Imperial College London

MEng Individual Project:
Interim Report

Imperial College London

Department of Computing

`parsley-garnish`: Static Analysis for Parser Combinator Libraries

Author:
Rocco Jiang

Supervisor:
Dr. Jamie Willis

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

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 sound verification of program properties [12].

cite

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 Purposes 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. These tools enable the developer to perform a variety of tasks and activities to aid them in writing idiomatic code.

A point of confusion is that many modern static analysis tools may perform multiple of these activities, and as such, terms referring to such activities are often used interchangeably. This section aims to establish definitions for some of these tasks, which will be used as a basis for subsequent discussion within this thesis. However, it is important to remember that the boundaries between them can be blurry.

2.1.1 Linting

Linting is the process of analysing source code to identify and report issues related to coding style, formatting, and potential programming errors. The term originates from the `lint` program [10], which examined C source code for bugs, as well as wasteful code patterns that are 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 rather than as part of a compiler, since their primary goal is to suggest improvements for code readability and maintainability, rather than code optimisations. The majority of modern-day linters are integrated into code editors and IDEs as plugins, allowing them to provide real-time feedback to developers as they write code. Code analysis performed by the linter is run incrementally in the background and then displayed directly in the editor. This enables developers to make informed decisions while writing code, rather than having to run the linter as part of compilation or as an extra step in their workflow.

Within the editor, linters can highlight problematic snippets of code and in some cases can offer relevant recommendations to fix them. This is achieved by detecting suspicious code patterns that may be indicative of possible bugs or poor coding practices: these are commonly referred to as **code smells** [15]. Furthermore, linters are also used as style checkers to enforce coding standards and best practices , ensuring that code is idiomatic, readable, and consistent across a codebase. Some linters may also provide automated code transformations to fix formatting issues or style violations that they detect. An example of a linter providing a warning and automated code fix to help improve code style is shown in fig. 2.1.

cite CheckStyle, ESLint?

```
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 2.1: An example of a warning from the Haskell linter `hlint` [16], suggesting a fix that a user can choose to automatically apply.

2.1.2 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". In essence, refactoring is a particular type of code restructuring that improves its design and eliminates code smells while preserving behaviour.

Static analysis tools can reason about how to safely refactor code in an automated manner, performing refactorings as source-to-source transformations. This is particularly useful for large codebases, where manual refactoring would be tedious and error-prone. Fig. 2.2 presents *Extract*

Function, an example of a common refactoring that can be performed automatically, where a block of code is extracted into its own function.

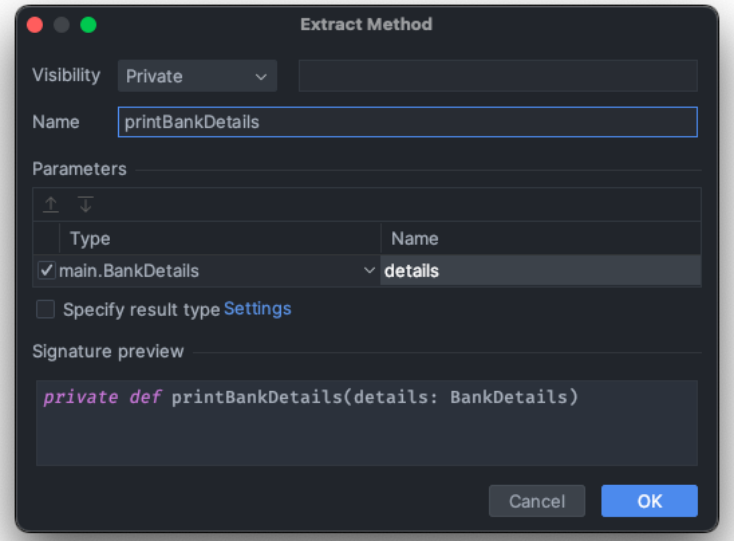
Generally, when a user makes use of an automated refactoring tool in an IDE, they will manually identify the snippet of code that they wish to refactor, and then select the appropriate refactoring from a list of available options. Linters can also aid in the refactoring process by identifying candidate areas for refactoring and suggesting appropriate actions that the user can take.

Given that many linters can perform automated code transformations, the distinction between refactoring and linting can be confusing. We make the key distinction that linting is concerned with the *detection* of issues in code, while refactoring is concerned with *transforming* code to avoid some of these issues. When linters apply automated code transformations to fix issues, some of these transformations may be considered refactorings. Thus, many static analysis tools provide both linting and refactoring functionalities. For example, when a linter detects a fragment of code that is repeated in multiple places, it may suggest the *Extract Function* refactoring to avoid code duplication. The linting tool itself may provide the automated refactoring functionality to perform the transformation. The η-reduction example in fig. 2.1 is another example of a refactoring suggested by a linter to conform to an idiomatic Haskell style. We see that `hlint` can both highlight the issue to the user and perform the suggested refactoring automatically.

2.1.3 Bug Fixing / Automated Program Repair

```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) A refactoring dialogue for the *Extract Function* refactoring in IntelliJ IDEA, which can be opened within the available refactorings by selecting the highlighted lines.

```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 Function* refactoring using the chosen parameters in fig. 2.2b.

Figure 2.2: An example of the *Extract Function* refactoring in IntelliJ IDEA [17].

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

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