented for the `AliasOrder` rule, described in the section 2.5. Unlike the previous rule, this one is more complex to fix and is done in several parts. At a high level, the following steps are executed to fix the violations of this rule:

1. Some metadata is added to every node under a module definition. In particular, the metadata is changed to contain information regarding the ordering of nodes under the module. This information is used in the next step.

2. AST is traversed again to find where it is even possible to reorder aliases. If there are aliases which use other aliases, the reordering may not be possible. In such cases, the particular aliases are marked not to be ordered.

3. Finally, in modules in which it is possible to reorder nodes, a sort function is applied to the nodes so that calls to `alias` are ordered lexicographically and other nodes keep their initial order.

3.4 Converting modified AST back to code

Once the corrections have been made to the AST, the last step of autocorrect is to convert the syntax tree back to code and write it to the corresponding source file. Although Elixir has a built-in function called `Macro.to_string`, which takes an AST as an argument and returns the source code corresponding to the AST, there is one significant problem with this function due to which it cannot be used in the autocorrect addon. Namely, this function does not preserve any formatting information about the initial source code, such as the number of spaces used for indentation, code comments and the number of line breaks between expressions. This is because when parsing the source code, some information related to formatting is discarded.

To preserve formatting information when converting the AST back to the source code, an Elixir library called Sourceror[3] was used in the autocorrect addon. Sourceror provides a function `Sourceror.parse_string` which creates an abstract syntax tree from the source file, but with the exception that the formatting information related to each node in AST is stored under the node metadata. In figure 15, on the left there is an AST created by Elixir's built-in `Code.string_to_quoted` while on the right there is the AST for the same source code, but created by Sourceror. Notice that the tree created by Sourceror contains extra formatting information.

[3] https://github.com/doorgan/sourceror

```
> Code.string_to_quoted!("""
    # Add two variables
    a + b
""")
{
  :+,
  [{:line, 2}],
  [
    {:a, [{:line, 2}] nil},
    {:b, [{:line, 2}], nil}
  ]
}
```

```
> Sourceror.parse_string!("""
    # Add two variables
    a + b
""")
{
 :+,
 [{:trailing_comments, []}, {:line, 2}, {:column, 7}],
 [
   {:a, [
    {:trailing_comments, []},
     {:leading_comments, [
       %{
        column: 5,
        line: 1,
        next_eol_count: 1,
        previous_eol_count: 1,
        text: "# Add two variables"
       }
     ]},
     {:line, 2},
     {:column, 5}
    ],
    nil
   },
   {:b, [
     {:trailing_comments, []},
     {:leading_comments, []},
     {:line, 2},
     {:column, 9}
    ], nil}
 ]
}
```

Figure 15. On the left is the AST generated by Elixir and on the right is the AST generated by Sourceror library.

When the correction modules described in section 3.3 are run on the AST created by Sourceror, the modified tree returned by the modules still has all the formatting information about the original source code available. Sourceror is then able to convert this modified tree back to the source code while preserving comments and formatting of the original code.

3.5 Developing a Credo Plugin

Credo has a plugin API that provides an interface to add new functionality to Credo without making changes to Credo's own source code. One feature that the plugin API provides is to allow plugins to define new commands for Credo, those commands can

then be invoked from the command line interface. When a user runs the command defined by a plugin, Credo calls plugin's callback function which handles the command. To be able to use commands defined by Credo plugins, the plugin has to be added to Credo's configuration file.

Autocorrect addon was implemented as a Credo plugin. The created plugin defines a new command `fix` which can be invoked from the command line with `mix credo fix`. When invoked, the command does the following:

1. It loads and validates all the source files in the project to be used for later analysis. Because Credo itself already does it, it was possible to reuse the same module.

2. It determines which checks are enabled, as users can disable checks in the configuration. This is needed as we need to know for which checks we need to run autocorrection. Here it was also possible to use the existing module from Credo.

3. For each of the enabled checks for which the autocorrection module has been implemented, the respective correction module is invoked.

4. Once all correction modules have been called, the changed syntax trees are converted back to Elixir code and written to the respective files.

4 Results

In this section, the developed autocorrect addon is applied to several Elixir projects that do not use Credo. The goal of this is to see if the autocorrect addon can fix the violations of the rules. Additionally, the unit tests are executed to ensure that the semantics of the code has remained the same after running autocorrect. The projects on which autocorrect was tested are:

- Broadway[4]
- Tesla HTTP Client[5]

These two projects were chosen from the popular Elixir projects list in GitHub. After Credo and autocorrect plugin were added to these projects, Credo itself and then the `mix credo fix` command was run. In Broadway, Credo reported 20 issues, of which 5 autocorrect addon were able to fix. In Tesla HTTP Client, Credo reported 50 issues out, of which 6 autocorrect addon were able to fix. After running autocorrect, unit tests were run in both projects with `mix test` to ensure that autocorrect had not caused any side effects. All of the tests passed.

4.1 Conclusion

The goal of this thesis was to build an addon for Credo linter that is able to fix some of the issues that Credo reports. Elixir, Credo and Elixir's abstract syntax tree and tools for manipulating it were studied in the first part of the thesis and based on this information the addon was built. The addon was applied to multiple Elixir projects where it was found that autocorrect is indeed able to fix violations of the rules for which it was developed. Additionally, it was ensured that autocorrect does not cause any side effects to the code it changes. Thus, it can be concluded that the goal of the thesis was achieved.

The code, along with the tutorial on how it can be added to an Elixir project can be found at https://github.com/bytebuf/credo_fixer.

4.2 Future Work

Autocorrect addon could be improved in the following ways:

- Autocorrect was developed only for a subset of rules. Support for additional rules could be added.
- Autocorrect could be integrated with an IDE such as IntelliJ IDEA so that the developer can run it directly from the IDE.

[4]https://github.com/dashbitco/broadway
[5]https://github.com/teamon/tesla

References

Elixir Use Cases. (n.d.). Retrieved April 26, 2022, from https://elixir-lang.org/cases.html

Johnson, B., Song, Y., Murphy-Hill, E., & Bowdidge, R. (2013). Why Don't Software Developers Use Static Analysis Tools to Find Bugs? *Proceedings of the 2013 International Conference on Software Engineering*, 672–681.

Juric, S. (2019). *Elixir in Action*. Manning. https://books.google.ee/books?id=Dt6xtAEACAAJ

McCord, C. (2015). *Metaprogramming Elixir*. The Pragmatic Programmers, LLC. https://books.google.ee/books/about/Metaprogramming_Elixir.html?id=IA9QDwAAQBAJ

Noria, X. (2017). How does Elixir compile/execute code? Retrieved April 28, 2022, from https://medium.com/@fxn/how-does-elixir-compile-execute-code-c1b36c9ec8cf

Rafnsson, W., Giustolisi, R., Kragerup, M., & Høyrup, M. (2020). Fixing Vulnerabilities Automatically with Linters. *14th International Conference on Network and System Security*. https://doi.org/10.1007/978-3-030-65745-1

Stack Overflow Developer Survey insights. (2021). Retrieved April 26, 2022, from https://insights.stackoverflow.com/survey/2021#technology-most-popular-technologies

Tómasdóttir, K. F., Aniche, M., & Van Deursen, A. (2020). The Adoption of JavaScript Linters in Practice: A Case Study on ESLint. *IEEE Transactions on Software Engineering*, *46*(8), 863–891. https://doi.org/10.1109/TSE.2018.2871058

Valim, J. (2020). Real time communication at scale with Elixir at Discord. Retrieved April 26, 2022, from https://elixir-lang.org/blog/2020/10/08/real-time-communication-at-scale-with-elixir-at-discord/

Appendix

I. Licence

Non-exclusive licence to reproduce thesis and make thesis public

I, **Aron Sisask**,
*(*author's name)

1. herewith grant the University of Tartu a free permit (non-exclusive licence) to

   reproduce, for the purpose of preservation, including for adding to the DSpace digital archives until the

2. expiry of the term of copyright,

   Automatic Error Correction for Elixir,
   *(*title of thesis)

   supervised by Vesal Vojdani.
   *(*supervisor's name)

3. I grant the University of Tartu a permit to make the work specified in p. 1 available to the public via the web

4. environment of the University of Tartu, including via the DSpace digital archives, under the Creative

5. Commons licence CC BY NC ND 3.0, which allows, by giving appropriate credit to the author, to reproduce,

6. distribute the work and communicate it to the public and prohibits the creation of derivative works and

7. any commercial use of the work until the expiry of the term of copyright.

8. I am aware of the fact that the author retains the rights specified in p. 1 and 2.

9. I certify that granting the non-exclusive licence does not infringe other persons' intellectual property rights

10. or rights arising from the personal data protection legislation.

Aron Sisask
10/05/2022