Imperial College London

MEng Individual Project:
Interim Report

Imperial College London

Department of Computing

`parsley-garnish`: A Static Analysis Tool for the `parsley` Parser Combinator Library

Supervisor:
Dr. Jamie Willis

Author:
Rocco Jiang

Second Marker:
Dr. Robert Chatley

January 24, 2024

Contents

Chapter 1

Introduction

1.1 Motivation

Parser combinators [1] are an elegant approach for writing parsers in a manner that remains close to the original grammar specification. `parsley` [2] is a parser combinator library implemented as an embedded domain-specific language (DSL) [3] in Scala, with an API inspired by the `parsec` [4] family of libraries in Haskell. However, as with many libraries, there exists a learning curve to utilising `parsley` and parser combinator libraries in an idiomatic manner.

While well-documented, the wealth of information to get started with `parsley` can be overwhelming for users, particularly those new to parser combinators. Furthermore, there exists a number of design patterns [5] for writing maintainable parsers, which even experienced users may be unaware of. A potential solution to this problem is tooling to provide automated code hints, which a user can use during the development cycle to evaluate if their code adheres to best practices.

A number of modern integrated development environments (IDEs) provide code hints to warn programmers about problems in their source code, highlighting offending snippets and suggesting actions to improve suboptimal or incorrect code [6]. Many of these code analysis tools are designed to detect general issues for the host language, rather than specifically for libraries. However, tools may also utilise domain-specific code analyses in order to detect issues specific to a particular system or problem domain [7, 8, 9].

This project aims to explore the potential of harnessing static code analysis techniques to develop a new tool, `parsley-garnish`, that offers code hints aimed at assisting programmers in writing idiomatic and correct `parsley` code. Additionally, for certain issues that can be automatically fixed, `parsley-garnish` will provide automated actions to resolve the issue. The goal of `parsley-garnish` is to be used as a companion library to `parsley`, in order to improve its ease of adoption and to help users enforce best practices.

1.2 Planned Objectives

Chapter 2

Static Analysis Tools

cite

Static program analysis is the process of automatically analysing source code to extract information about its behavior without executing it, as opposed to dynamic analysis, which is performed on programs as they are run. Static analysis tools can ease the burden of software development by automating tasks that would otherwise require manual effort and meticulous attention to detail. These tools can perform a variety of tasks, ranging from detecting possible bugs [10, 11] to formal software verification of program properties [12].

Static analysis tools are increasingly becoming more important in modern software development, as modern code continues to become more complex and difficult to reason about. Industry leaders, such as Google [13] and Meta (formerly Facebook) [14], have embraced static analysis tools as integral components of their software development workflows.

2.1 Applications of Static Analysis Tools

Typically in a software development workflow, multiple static analysis tools are used in conjunction to provide a comprehensive suite of checks. Often, these tools are integrated into an IDE as plugins, allowing them to provide real-time feedback to aid developers in writing idiomatic code.

Refactoring and linting are two major functionalities that these tools provide. However, because many of these tools include both capabilities, the distinction between them can be blurry. This section aims to establish definitions for the two terms, which will be used as a basis for subsequent discussion within this thesis.

2.1.1 Automated Refactoring

Code refactoring is a well-established practice in software development. In his influential book *Refactoring: Improving the Design of Existing Code* [15], Fowler defines **refactoring** as "the process of changing a software system in such a way that it does not alter the external behavior of the code yet improves its internal structure". Refactoring may be employed to eliminate **code smells**, which are surface indications that could indicate deeper problems in the system. Code smells are not necessarily problematic on their own, however, they may lead to issues such as bugs or poor maintainability if left unchecked. Examples of code smells include duplicated code, which can be hard to update without introducing bugs, and long methods, which can be difficult to understand and maintain. Therefore, it is often productive to refactor code to eliminate code smells, even if the code is still correct and functional.

Static analysis tools can reason about how to safely refactor code in an automated manner, performing refactorings as source-to-source transformations. These transformations may be implemented as simple text-based replacements or more robust rewrite rules that operate on the abstract syntax tree (AST) of the source code.

Automated refactoring support is particularly useful for large codebases, where manual refactoring would be tedious and error-prone. Generally, when a user makes use of an automated refactoring tool in an IDE, they will manually identify the snippet of code that they wish to refactor, and then select the appropriate refactoring from a list of available options. Fig. 2.1 presents *Extract Method* [15], an example of a common refactoring that can be performed automatically on a block code selected by the user.

2.1.2 Linting

Linting is the process of analysing source code to identify and report issues related to coding style and potential logical errors. The term originates from the `lint` program [10], which examined C source code for bugs, as well as wasteful code patterns that may be legal but error-prone. The tool was also utilised to enforce portability restrictions which aided users in writing portable code that could be compiled on multiple platforms. Since the release of `lint`, many linting tools, known as **linters**, have been developed for a wide range of programming languages.

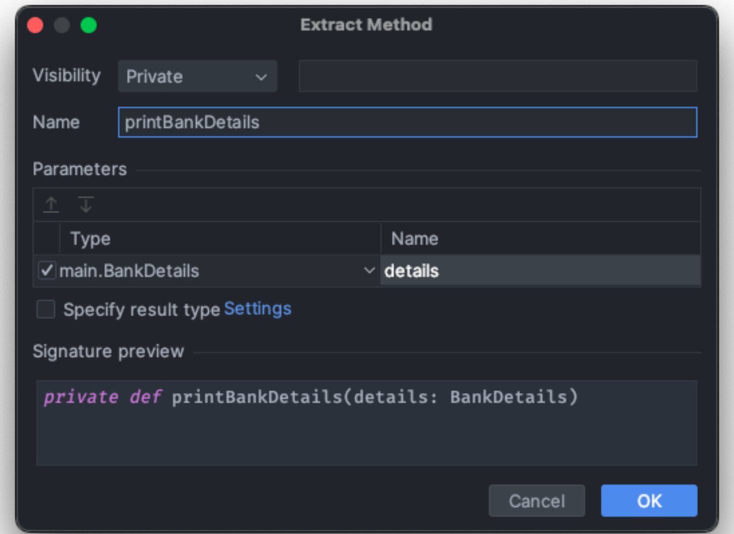
Linters are provided as standalone tools separate from a compiler, since their primary goal is to suggest improvements for code readability and maintainability, rather than code optimisations. Modern linters are commonly integrated into IDEs, where code analysis performed by the linter is run incrementally in the background. Any violations found by the linter are displayed directly in the editor as warnings or errors at the relevant locations in the source code. This provides an ergonomic user experience, as the user can see the results of the analysis in real-time as part of the development workflow.

Furthermore, advanced linters can provide automated code transformations to fix violations which can be corrected automatically by static analysis. When a linter integrated into an IDE reports a warning, it may present the user with a *code action* or *quick fix* to apply the fix automatically.

Many linters are configurable with a set of rules, which specify the types of issues that the linter should detect. These rules can be enabled or disabled by the user, allowing them to customise the linter to their needs. Rules can be categorised by their purpose: some rules are concerned with enforcing code style, while others are concerned with detecting code smells or other suspicious code patterns indicative of possible bugs.

```scala
object Main {
  def main(args: Array[String]): Unit = {
    val bankDetails = getBankDetails()
    println(s"Account name: ${bankDetails.name}")
    println(s"Account balance: ${bankDetails.balance}")
  }
}
```

(a) A snippet of Scala code. A user may wish to extract the highlighted lines into a separate function.



(b) When a user selects the highlighted lines from fig. 2.1a in IntelliJ IDEA, choosing the *Extract Method* refactoring will open this dialogue to preview changes before applying them.

```scala
object Main {
  def main(args: Array[String]): Unit = {
    val bankDetails = getBankDetails()
    printBankDetails(bankDetails)
  }

  private def printBankDetails(details: BankDetails): Unit = {
    println(s"Account name: ${details.name}")
    println(s"Account balance: ${details.balance}")
  }
}
```

(c) The result of applying the *Extract Method* refactoring using the chosen parameters in fig. 2.1b.

Figure 2.1: An example of the *Extract Method* refactoring in IntelliJ IDEA [16].

Style checking and code appearance

Linters can be configured to enforce a particular style guide defining a set of conventions for how idiomatic code should be written. They can highlight style violations and in basic cases, automatically rewrite code to conform to the correct style. For example, the *Flake8* linter for Python enforces coding conventions from the PEP 8 style guide . Stylistic rules are especially helpful for large projects with multiple contributors, where a consistent coding style can improve readability and maintainability.

cite

cite

Identifying opportunities for refactoring

Certain linting rules can aid in the refactoring process by broadly identifying code smells and candidate areas for refactoring, suggesting appropriate actions that the user can take. As an example, a linter may detect a fragment of code that is repeated in multiple places: this is a code smell, as discussed previously. The linter may then suggest a code action to automatically apply the *Extract Method* refactoring to avoid code duplication.

Suggesting idiomatic usage

Other rules can suggest opportunities to improve more precise snippets of code by utilising language features in a more idiomatic manner. These rules are especially helpful for new users of a language, who may be unaware of useful language constructs and idioms. For example, the *Clippy* linter for Rust categorises a collection of rules as `clippy::complexity` rules to detect code that does something simple in a complex way and suggests a simpler alternative. Fig. 2.2 provides an example of a similar rule in Haskell, from the *HLint* linter [17]. The rule suggests an η-reduction refactoring, presented to the user as a code action that can be applied automatically.

cite article

```haskell
foo xs = map (+1) xs
```

(a) A Haskell function `foo`, which can be made more concise using η-reduction.

```
Eta reduce
Found:
  foo xs = map (+ 1) xs
Why not:
  foo = map (+ 1)
hlint(refact:Eta reduce)
```

(b) The linter warning shown for `foo`.

Figure 2.2: An example of a warning from the Haskell linter `hlint`, suggesting a fix that a user can choose to automatically apply.

Many idiomatic practices exist to avoid common pitfalls that may lead to unintended behaviour. By highlighting good practices, linters can help users avoid these common mistakes that may cause bugs. For example, *ESLint* , one of the most popular JavaScript linters, warns

cite

against common JavaScript pitfalls such as using the regular equality operator == instead of its type-safe alternative ===.

Linters developed for a specific library provide rules to enforce idiomatic usage specific to the domain of the library. A library or especially an embedded DSL may require a particular style of usage that is different from the host language. Therefore, a regular linter for the host language may not be able to detect issues specific to the library. In this case, an accompanying linter can greatly benefit users: common misuses can be detected and sometimes automatically fixed, and users can be directed to relevant documentation to learn more about correct usage. For instance, the *xUnit.net* testing framework for C# is accompanied by an analyser package [9] which provides linting rules to enforce best practices specific to xUnit.

Detecting potential bugs

Linters may also directly attempt to detect more serious issues in code, such as possible logic errors. This can be helpful for even experienced users to avoid common pitfalls. Clippy has `clippy::suspicious` and `clippy::correctness` rule categories to identify code that is very likely to be incorrect or useless. ESLint provides several rules to warn against code patterns that are likely to cause runtime errors, such as re-assigning a `const` variable.

2.2 Implementing Static Analysis Tools

This section first discusses the choices available for implementing a static analysis tool in Scala. We then discuss how the chosen implementation tool provides a framework for implementing linting and refactoring rules.

Chapter 3

Parser Combinators

Chapter 4

Project Plan

Chapter 5

Evaluation

Bibliography

[1] Hutton G. Higher-order functions for parsing. Journal of Functional Programming. 1992 Jul;2(3):323-43. Available from: https://doi.org/10.1017/S0956796800000411.

[2] Willis J, Wu N. Garnishing parsec with parsley. In: Proceedings of the 9th ACM SIGPLAN International Symposium on Scala. Scala 2018. New York, NY, USA: Association for Computing Machinery; 2018. p. 24-34. Available from: https://doi.org/10.1145/3241653.3241656.

[3] Hudak P. Building domain-specific embedded languages. ACM Comput Surv. 1996 dec;28(4es):196-es. Available from: https://doi.org/10.1145/242224.242477.

[4] Leijen D, Meijer E. Parsec: Direct Style Monadic Parser Combinators for the Real World; 2001. UU-CS-2001-27. User Modeling 2007, 11th International Conference, UM 2007, Corfu, Greece, June 25-29, 2007. Available from: https://www.microsoft.com/en-us/research/publication/parsec-direct-style-monadic-parser-combinators-for-the-real-world/.

[5] Willis J, Wu N. Design patterns for parser combinators in scala. In: Proceedings of the Scala Symposium. Scala '22. New York, NY, USA: Association for Computing Machinery; 2022. p. 9-21. Available from: https://doi.org/10.1145/3550198.3550427.

[6] Kurbatova Z, Golubev Y, Kovalenko V, Bryksin T. The IntelliJ Platform: A Framework for Building Plugins and Mining Software Data. In: 2021 36th IEEE/ACM International Conference on Automated Software Engineering Workshops (ASEW); 2021. p. 14-7. Available from: https://doi.org/10.1109/asew52652.2021.00016.

[7] Renggli L, Ducasse S, Gîrba T, Nierstrasz O. Domain-Specific Program Checking. In: Vitek J, editor. TOOLS'10: Proceedings of the 48th International Conference on Objects, Models, Components, Patterns. vol. 6141 of Lecture Notes in Computer Science. Berlin, Heidelberg: Springer; 2010. p. 213-32. Available from: https://doi.org/10.1007/978-3-642-13953-6_12.

[8] Gregor D, Schupp S. STLlint: lifting static checking from languages to libraries. Software: Practice and Experience. 2006;36(3):225-54. Available from: https://doi.org/10.1002/spe.683.

[9] xUnit net. xUnit.net Analyzers package;. [Accessed: 2024 Jan 22]. Available from: https://github.com/xunit/xunit.analyzers.

[10] Johnson SC. Lint, a C program checker. AT&T Bell Laboratories; 1978. Technical Report 65.

[11] Hovemeyer D, Pugh W. Finding bugs is easy. SIGPLAN Not. 2004 dec;39(12):92-106. Available from: https://doi.org/10.1145/1052883.1052895.

[12] Blanchet B, Cousot P, Cousot R, Feret J, Mauborgne L, Miné A, et al. A static analyzer for large safety-critical software. In: Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation. PLDI '03. New York, NY, USA: Association for Computing Machinery; 2003. p. 196-207. Available from: https://doi.org/10.1145/781131.781153.

[13] Sadowski C, Aftandilian E, Eagle A, Miller-Cushon L, Jaspan C. Lessons from building static analysis tools at Google. Commun ACM. 2018 mar;61(4):58-66. Available from: https://doi.org/10.1145/3188720.

[14] Calcagno C, Distefano D, Dubreil J, Gabi D, Hooimeijer P, Luca M, et al. Moving Fast with Software Verification. In: Havelund K, Holzmann G, Joshi R, editors. NASA Formal Methods. Cham: Springer International Publishing; 2015. p. 3-11.

[15] Fowler M. In: Refactoring: Improving the Design of Existing Code. Addison-Wesley; 2018. .

[16] JetBrains. IntelliJ IDEA; 2024. [Accessed: 2024 Jan 24]. Available from: https://www.jetbrains.com/idea/.

[17] Mitchell N. HLint: Source code suggestions;. [Accessed: 2024 Jan 23]. Available from: https://hackage.haskell.org/package/hlint.