Imperial College London

parsley-garnish

A linter for the `parsley` parser combinator library

Rocco Jiang

June 4, 2024

A dissertation submitted in fulfilment of the requirements for

the degree of Master of Engineering

in the

Department of Computing

Imperial College London

Statement of Originality

Write about how this is your own work etc etc (check what's needed)

Rocco Jiang

June 4, 2024

Contents

1 Introduction **1**

1.1 Motivation . . . 1

1.2 Project Goals . . . 1

2 Background **2**

2.1 Linters . . . 2

2.1.1 Categories of Lint Rules . . . 2

2.1.2 Implementing Linters . . . 4

2.2 Writing Linters for Scala . . . 6

2.2.1 Choice of Tooling . . . 6

2.2.2 A Crash Course on Scalafix . . . 8

2.3 Parser Combinators . . . 10

2.3.1 Parsley . . . 12

2.3.2 Design Patterns for Parsley . . . 13

3 Design and Implementation **14**

3.1 Parser Representation . . . 14

3.1.1 Detecting Top-Level Parsers . . . 14

3.1.2 Converting Scalameta Terms to the Parser ADT . . . 15

3.1.3 Building New Parsers From Existing Parsers . . . 16

3.1.4 Simplifying Parsers Using Parser Laws . . . 16

3.1.5 Converting Parsers Back to Scalameta Terms . . . 19

3.2 Function Representation . . . 19

3.2.1 Motivation . . . 19

3.2.2 Approach . . . 19

3.2.3 Converting Scalameta Terms to the Function ADT . . . 21

3.2.4 Normalising Function Terms . . . 21

3.2.5 Converting Functions back to Scalameta Terms . . . 23

3.2.6 Further Work? . . . 23

3.3 Limitations . . . 24

4 Lint Rules in `parsley-garnish` **25**

4.1 Avoid Redefining Existing Parsers . . . 25

4.2 Simplify Complex Parsers . . . 25

4.3 Ambiguous Implicit Conversions . . . 25

4.4 Remove Explicit Usage of Implicit Conversions . . . 28

4.5 Refactor to use Parser Bridges . . . 28

4.6 Left Recursion Factoring . . . 28

5 Evaluation **29**

6 Related Work **30**

7 Conclusion **31**

7.1 Review 31

7.2 Future Work 31

Bibliography **32**

A Proofs of Parser Laws **34**

A.1 Left absorption for `fmap` 34

List of Figures

2.1 An example of a warning from the Haskell linter `hlint`, suggesting a fix that a user can choose to automatically apply. . . . 3
2.2 An example of the *Extract Method* refactoring in IntelliJ IDEA. . . . 5
2.3 A syntactic rule that makes all `val` definitions lazy. . . . 9
2.4 Using Scalameta quasiquotes to construct a tree. . . . 10
2.5 An example showing how semantic rules are semantic-aware on overloaded method names. . . . 11
2.6 The grammar and AST for our simple expression language. . . . 13
2.7 A parser for our simple expression language. . . . 13

3.1 Functor (3.1), Applicative (3.2, 3.3), and Alternative (3.4–3.8) laws. . . . 17
3.2 Normalisation by evaluation for the `Function` datatype. . . . 22

4.1 A minimal Parsley program which fails to compile due to ambiguous implicits in the `parser` object. 27
4.2 The warning message produced by the `AmbiguousImplicitConversions` lint rule. . . . 27

Chapter 1
Introduction

1.1 Motivation

Parser combinators [Hutton 1992] are an elegant approach for writing parsers in a manner that remains close to the original grammar specification. `parsley` [Willis and Wu 2018] is a parser combinator library implemented as an embedded domain-specific language (DSL) [Hudak 1996] in Scala, with an API inspired by the `parsec` [Leijen and Meijer 2001] 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 [Willis and Wu 2022] 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 [Kurbatova et al. 2021]. 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 [Renggli et al. 2010; Gregor and Schupp 2006].

1.2 Project Goals

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.

Chapter 2
Background

2.1 Linters

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 [Johnson 1978], 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.

Nowadays, many linters can be 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 brings early, real-time feedback to the programmer, allowing them to address issues as they write code, with minimal interruption to their development workflow. Linters can also be integrated as part of the code review process, or into continuous integration (CI) pipelines to ensure that code adheres to a set of standards before being merged into the main codebase.

Although the traditional definition for linting is concerned only with *detecting* issues in code, modern linters have broadened their scope significantly. In addition to detecting issues, many linters provide *auto-fix* capabilities to correct issues by automatically rewriting the offending code snippets. This feature is often integrated into IDEs as well: the popular Language Server Protocol for defining IDE features enables these auto-fix features via *code actions* [Gunasinghe and Marcus 2022]. When a section of code is highlighted by a linter warning, a user can apply a code action to automatically fix the issue with a single click.

2.1.1 Categories of Lint Rules

Many linters are configurable with a set of *rules*, which specify the categories of issues that the linter should detect. These rules can be enabled or disabled by users, allowing them to customise the linter to their needs. Rules are usually grouped by purpose: some rules are concerned with simply improving code style, while others are concerned with detecting suspicious code patterns indicative of potential bugs.

Style checks and code quality

Linters can suggest opportunities to improve code by utilising language features in a more idiomatic manner. Snippets of code that violate these stylistic rules are not necessarily incorrect, but should be fixed as they may be harder to read or maintain in the long term. Furthermore, many idiomatic practices exist to avoid common pitfalls that could lead to unintended behaviour. By highlighting good practices, linters can help users avoid these common mistakes that may cause bugs. For example, *ESLint*[1], one of the most popular JavaScript linters,

[1] https://eslint.org/docs/latest/rules/

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

A well-designed linter can help programmers learn about useful language constructs by suggesting them in the context of their code, aiding them in adhering to best practices and common style conventions. This category of rules is therefore especially helpful as an educational tool for new users of a language, who may be unaware of these idioms. For example, the *Clippy*[2] linter for Rust [Li et al. 2023] 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. 2.1 provides an example of a similar rule in Haskell, from the *HLint*[3] linter. The rule suggests to rewrite the function into an equivalent but more concise form via η-reduction, presented to the user as a code action that can be applied automatically.

```
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`.

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

Domain-specific idioms A library or especially an embedded DSL may require a particular style of usage that is different from the host language [Hora et al. 2012]. The majority of linters are designed for general-purpose application domains, so they are unlikely to detect issues specific to a more specialised domain. Therefore, linters may be developed for specific libraries or DSLs, with their own set of domain-specific rules. In this case, the accompanying linter can benefit users and improve developer productivity in a similar manner to general-purpose linters: 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 the `xunit.analyzers`[4] package which provides linting rules to enforce best practices specific to *xUnit*.

Code smells and opportunities for refactoring

Code refactoring is a well-established practice in software development. In his influential book *Refactoring: Improving the Design of Existing Code* [Fowler 2018], Fowler defines **refactoring** as "the process of changing a software system in such a way that it does not alter the external behaviour 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,

[2] https://doc.rust-lang.org/clippy/
[3] https://hackage.haskell.org/package/hlint
[4] https://github.com/xunit/xunit.analyzers

they may lead to issues such as bugs or poor maintainability if left unchecked. They are conceptually similar to the stylistic issues mentioned earlier, however they may encompass higher-level structural and design-based problems that are not easily fixed by simple stylistic changes. 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.

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 an *Extract Method* [Fowler 2018] refactoring to avoid code duplication: fig. 2.2 demonstrates how this automatic refactoring process can be performed in the IntelliJ IDEA[5] IDE.

Likely bugs or errors

In some cases, linters may be able to detect with relatively high confidence that a piece of code is likely to contain a bug. Rules that detect more egregious issues like this are helpful for even experienced users to avoid common pitfalls. For example, *Clippy* has `clippy::correctness` rules which identify code very likely to be incorrect or useless, and `clippy::suspicious` rules for code that is *very sus* but could be a false positive. *ESLint* also provides several rules to warn against code patterns that are likely to cause runtime errors, such as re-assigning a `const` variable.

Again, linters may attempt to provide code fixes for these issues where possible. However, these issues are usually more complex, which may limit the effectiveness or usefulness of auto-fixes: in the case of a suspicious code pattern, the programmer's intent may not be clear, causing the linter to suggest a fix that does not align with the user's expectations.

2.1.2 Implementing Linters

Static vs Dynamic Analysis

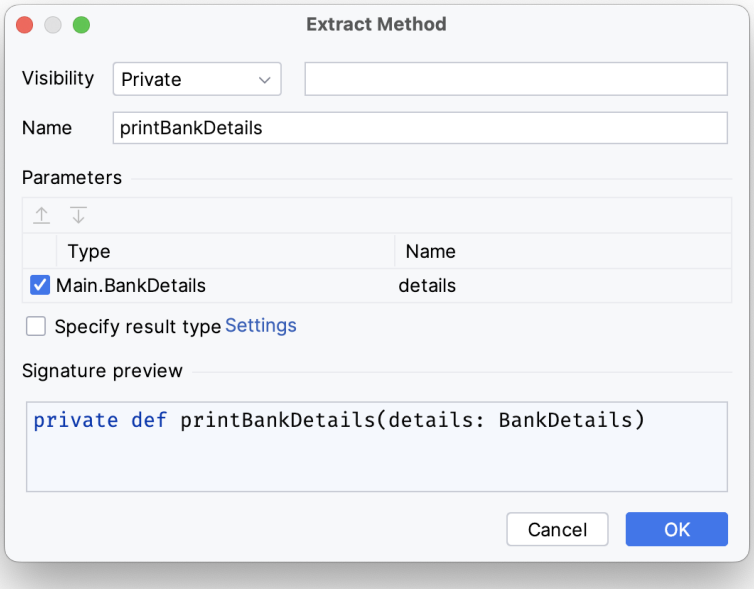
Static analysis analyses source code to extract information about its behaviour without executing the program itself. This is in contrast to *dynamic analysis*, which is performed on programs as they are run. Although dynamic analyses can yield more precise results as they observe the actual runtime behaviour of the program, they are more heavyweight as a result. Generally, linters are static analysis tools, as by design they must be lightweight and fast to provide real-time feedback to users as they write code.

In some instances, however, dynamic languages such as JavaScript present challenges for static analysis due to their dynamic runtime nature. This has led to the exploration of linters that incorporate dynamic analysis techniques to identify issues that static methods cannot detect [Gong et al. 2015]. Regardless, since Scala is strongly and statically-typed it does not suffer from the same downsides, so the focus of this thesis will be on implementing `parsley-garnish` using static analysis.

[5] https://www.jetbrains.com/idea/

```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. 2.2a 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. 2.2b.

Fig. 2.2: An example of the *Extract Method* refactoring in IntelliJ IDEA.

Metaprogramming

Linters that provide code fixes are *metaprograms*: they are programs that manipulate programs. Auto-fixes and refactorings are performed as source-to-source transformations, which 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.

Analyses over ASTs are typically semantic-aware, respecting scoping rules and type information to provide more accurate code fixes. This semantic information is often provided by the compiler, so there's the choice of lint-on-compile vs lint-after-compile https://scalacenter.github.io/scalafix/docs/users/related-projects.html

Code-Fixing Approaches

Our approach is formal methods using domain-specific knowledge of parser laws Not data-driven or ML/LLM approach

2.2 Writing Linters for Scala

DSL linting is hard but luckily Parsley is an eDSL so we can just use Scala metaprogramming utilities

2.2.1 Choice of Tooling

The goal of `parsley-garnish` is to provide linting and refactoring capabilities for the `parsley` parser combinator library. Since `parsley` is a Scala library, this project must be implemented using a tool capable of statically analysing Scala code. This section will therefore discuss and evaluate the choices available for implementing `parsley-garnish`.

Scala compiler plugins The most powerful approach would be to implement `parsley-garnish` as a compiler plugin [Pickering, Wu, and Németh 2019]. Using the low-level compiler API, it is possible to perform arbitrary code transformations at any step of the compilation process. Compiler plugins therefore offer full freedom to extend the Scala compiler with extra functionality, such as extra passes for code analysis and emitting lint warnings as diagnostics or even compiler errors.

However, this approach has several drawbacks. Firstly, compiler plugins are tightly coupled with the compiler itself, and therefore not portable across major compiler versions. For instance, plugins written for the Scala 3 compiler, known as `dotty`, are completely incompatible with Scala 2 plugins [LAMP/EPFL 2022]. Additionally, developing compiler plugins requires a deep understanding of arcane and poorly documented compiler internals. Exposing the full compiler API permits unsafe operations that may violate undocumented invariants assumed by the compiler, leading to exceptions during compilation or even malformed bytecode [Sherwany, Zaza, and Nystrom 2015]. The lack of higher-level abstractions also makes it difficult to implement even trivial tasks such as renaming a field.

For these reasons, it would be preferable to explore other tools that may use compiler plugins themselves but provide a higher-level interface for implementing code analysis and transformations.

Scalameta *Scalameta*[6] is a metaprogramming framework for Scala that provides a unified interface for performing common metaprogramming tasks. Many Scala tools are built with Scalameta, most notably the *Metals*[7] LSP server for Scala. It provides a high-level syntactic API for transforming and pretty-printing Scala source code, as well as a semantic API providing access to semantic information such as type inference and name resolution. Scalameta builds upon the earlier `scala.reflect` metaprogramming framework, unifying runtime and compile-time metaprogramming under a single API. Previously under `scala.reflect`, source code was parsed into lossy trees that discarded syntactic *trivia* such as comments and whitespace [Burmako 2017]. On the other hand, Scalameta trees are fully lossless and preserve all syntactic details: a crucial feature for code transformations and refactorings that must preserve formatting details.

Scalameta's semantic API is powered by *SemanticDB*, a compiler-agnostic data model for semantic information in Scala programs. This allows Scalameta to extract semantic information via compiler plugins that emit data in the SemanticDB format. Thus, Scalameta can work with any compiler that supports SemanticDB, rather than being tied to a specific compiler implementation.

Since Scalameta provides a high-level interface for manipulating syntactic and semantic information, it is a promising choice for this project. Being able to access semantic information is especially helpful for implementing more complex code analyses. However, Scalameta's primary focus is on providing a general metaprogramming framework and therefore lacks API support specifically for implementing linting and refactoring rules. For example, the Scalameta tree transformation utilities do not fully preserve formatting details when pretty-printed, despite the underlying trees containing this information.

Scalafix *Scalafix*[8] is a code linting and rewriting tool built on top of Scalameta. It provides a higher-level API over Scalameta's tree transformation capabilities, specifically designed for implementing fine-grained code transformations that preserve the original formatting of the source file. Scalafix provides a framework for implementing linting rules to emit diagnostic warnings, as well as rewrite rules to perform automated code transformations [Geirsson 2017]. Since it is built on Scalameta, a major advantage of Scalafix is that it is also compiler-agnostic and could be integrated into any IDE if a plugin is developed for it.

Originally, Scalafix was designed to help automate the process of migrating code from Scala 2 to 3, which involved many breaking changes to the language [Geirsson 2016]. However, Scalafix has since evolved into a general-purpose tool for implementing generic code transformations and analyses. Scalafix rules can be either *syntactic* or *semantic* depending on whether they require semantic information, such as type signatures, to perform their analysis [Scala Center 2024]. Syntactic rules are faster to run, operating purely on the AST without requiring compilation, but are more limited in the analyses they can perform. Semantic rules, on the other hand, are more complicated to run: they require augmenting the Scala compiler with the SemanticDB compiler plugin to provide Scalafix with the required semantic information. They are also slower as a result, since they require the source file to be compiled.

A drawback of Scalafix is that it is primarily a command-line tool, and currently does not provide an interactive interface within any code editor. Ideally, users should be able to see inline diagnostics, as well as preview and

[6] https://scalameta.org/
[7] https://scalameta.org/metals/
[8] https://scalacenter.github.io/scalafix/

select auto-fixes interactively like in fig. 2.2. A Metals integration would be the most favourable outcome, as they already share a Scalameta backend, and it would allow Scalafix to be integrated into any code editor supporting the LSP protocol. However, at the time of writing, Scalafix only has basic integrations with Metals to run rules, but no interactive in-editor integrations [Jaglin 2024]. This feature request is an open issue on GitHub[9], but no timeline has been provided for its implementation.

Nonetheless, Scalafix emerges as the most favourable choice for implementing `parsley-garnish`. It provides high-level APIs specifically for implementing linting and rewrite rules without necessitating extensive knowledge of compiler internals. Scalafix is also being actively maintained, with good basic documentation and a growing number of examples of usage in the wild.

Other tools considered The main alternate contender to Scalafix is the IntelliJ Scala Plugin[10]. However, while the plugin offers superior interactive usage within the IntelliJ IDEA IDE, it is tied to the IntelliJ Scala compiler and therefore not portable across other compilers. To maintain flexibility and not tie `parsley-garnish` to a particular compiler or code editor, Scalafix is a preferable option. Furthermore, documentation is less clear on how to write a Scala plugin for IntelliJ compared to the Scalafix documentation.

WartRemover[11] is a linter implemented as a compiler plugin, with support for writing custom rules. However, it only can emit diagnostics and does not support code fixes, making it unsuitable for `parsley-garnish`'s goals.

ScalaStyle[12] is primarily a style checker which also supports custom rules. However, it is only able to perform syntactic analyses and does not have access to semantic information, restricting the types of analyses it can perform.

2.2.2 A Crash Course on Scalafix

This section briefly explains how Scalafix rules are written, providing the Scalafix-specific terminology needed to understand the implementation details of `parsley-garnish`.

Rules

Scalafix rules are implemented as generic top-to-bottom traversals through the Scala AST, represented as a Scalameta `Tree` data structure. This is achieved with a collection-like API, exposing these operations in the style of standard Scala collection methods such as `traverse` and `collect`. During traversal, side effects are generated using the `Patch` data structure, which represents either a lint diagnostic or a code rewrite. Rules are applied file-by-file, and the patches generated by each rule are collected and applied by Scalafix as a batch rewrite after all rules have finished running.

Fig. 2.3 showcases the basic structure of a simple syntactic rule:

- `doc.tree.collect` performs the tree traversal, where AST nodes of interest can be visited via pattern matching.

[9] https://github.com/scalameta/metals-feature-requests/issues/230
[10] https://github.com/JetBrains/intellij-scala
[11] https://www.wartremover.org/
[12] http://www.scalastyle.org/

- `case v: Defn.Val if !v.mods.exists(_.is[Mod.Lazy])` visits a `val` definition node that is not already marked as `lazy`.
- `Patch.addLeft(v, "lazy ")` describes a patch to add the `lazy` modifier to the left side of the node that was matched on.

```
class MakeValsLazy extends SyntacticRule("MakeValsLazy") {
  override def fix(implicit doc: SyntacticDocument): Patch = {
    doc.tree.collect {
      case v: Defn.Val if !v.mods.exists(_.is[Mod.Lazy]) => Patch.addLeft(v, "lazy ")
    }.asPatch
  }
}
```

Fig. 2.3: A syntactic rule that makes all `val` definitions lazy.

Safer Patches Using Quasiquotes

A careful reader may notice from fig. 2.3 that the `Patch` rewrite method receives a raw string value, which seems unsafe and could potentially lead to malformed code. That careful reader would indeed be correct: Scalafix provides no guarantees that the output of a patch is a well-formed program, and it is the rule author's responsibility to ensure so.

The approach taken by `parsley-garnish` is to represent intended rewrites as Scalameta `Tree` objects, and only convert them to strings immediately before applying the patch. Fig. 2.4 shows how *quasiquotes* [Shabalin, Burmako, and Odersky 2013] can be used as syntactic sugar to construct trees in a convenient manner:

- `q"..."` is the quasiquote's string interpolation syntax to build a `Term` node, which is a subclass of `Tree`.
- `$x` splices the value of the expression `x` into the constructed tree. Ordinarily, spliced values need to be an instance of a `Tree`, but simple values such as integers (in this case) can be automatically lifted into the appropriate tree node.
- `Tree.structure` represents the resulting `Tree` data structure as a string, useful for debugging.
- `Tree.syntax` pretty-prints the tree back into its source code representation, which can then be fed into a `Patch`.

Quasiquotes expand at compile-time into `Tree` constructors, so they do not preserve formatting details. This turns out to be very convenient for pretty-printing purposes: often, spliced values may contain superfluous parentheses, which are automatically removed (as seen in the final example within fig. 2.4).

By using quasiquotes instead of raw strings, the syntactic correctness of a rewrite is guaranteed. However, this safety is not foolproof: quasiquotes are not guaranteed to be well-typed or well-scoped, so the rewritten program still might not be able to compile. They are also not *hygienic*: generated code will not be able to avoid name clashes with regular code [Burmako 2017]. The lack of hygiene can cause issues with variable capture, allowing a variable to be unintentionally shadowed by a generated variable. Again, it is the rule author's responsibility to ensure that variable capture does not occur: §3.2 in part discusses how this is handled in `parsley-garnish`.

```scala
val x = 1
val tree = q"$x + 2"

println(tree.structure)
// Term.ApplyInfix(
//   Lit.Int(1),
//   Term.Name("+"),
//   Type.ArgClause(Nil),
//   Term.ArgClause(List(Lit.Int(2)), None)
// )

println(tree.syntax)
// 1 + 2
println(q"($tree)".syntax)
// 1 + 2
```

Fig. 2.4: Using Scalameta quasiquotes to construct a tree.

Semantic Information

Semantic rules are implemented in the same manner as fig. 2.3, but instead take the more powerful `SemanticDocument` as an implicit parameter. This allows rule authors to query AST nodes for their associated symbols: textual identifiers that represent definitions in the Scala program. Although symbols are not guaranteed to be unique outside the document they are defined in [Scalameta 2023], within the context of Scalafix rule authoring, they can be treated as unique since rules are applied per-file.

The following methods are the most important for querying the semantic information associated with a tree node:

- `Tree.symbol` retrieves the symbol associated with a node.
- `Symbol.info` queries a symbol for any `SymbolInformation` attached to it, including metadata such as its type signature.
- `Tree.synthetics` reveals extra compiler-generated tree nodes not present in the original source code, such as implicit conversions or inferred `.apply` methods.

Additionally, the `SymbolMatcher` trait provides a convenient way to create predicates that match specific symbols. Fig. 2.5 demonstrates how symbol matching can be used to detect integer addition operations, emitting a diagnostic warning when found. The `intAdd` symbol matcher only matches the `+` method on integers, so the rule does not emit a diagnostic for addition operations on other types, as seen in fig. 2.5b.

2.3 Parser Combinators

Parsing is the process of extracting structured information from a flat, unstructured representation of the data. Parsers are programs that perform this process, using a specified grammar to determine the structure of the

```scala
class IdentifyIntAddition extends SemanticRule("IdentifyIntAddition") {
  val intAdd = SymbolMatcher.normalized("scala.Int.`+`")

  override def fix(implicit doc: SemanticDocument): Patch = doc.tree.collect {
    case Term.ApplyInfix(_, intAdd(node), _, _) =>
      Patch.lint(IntAdditionLint(node))
  }.asPatch
}

case class IntAdditionLint(tree: Tree) extends Diagnostic {
  override def position: Position = tree.pos
  override def severity: LintSeverity = LintSeverity.Info
  override def message: String = "Found an addition operation on integers"
}
```

(a) A semantic rule that generates lint diagnostics *only* on integer addition operations.

```scala
object Main {
  val stringAdd = "parsley" + "garnish"
  val intAdd = 1 + 2
  val floatAdd = 1.0 + 2.0
}

// Main.scala:3:18: info: [IdentifyIntAddition]:
// Found an addition operation on integers
//   val intAdd = 1 + 2
//                  ^
```

(b) An example program showcasing the rule in fig. 2.5a.

Fig. 2.5: An example showing how semantic rules are semantic-aware on overloaded method names.

data. They are utilised in a variety of applications such as compilers, interpreters, and processing of data storage formats such as JSON and XML.

Traditionally, parsers have either been written by hand or by using parser generator frameworks such as ANTLR [Parr 2013]. Hand-rolling a parser is a tedious process, requiring the programmer to manually implement the parsing algorithm for the grammar. However, this approach is the most powerful and flexible and can provide excellent performance. Alternatively, parser generators lift the burden of implementing the parsing algorithm, instead requiring the programmer to specify the grammar in the format of a domain-specific language (DSL) similar to a high-level grammar. The grammar is then compiled by the parser generator tool to produce a parser in a target language. This approach is less flexible but can be more convenient and less error-prone.

Parser combinators [Hutton 1992], which stem from a functional programming background, are a middle ground between the two approaches. They take the form of an embedded DSL written directly in a general-purpose language, rather than the parser generator approach where the DSL is a separate language. With a parser generator, the provided DSL is often limited in its expressiveness. This is not the case with parser combinators, as the full power of the host language is available to the programmer. This approach also reduces boilerplate code: for example, the programmer does not need to convert between the AST produced by the parser generator and their own AST.

A downside of parser combinators, however, is that they are unstandardised compared to parser generators. Across different implementations, parser combinator APIs can vary significantly, making it difficult to transfer knowledge between different libraries. Experienced users of parser combinators may approach a new library with prior knowledge of general concepts but may have misconceptions about the specifics of the API which can lead to confusion and frustration. This is another motivating reason for the development of `parsley-garnish`, to lower the barrier of entry for new users of the `parsley` library.

2.3.1 Parsley

TODO: proper, worked example showcasing relevant design patterns and stuff which will be picked up by the linter

`Parsley` [Willis and Wu 2018] is a parser combinator library for Scala that provides an API inspired by the `parsec` [Leijen and Meijer 2001] style of parser combinators. This section provides an illustrative example of a simple expression parser to demonstrate what a parser written in `parsley` looks like.

Consider the EBNF grammar for a simple expression language shown in fig. 2.6a. The parser in fig. 2.7 will parse an expression into the AST represented by the Scala datatype in fig. 2.6b.

Notice how the parser closely resembles the high-level EBNF grammar. The main differences of note include the use of:

- `map` to transform the result of a parser to help construct tree nodes consisting of a single value.
- `zipped` to combine the results of two parsers to help construct tree nodes consisting of multiple values.
- `<~` and `~>` operators to guide the direction of parsers.

Except for the possibly cryptic-looking implementation of num to parse a series of digits into an integer, the parser is relatively straightforward to understand.

```
ident ::= "x" | "y" | "z"
num ::= digit+
expr ::= factor "+" expr
factor ::= atom "*" factor
atom ::= ident | num | "(" expr ")"
```

(a) The grammar in EBNF.

```
sealed trait Expr
case class Ident(name: String) extends Expr
case class Num(x: Int) extends Expr
case class Add(x: Expr, y: Expr) extends Expr
case class Mul(x: Expr, y: Expr) extends Expr
```

(b) The Scala AST to parse into.

Fig. 2.6: The grammar and AST for our simple expression language.

```
val ident = "x" | "y" | "z"
val num: Parsley[Int] = digit.foldLeft1(0)((n, d) => n * 10 + d.asDigit)

lazy val expr: Parsley[Expr] = (factor, "+" ~> expr).zipped(Add)
lazy val factor: Parsley[Expr] = (atom, "*" ~> factor).zipped(Mul)
lazy val atom: Parsley[Expr]
  = ident.map(Ident) | num.map(Num) | "(" ~> expr <~ ")"
```

Fig. 2.7: A parser for our simple expression language.

2.3.2 Design Patterns for Parsley

(This background section is a work-in-progress, and will likely expand to include more information about the specific problems and design patterns I choose to explore in the project.)

Willis and Wu [2022] describe several design patterns for writing maintainable parsers using parser combinators in Scala. They identified common problems and anti-patterns in parser design, and proposed solutions in the form of design patterns. This provides a guideline for writing idiomatic parsley code for practical parser design, which enables opportunities for the development of linting and refactoring rules.

This thesis hopes to explore how these common problems can be formalised into code smells and suspicious code patterns that can be automatically detected using linting rules. Some of the design patterns are also theoretically amenable to automated refactoring, which we hope to explore and implement in parsley-garnish.

Misc. things to write (might move to other sections): * Top-down parsers – intro to left-recursion and the types of left recursion in PEG grammars

Chapter 3

Design and Implementation

The purpose of this chapter is to describe the intermediate representations of parsers (§3.1) and functions (§3.2).

3.1 Parser Representation

This is an INTERMEDIATE SYMBOLIC REPRESENTATION (?) more specialised than general-purpose scala ast
This section is about simplifying in our semantic domain (parsers)

Several of the more complex lint rules, most notably §4.6, require manipulating parser combinators in a high-level manner. Scalafix works with code represented as data, in the form of a AST-like representation i.e. Scalameta `Tree` nodes. It would be possible, but extremely cumbersome, to work directly with these AST nodes.

Left-recursion factoring is based on lawful transformations and rearrangements of parsers. There are many instances where given two parsers `p` and `q`, the transformation requires combining them with another combinator, such as `<*>`. We can use quasiquotes `q"p <*> q"` which expands to a `Term` node resembling `ApplyInfix(lhs = p, op = Term("<*>"), args = q)` (in pseudocode for readability purposes). This is ok in terms of ergonomics, but we lose the static inspectability of the individual parsers `p` and `q`.

Instead, represent parsers as an algebraic data type ADT in the same way that Parsley itself uses a deep embedding to represent combinators as objects. Methods on these objects can then be used to manipulate them, and the resulting object can still be pattern matched, maintaining the static inspectability of the parsers. So then it's just like writing parsers in Parsley itself: `p <*> q` constructs a `Ap(p, q)` node which can still be pattern matched on. And similar to Parsley, representing everything as objects makes it easy to optimise using pattern matching on constructors. This representation also then gives us for free the implementation for lint rules such as *Simplify Complex Parsers* rule, which applies parser laws to simplify parsers.

This section explores the motivation behind this and the design choices made in the implementation. Use the §4.6 rule as a basis/context to demonstrate the utility of this representation.

Running example to motivate all requirements for the parser representation – removing left recursion from the following simple parser:

```
lazy val expr: Parsley[String] = (expr, string("a")).zipped(_ + _) | string("b")
```

3.1.1 Detecting Top-Level Parsers

Each source file is represented by Scalafix as a large AST. `parsley-garnish` would benefit from a higher-level abstraction. A more useful representation for `parsley-garnish` is a map of all parsers defined within that file, indexed by the symbol of the definition.

To achieve this, it is necessary to identify the AST nodes of interest corresponding to Parsley parsers. This involves pattern matching on `val`, `var`, and `def` definitions with a type inferred to be of form `Parsley[_]`. Consider `expr`, which is a top-level definition of a parser:

```
lazy val expr: Parsley[String] = (expr, string("a")).zipped(_ + _) | string("b")
//       ^^^^^^^^^^^^^^^^^^^^   ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
//         key: expr.symbol          RHS of definition: convert to a Parser
```

The right hand side of the definition is a Scalameta `Term` node representing the definition of the parser. During traversal, the right hand side of the definition is converted to a `Parser` instance. This is then accumulated in a map, with the symbol of the definition as the key. A full traversal through a source file obtains all the non-terminals in the grammar defined within that file.

3.1.2 Converting Scalameta Terms to the Parser ADT

Having identified the AST nodes which represent parsers, they need to be transformed into the appropriate `Parser` representation. This involves pattern matching on the `Term` to determine which parser combinator it represents, and then constructing the appropriate `Parser` instance.

Each Parser defines a partial function `fromTerm` which creates an instance of that parser from the appropriate Scalameta `Term`. These `fromTerm` methods define a `toParser` extension method on Scalameta `Term`s to transform them to the appropriate parser. Use Scalafix's `SymbolMatcher` to match tree nodes that resolve to a specific set of symbols. This makes use of semantic information from SemanticDB, so we are sure that a `<*>` is actually within the `parsley.Parsley` package, rather than some other function with the same name. This is much more robust compared to HLint, which suffers from false positives due to its reliance on syntactic information only.

The top-level combinator that makes up `expr`'s definition is the choice combinator, `|`. Taking a look at the Scalameta `Term` representing this definition `(expr, string("a")).zipped(_ + _) | string("b")` (this is cleaned up to hide the large term representing the LHS of the choice):

```
Term.ApplyInfix(
  Term.Apply(...), // (expr, string("a")).zipped(_ + _)
  Term.Name("|"),
  Type.ArgClause(List()),
  Term.ArgClause(
    List(
      Term.Apply(
        Term.Name("string"),
        Term.ArgClause(List(Lit.String("b")), None)
      )
    ),
    None
  )
)
```

The goal is to pattern match on this term to construct a `Choice` node, which represents the choice combinator. Thus, `Choice.fromTerm` should be defined to match on an infix application on a `|` (or `<|>`) combinator:

```
object Choice {
  val matcher = SymbolMatcher.normalized("parsley.Parsley.`|`", "parsley.Parsley.`<|>`")

  def fromTerm(implicit doc: SemanticDocument): PartialFunction[Term, Choice] = {
    case Term.ApplyInfix(p, matcher(_), _, Term.ArgClause(List(q), _)) =>
```

```
      Choice(p.toParser, q.toParser)
  }
}
```

If it finds such a term, it constructs a `Choice` node, and recurses on `p` and `q` to also convert them to parser nodes. After all sub-terms also get converted, the `expr` parser is represented as:

```
Choice(
  Zipped(Function(_ + _), List(NonTerminal(expr), Str(a))),
  Str(b)
)
```

For now we are only concerned with the parser representation; we will touch upon how `Function`s are represented in the next section.

3.1.3 Building New Parsers From Existing Parsers

Now that we have parsers represented as an ADT, we can easily build new parsers from existing parsers. This is crucial for the left-recursion factoring rule, which "unfolds" parsers into separate parsers representing the left-recursive and non-left-recursive parts. These are then recombined to form parsers which are free from left recursion.

Make this even easier by utilising Scala's ability to define infix operators, define them as extension methods on the `Parser` trait. For example:

```
implicit class ParserOps(private val p: Parser) extends AnyVal {
  def <*>(q: Parser): Parser = Ap(p, q)
  def <|>(q: Parser): Parser = Choice(p, q)
  def map(f: Function): Parser = FMap(p, f)
}
```

This makes it more ergonomic to manipulate parsers, it's like we're writing Parsley code itself. A small example snippet from the `unfold` method on the `Ap` parser:

```
val lefts = {
  val llr = pl.map(flip) <*> q
  val rlr = pe.map(f => ql.map(composeH(f))).getOrElse(Empty)
  llr <|> rlr
}
```

Notice how the code closely resembles the high-level description of the transformation, using `<*>`, `<|>`, `map`, operators.

3.1.4 Simplifying Parsers Using Parser Laws

Once all the unfolded parsers have been recombined, the raw output is very noisy and difficult to read. Again, ignore the functions, these will be covered in §3.2.

```
lazy val expr: Parsley[String] = chain.postfix(
  empty | (empty.map(a => b => a + b) | empty <*> expr) <*> string("a")
    | string("b") | empty
```

```
)(
  (empty.map(FLIP) <*> expr | pure(ID).map(COMPOSE(a => b => a + b)))
    .map(FLIP) <*> string("a")
    | empty | empty
)
```

This is obviously unacceptable and completely obfuscates the intent of the parser. For human readability of the transformed output, it is therefore important to simplify the parser as much as possible. Now that the parsers are represented as objects, it is easy to pattern match on their constructors. This improved static inspectability allows us to perform simplifications using the laws that govern parser combinators – these often form a natural simplification in one direction. This is similar to the high-level optimisations performed in the Parsley backend as described by Willis, Wu, and Pickering [2020], using the same parser laws.

Fig. 3.1 shows the subset of parser laws utilised by `parsley-garnish` for parser simplification. Most of the laws in fig. 3.1 have already been shown to hold for Parsley in [Willis and Wu 2018]; an additional proof for eq. (3.8) can be found in appendix A.

$$\texttt{p.map(f).map(g)} = \texttt{p.map(g compose f)} \tag{3.1}$$
$$\texttt{pure(f) <*> pure(x)} = \texttt{pure(f(x))} \tag{3.2}$$
$$\texttt{pure(f) <*> x} = \texttt{x.map(f)} \tag{3.3}$$
$$\texttt{empty | u} = \texttt{u} \tag{3.4}$$
$$\texttt{u | empty} = \texttt{u} \tag{3.5}$$
$$\texttt{pure(x) | u} = \texttt{pure(x)} \tag{3.6}$$
$$\texttt{empty <*> u} = \texttt{empty} \tag{3.7}$$
$$\texttt{empty.map(f)} = \texttt{empty} \tag{3.8}$$

Fig. 3.1: Functor (3.1), Applicative (3.2, 3.3), and Alternative (3.4–3.8) laws.

In the previous example, it is evident that the most noise results from the `empty` combinators. These can be eliminated using eqs. (3.4), (3.5), (3.7), and (3.8):

```
lazy val expr: Parsley[String] = chain.postfix(string("b"))(
  (pure(ID).map(COMPOSE(a => b => a + b))).map(FLIP) <*> string("a")
)
```

The complicated term in the postfix operator can then be simplified as follows:

```
    (pure(ID).map(COMPOSE(a => b => a + b))).map(FLIP) <*> string("a")
=       { eqs. (3.2) and (3.3) }
    pure(COMPOSE(a => b => a + b)(ID)).map(FLIP) <*> string("a")
=       { eqs. (3.2) and (3.3) }
    pure(FLIP(COMPOSE(a => b => a + b)(ID))) <*> string("a")
=       { eq. (3.3) }
    string("a").map(FLIP(COMPOSE(a => b => a + b)(ID)))
```

This results in the most simplified form of the parser:

```
val f: Function = FLIP(COMPOSE(a => b => a + b)(ID))
lazy val expr: Parsley[String] = chain.postfix(string("b"))(string("a").map(f))
```

Implementation

These simplifications are applied akin to peephole optimisations in a bottom-up traversal of the recursive `Parser` ADT. There are many instances of Parsers, so this leads to a lot of boilerplate code recursing through each case, which is prone to error when we implement it. To avoid this, we decouple the application of a generic transformation function from the recursive traversal through the datatype. This is still a hand-written traversal, but heavily inspired by the generic traversal patterns in [Mitchell and Runciman 2007]. Bottom-up transformation takes a partial function, applying the transformation at nodes where it is defined. The resulting `transform` method on parsers resembles the following (only a few cases shown for brevity):

```
def transform(pf: PartialFunction[Parser, Parser]): Parser = {
  val p = this match {
    case Ap(p, q)      => Ap(p.transform(pf), q.transform(pf))
    case Zipped(f, ps) => Zipped(f, ps.map(_.transform(pf)))
    case Pure(f)       => Pure(f)
    ...
  }
  if (pf.isDefinedAt(p)) pf(p) else p
}
```

There is also a need for a `rewrite` method to apply a transformation exhaustively until a normal form is reached. This is implemented in terms of `transform`, applying the partial function everywhere and re-applying it until it no longer makes a change.

```
def rewrite(pf: PartialFunction[Parser, Parser]): Parser = {
  def pf0(p: Parser) = if (pf.isDefinedAt(p)) pf(p).rewrite(pf) else p
  this.transform(pf0)
}
```

Therefore, any transformation on parsers can be defined without having to worry about any recursive traversal boilerplate. Using `rewrite`, parser simplification can then be expressed in a clean and maintainable manner:

```
def simplify: Parser = this.rewrite {
  // p.map(f).map(g) == p.map(g compose f)
  case FMap(FMap(p, f), g) => FMap(p, composeH(g, f))
  // u <|> empty == u
  case Choice(u, Empty) => u
  // pure(f) <|> u == pure(f)
  case Choice(Pure(f), _) => Pure(f)
  ...
}
```

Additionally, the `Parser` trait is sealed, so there will be compiler warnings if a new case is added and the `transform` method is not updated. Overall, this approach still requires a hand-written traversal so it is more error-prone than a generic derivation. However, that would require usage of an external library such as `shapeless`[1], which is

[1] https://github.com/milessabin/shapeless

not desired as the complexity of the ADT is not high enough to warrant bringing in an extra dependency just for this purpose.

3.1.5 Converting Parsers Back to Scalameta Terms

After the transformations on parsers are complete, they need to be converted back to a textual representation to be applied as a Scalafix patch. It is actually rather trivial to do so, by borrowing the pretty-printing capabilities of Scalameta terms. This transformation is thus the inverse of the `fromTerm` transformation. This can be written using Scalameta quasiquotes to construct the `Term` nodes. The `Parser` trait defines this transformation as the method `term`, for example:

```
case class Zipped(func: Function, parsers: List[Parser]) extends Parser {
  val term: Term = q"(..${parsers.map(_.term)}).zipped(${func.term})"
}
```

3.2 Function Representation

This section is about simplifying in the general domain, so really Squid can do all of this? Still an interesting approach ig And shows a shortcoming of scalameta quasiquotes

The work in §3.1 showed that it is possible to lift AST nodes corresponding to parser terms into a more statically inspectable `Parser` datatype. Furthermore, §3.1.4 demonstrated that this representation allows parsers to be transformed into a simpler form. However, so far, the functions passed to lifting combinators have been treated in an opaque manner, and have not been subject to the same simplifications as parsers. This is unsatisfactory, as these functions are essential to combining and sequencing parsers to build more complex parsers.

Additionally, the transformations on parsers during left-recursion factoring involve mapping over parsers with higher-order functions such as `flip` to reverse the order of arguments, and `compose` to compose functions. Looking back at the running example in the previous section, the function passed to the `map` combinator has ended up in an unnecessarily complex form:

```
val f: Function = FLIP(COMPOSE(a => b => a + b)(ID))
```

It would be ideal to make functions inspectable in a similar manner to parsers, so that they can be *normalised* into a semantically equivalent but visually simpler form.

3.2.1 Motivation

Where are functions used in parsley? Lowest-level: map and family

3.2.2 Approach

It may be useful to view this as statically evaluating as much as possible – however, not all inputs are known so we cannot fully evaluate the result value. Normalisation is therefore just evaluation, but in the presence of unknown terms.

Assume functions passed to parsers are pure, because we assume sane users. So it is reasonable to model functions as a lambda calculus, and this can also be achieved with an ADT. As an aside, this is fine – Scala 3 has been formalised to the DOT calculus, and the subset that we are interested in is equivalent to the lambda calculus. In the simple lambda calculus all functions are curried i.e. each function only takes one argument, and multi-argument functions are represented as a chain of single-argument functions. In Scala, uncurried functions are preferred for performance reasons, but it allows currying as functions can have multiple parameter lists. We could desugar all this and curry the functions, but we want to be able to easily transform our output back into Scala code. So we extend the lambda calculus to be able to represent multi-argument functions using n-ary abstraction and application.

```
trait Function

trait Lambda extends Function
case class Abs(xs: List[Var], f: Function) extends Lambda
case class App(f: Function, xs: List[Function]) extends Lambda
case class Var(name: VarName) extends Lambda
case class Translucent(t: Term, env: Map[VarName, Function]) extends Lambda

case object Identity extends Function
case object Flip extends Function
case object Compose extends Function
```

Normalisation by Rewriting

Essentially term-rewriting, similar to how Parsers are simplified. Strong reduction – allow beta reduction to occur deep inside lambda terms, in all redexes of a term, until no more reductions can be made. To beta normal form, not just beta-WHNF (weak head normal form)

Aside: Untyped lambda calculus is not strongly normalising, but these terms are converted from valid typed Scala programs. Although not formally proven, it is reasonable to assume that these terms are a subset of the simple lambda calculus that are strongly normalising – they won't be particularly complex.

`Function` has one binding construct – the n-ary abstraction. This introduces variables and the need for capture-avoiding substitution. Capture-avoiding substitution is hard, many approaches e.g. de Bruijn indices (but inefficient to open/close terms so much, so the correct choice is levels). Attempted a barendregt's but the optimised form (not freshening on every substitution) might be subtly incorrect, not sure tbh.

Normalisation by Evaluation

Fortunately, there is a cheat to completely circumvent the nightmare of capture-avoiding substitution. Unlike traditional normalisation techniques, NbE bypasses rewriting entirely, instead appealing to their denotational semantics. (fuck yeah bitches) Interpret the term into a denotational model which can then be evaluated, and reify the result back into a term. Use HOAS to leverage the meta-language (i.e. Scala) as our semantic model – blazingly fast! (theoretically lol) No rewriting = no worrying about avoiding capture. Shift that burden onto the scala compiler.

Add diagram showing $[\![-]\!]$

3.2.3 Converting Scalameta Terms to the Function ADT

Three cases:

1. `Term.Function` nodes, representing lambda expressions
2. `Term.AnonymousFunction` nodes, representing lambda expressions using placeholder syntax
3. Any other term, which will be *eta*-expanded if possible

Lambda Expressions

Often found as relatively simple functions to glue together parsers, or transform the result of a parser:

```
val asciiCode: Parsley[Int] = item.map(char => char.toInt)
```

Choose a particularly convoluted example:

```
a => (a, b) => a + b
```

In this example, the a in the function body refers to the a in the second parameter list, as it shadows the a in the first parameter list.

Conversion "flattens" the lambda expression into a chain of n-ary abstractions, with the final term being the body of the lambda. Because we lose scoping information this way, perform α-conversion on variables. The body becomes a `Translucent` term.

```
λ(_l1). λ(_l2, _l3). Translucent(_l2 + _l3, env = {_l1 → _l1, _l2 → _l2, _l3 → _l3})
```

Values in bold are Scalameta tree nodes, so the body term's environment maps `Term.Name` nodes to their corresponding variable terms.

Placeholder Syntax

Scala supports placeholder syntax for terseness, so the earlier parser can be rewritten as:

```
val asciiCode: Parsley[Int] = item.map(_.toInt)
```

Eta-Expansion

```
case class AsciiCode(code: Int)
object AsciiCode extends ParserBridge1[Char, AsciiCode] {
  def apply(char: Char): AsciiCode = AsciiCode(char.toInt)
}
val asciiCode = AsciiCode(item) // desugars to item.map(AsciiCode.apply)
```

3.2.4 Normalising Function Terms

```
trait HOAS
case class HAbs(n: Int, f: List[HOAS] => HOAS) extends HOAS
case class HApp(f: HOAS, xs: List[HOAS]) extends HOAS
case class HVar(name: VarName) extends HOAS
case class HTranslucent(t: Term, env: Map[VarName, HOAS]) extends HOAS
```

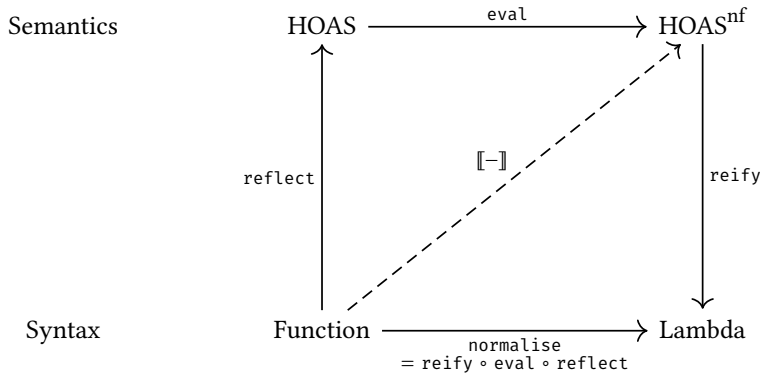


Fig. 3.2: Normalisation by evaluation for the `Function` datatype.

Abs represents n-ary λ-abstraction; App is n-ary function application; Var represents opaque free variables. Translucent is treated as opaque during normalisation, but contains normalisable terms in an environment, where they are substituted back in after normalisation.

Reflection

```
trait Function {
  def reflect: HOAS = {
    def reflect0(func: Function, boundVars: Map[Var, HOAS]): HOAS = func match {
      case Abs(xs, f) =>
        HAbs(xs.size, vs => reflect0(f, boundVars ++ xs.zip(vs)))
      case App(f, xs) =>
        HApp(reflect0(f, boundVars), xs.map(reflect0(_, boundVars)))
      case v @ Var(name) =>
        boundVars.getOrElse(v, HVar(name))
      case Translucent(term, env) =>
        HTranslucent(term, env.map { case (v, func) => v -> reflect0(func, boundVars) })

      case Id => HAbs(f => f)
      case Flip => HAbs(f => HAbs(x => HAbs(y => HApp(HApp(f, y), x))))
      case Compose => HAbs(f => HAbs(g => HAbs(x => HApp(f, HApp(g, x)))))
    }

    reflect0(this, Map.empty)
  }
}
```

Evaluation

```
trait HOAS {
  def eval: HOAS = this match {
    case HAbs(n, f) => HAbs(n, x => f(x).eval)
    case HApp(f, x) => f.whnf match {
      case HAbs(_, g) => g(x).eval
      case g          => HApp(g.eval, x.map(_.eval))
    }
    case HTranslucent(t, env) => HTranslucent(t, env.map { case (k, v) => k -> v.eval })
```

```
    case _ => this
  }

  private def whnf: HOAS = this match {
    case HApp(f, x) => f.whnf match {
      case HAbs(_, g) => g(x).whnf
      case g          => HApp(g, x)
    }
    case HTranslucent(t, env) => HTranslucent(t, env.map { case (k, v) => k -> v.whnf })
    case _ => this
  }
}
```

Reification

```
trait HOAS {
  def reify: Function = this match {
    case HAbs(n, f) =>
      val params = (1 to n).map(_ => Var.fresh()).toList
      Abs(params, f(params.map(x => HVar(x.name))).reify)
    case HApp(f, xs) => App(f.reify, xs.map(_.reify))
    case HVar(name) => Var(name)
    case HTranslucent(t, env) => Translucent(t, env.map { case (v, f) => v -> f.reify })
  }
}
```

3.2.5 Converting Functions back to Scalameta Terms

Surprise bitches same shit again, quasiquotes ftw

3.2.6 Further Work?

Eta reduction – this is more complicated than in Haskell since Scala has special syntax Partial evaluation, not just normalisation (if we reduce to fully closed terms 1+1 can we get it to evaluate to 2? – except currently this would be a Translucent term)

TODO: Abstraction built over scalafix/meta ASTs to represent functions. Allows us to statically evaluate function composition/flipping etc, so it doesn't turn into one big mess – again, human readability of the transformed output is the goal. Abstraction is again an ADT as a lambda calculus, but with parameter lists so not everything is curried. îdk, this is still a work-in-progress. Seems that there might not be enough time to uncurry the leftrec analysis so this design decision might not be super important. Representation as a lambda calc has allocation overhead, but greatly simplifies function evaluation via beta reduction, instead of having to deal with high-level representations of compose/id (not too bad tbh) and flip (annoying). Also attempted to make it typed but that didn't go so well with Scala's limitations on type inference.

* Extracting method arguments (alongside their types) is very painful * Need to unify information from signature (within symbolinformation) and synthetics * synthetics exist in certain cases: .apply methods, showing the concrete type of a generic argument, implicit conversions * from https://scalacenter.github.io/scalafix/docs/developers/semantic-

tree.html: SemanticTree is a sealed data structure that encodes tree nodes that are generated by the compiler from inferred type parameters, implicit arguments, implicit conversions, inferred .apply and for-comprehensions.

* Don't have full access to type information - can do more work here theoretically, but its difficult and error-prone * So we don't model a typed lambda calculus, just have it untyped

Approaches - AVOIDING capture via substitution * Substitution approaches * De Bruijn indices - inefficient to open/close terms so much - De Bruijn levels as an alternative * HOAS * Normalisation by evaluation

3.3 Limitations

Tried to make Parser and Function typed, but it didn't work out due to limitations in Scala's type inference with GADTs.

Chapter 4

Lint Rules in parsley-garnish

Catalogue of lint rules implemented.

Categorise these – but also somehow split into the ”simple” rules and the ”complex” rules. Simple rules can consist of a single heading, containing: * Explanation of the rule * Simple example to show a diagnostic, and a before and after if it's fixable * How it's implemented in the code * Proof (if applicable) * Limitations

Simple rule ideas: * Overly complex parser definitions * Manually calling implicitSymbol instead of using the implicit

Not sure how to lay out the complex rules yet – so far this is just the left-recursion removal rule. The other complex rule(s) will likely share implementation details with the Parser/Func representation, so work from there.

4.1 Avoid Redefining Existing Parsers

* Catch cases when user manually writes out a parser that is already defined in the library

4.2 Simplify Complex Parsers

* Apply parser laws, re-using Parser and Func representations to do cool things <- should this be a separate rule?

4.3 Ambiguous Implicit Conversions

Problem

Implicit conversions are a powerful feature in Scala, allowing users to supply an argument of one type when another is expected, to reduce boilerplate. As noted by Willis and Wu [2022], implicit conversions are particularly useful for designing DSLs. In the context of parser combinators, they introduce the usage of implicit conversions to automatically lift string and character literals into parsers in the *Implicit Conversions* design pattern. This eliminates the need to explicitly wrap these elements in combinators: `string("parsley") | string("garnish")` can now be expressed as just `"parsley" | "garnish"`, more closely resembling the style of a BNF grammar.

The *Implicit Lexer* pattern is a further specialisation of this approach, hiding the boilerplate of whitespace handling entirely within a `lexer` object. This design pattern allows whitespace handling to be encapsulated as private combinators within the `lexer` object, which are then made available only through implicit conversions automatically applied by the Scala compiler.

However, due to their utility, implicit conversions are also an easily abused feature of Scala. They can obscure the flow of the program, making it difficult to understand what the code is doing and potentially hiding side effects or costly operations. A downside particularly relevant to Parsley is that implicit conversions often lead to confusing error diagnostics when the compiler is unable to resolve them.

One common issue arises from ambiguous implicits when there are multiple implicit conversions in scope. Parsley provides `stringLift` and `charLift` combinators in the `parsley.syntax.character` package for the *Implicit Conversions* pattern, and exposes an `implicitSymbol` combinator for lexers to use in the *Implicit Lexer* pattern. For novice users, it is easy to accidentally import both sets of these implicits, when it is likely that they only intended to use the `implicitSymbol` implicit. For example, consider the following code snippet:

```
val p = 'g' ~> "arnish"
p.parse("garnish")
// [error] type mismatch;
//   found   : String("arnish")
//   required: Parsley[?]
//  Note that implicit conversions are not applicable because they are ambiguous:
//   both method stringLift in object character of type (str: String): Parsley[String]
//   and method implicitSymbol in class ImplicitSymbol of type (s: String): Parsley[Unit]
//   are possible conversion functions from String("arnish") to Parsley[?]
//    val p = 'g' ~> "arnish"
//                   ^^^^^^^^
```

Here, the compiler provides a detailed error message indicating the ambiguity between two possible implicit conversions. However, the compiler is not always able to report such issues clearly. For instance, switching the position of the intended implicit conversion results in a less helpful message:

```
val p = "garnis" <~ 'h'
p.parse("garnish")
// [error] value <~ is not a member of String
//    val p = "garnis" <~ 'h'
//            ^^^^^^^^^^
```

Solution

Ideally, this issue would be addressed by implementing a lint-on-compile rule, which could annotate the compiler error message at the exact location of the issue. If this were implemented as a compiler plugin, partial information available from the compiler stages before the error could potentially provide enough detail to identify the exact clashing implicits. This approach would allow leveraging domain knowledge to update the error message with more useful Parsley-specific diagnostics.

Incidentally, WartRemover has a related lint rule for implicit conversions[1], although it only targets the locations where implicit conversions are *defined*, not where they are *applied*. Despite this limitation, it serves as a proof of concept demonstrating the feasibility of such an approach.

Unfortunately, Scalafix restricts usage to only syntactic rules on the bare AST or semantic rules that operate fully post-compilation. Since the ambiguous implicit conversions will cause compilation failures, this lint must be implemented as a syntactic rule. Consequently, the solution takes a different approach: estimating the presence of clashing implicits by examining their import statements within each scope.

[1] http://www.wartremover.org/doc/warts.html#implicitconversion

Example

Fig. 4.1 extends the previous example to a full Scala source file following the *Implicit Lexer* pattern, but where the user has erroneously additionally imported the `stringLift` implicit from the *Implicit Conversions* pattern. This results in the Scala compiler throwing an error on line 6 due to ambiguous implicits. When run on this file, `parsley-garnish` will report a warning at line 3 similar to that shown in fig. 4.2.

```
1  object parser {
2    import parsley.syntax.character.stringLift
3    import lexer.implicits._
4
5    val p = "garnis" <~ 'h'
6  }
7
8  object lexer {
9    import parsley.token.Lexer, parsley.token.descriptions.LexicalDesc
10
11   private val lexer = new Lexer(LexicalDesc.plain)
12   val implicits = lexer.lexeme.symbol.implicits
13 }
```

Fig. 4.1: A minimal Parsley program which fails to compile due to ambiguous implicits in the `parser` object.

```
warning: [AmbiguousImplicitConversions] This import may cause clashing implicit conversions:
* import parsley.syntax.character.stringLift at line 2
* import lexer.implicits._ at line 3
If this is the case, you may encounter confusing errors like 'method is not a member of String'.
To fix this, ensure that you only import a single implicit conversion.

  import lexer.implicits._
  ^^^^^^^^^^^^^^^^^^^^^^^^
```

Fig. 4.2: The warning message produced by the `AmbiguousImplicitConversions` lint rule.

Implementation

Unlike Java, Scala offers more flexibility with import statements, allowing them to appear anywhere in source files rather than just at the top. Scala's import statements are lexically scoped, allowing their visibility to be limited to a single class, object, or function. Additionally, Scala processes import statements in a top-down order within the file, further restricting their visibility, as scopes above an import cannot see the imports defined below them.

* Scalafix (via scalameta) provides a generic traversal of the AST: filter to find all import statements in top-down order * This allows the scope to be lexically managed – traversal in the same order that the compiler reads imports * The ancestor AST node of an import statement is its enclosing scope * Use ancestor information to determine which of the visited imports are in scope at that point

* to find stringLift: Pattern match to find if import is of form ‘import parsley.syntax.character._‘ * to find implicit lexer: pattern match to find if there is an importee called ‘implicitSymbol‘ or if an import contains keywords ‘lexer‘ and ‘implicit(s)‘

* if at any point in the traversal, both types of imports are in scope, report a warning

4.4 Remove Explicit Usage of Implicit Conversions

4.5 Refactor to use Parser Bridges

* This would be cool, idk if I have time though, but this should also piggyback off of Func * the pos bridges don’t actually exist, so we can ignore that case and just say its too much code synthesis * shouldn’t be too bad? idk * indicate limitations that this will only work if the ADT is defined in the same file, in order to extend it

4.6 Left Recursion Factoring

Chapter 5

Evaluation

Chapter 6

Related Work

Library-specific linters are rare because they require a non-trivial amount of work to implement. Even more niche is linters for embedded DSLs. This thesis can also be seen as a rough framework for implementing a linter for an eDSL (esp. Parser datatype to represent DSL model). afaik nothing this complex has been done in scalafix

Related to scalafix, so it could be possible to do very similar things as I've done: Rust (I can see this as possible, ask Boogle people for thoughts?) – Clippy lint passes C# – Roslyn

Library-specific linters: xunit.analyzers – seems to be relatively simple one-off rules though, not a DSL

Things that could help: Squid quasiquotes: type-safe and hygienic quasiquotes – but only for scala 2.11/12 * squid quasiquotes have rudimentary function inlining features https://infoscience.epfl.ch/record/231700 Scala 3 macros: I wonder if this could help?

Related concepts to the Function stuff: Metaprogramming and multi-staged programming? a dual: instead of for optimisation/code-generation purposes, we do it just for stringifying Actually I feel like I've borrowed a lot of stuff from metaprogramming optimisation techniques – Haskell Parsley itself

Chapter 7

Conclusion

7.1 Review

7.2 Future Work

Bibliography

Burmako, Eugene (Mar. 2017). "Unification of Compile-Time and Runtime Metaprogramming in Scala". Ph.D. Thesis. Lausanne: EPFL. DOI: 10.5075/epfl-thesis-7159. URL: https://doi.org/10.5075/epfl-thesis-7159.

Fowler, Martin (2018). *Refactoring: Improving the Design of Existing Code*. Addison-Wesley. ISBN: 9780134757681.

Geirsson, Ólafur (Oct. 2016). *Introducing Scalafix: a code migration tool for Scala*. URL: https://scala-lang.org/blog/2016/10/24/scalafix.html (visited on 01/25/2024).

— (Nov. 2017). *Catch bugs with Scalafix v0.5*. URL: https://www.scala-lang.org/blog/2017/09/11/scalafix-v0.5.html (visited on 01/25/2024).

Gong, Liang et al. (July 2015). "DLint: dynamically checking bad coding practices in JavaScript". In: *Proceedings of the 2015 International Symposium on Software Testing and Analysis*. ISSTA 2015. Baltimore, MD, USA: Association for Computing Machinery, pp. 94–105. ISBN: 9781450336208. DOI: 10.1145/2771783.2771809. URL: https://doi.org/10.1145/2771783.2771809.

Gregor, Douglas and Sibylle Schupp (2006). "STLlint: lifting static checking from languages to libraries". In: *Software: Practice and Experience* 36.3, pp. 225–254. DOI: 10.1002/spe.683. URL: https://doi.org/10.1002/spe.683.

Gunasinghe, Nadeeshaan and Nipuna Marcus (2022). *Language Server Protocol and Implementation: Supporting Language-Smart Editing and Programming Tools*. en. Berkeley, CA: Apress. ISBN: 978-1-4842-7791-1. DOI: 10.1007/978-1-4842-7792-8. URL: https://doi.org/10.1007/978-1-4842-7792-8 (visited on 01/25/2024).

Hora, André et al. (2012). "Domain specific warnings: Are they any better?" In: *2012 28th IEEE International Conference on Software Maintenance (ICSM)*, pp. 441–450. DOI: 10.1109/ICSM.2012.6405305. URL: https://doi.org/10.1109/ICSM.2012.6405305.

Hudak, Paul (Dec. 1996). "Building domain-specific embedded languages". In: *ACM Comput. Surv.* 28.4es, 196–es. ISSN: 0360-0300. DOI: 10.1145/242224.242477. URL: https://doi.org/10.1145/242224.242477.

Hutton, Graham (July 1992). "Higher-order functions for parsing". In: *Journal of Functional Programming* 2.3, pp. 323–343. ISSN: 1469-7653. DOI: 10.1017/s0956796800000411. URL: https://doi.org/10.1017/S0956796800000411.

Jaglin, Brice (Feb. 2024). *Unleashing Scalafix potential with custom rules*. en. Nantes, France. URL: https://scala.io/talks/unleashing-scalafix-potential (visited on 03/25/2024).

Johnson, Stephen C (1978). *Lint, a C program checker*. Technical Report 65. AT&T Bell Laboratories.

Kurbatova, Zarina et al. (2021). "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)*, pp. 14–17. DOI: 10.1109/asew52652.2021.00016. URL: https://doi.org/10.1109/asew52652.2021.00016.

LAMP/EPFL (Nov. 2022). *Changes in Compiler Plugins*. URL: https://docs.scala-lang.org/scala3/reference/changed-features/compiler-plugins.html (visited on 01/25/2024).

Leijen, Daan and Erik Meijer (July 2001). *Parsec: Direct Style Monadic Parser Combinators for the Real World*. Tech. rep. UU-CS-2001-27. User Modeling 2007, 11th International Conference, UM 2007, Corfu, Greece, June

25-29, 2007. URL: https://www.microsoft.com/en-us/research/publication/parsec-direct-style-monadic-parser-combinators-for-the-real-world/.

Li, Chunmiao et al. (Oct. 2023). *Unleashing the Power of Clippy in Real-World Rust Projects*. DOI: 10.48550/arXiv.2310.11738. URL: https://doi.org/10.48550/arXiv.2310.11738 (visited on 01/25/2024).

Mitchell, Neil and Colin Runciman (2007). "Uniform boilerplate and list processing". In: *Proceedings of the ACM SIGPLAN Workshop on Haskell Workshop*. Haskell '07. Freiburg, Germany: Association for Computing Machinery, pp. 49–60. ISBN: 9781595936745. DOI: 10.1145/1291201.1291208. URL: https://doi.org/10.1145/1291201.1291208.

Parr, Terence (Jan. 2013). *The Definitive ANTLR 4 Reference*. 2nd. Pragmatic Bookshelf. ISBN: 1934356999.

Pickering, Matthew, Nicolas Wu, and Boldizsár Németh (2019). "Working with source plugins". In: *Proceedings of the 12th ACM SIGPLAN International Symposium on Haskell*. Haskell 2019. Berlin, Germany: Association for Computing Machinery, pp. 85–97. ISBN: 9781450368131. DOI: 10.1145/3331545.3342599. URL: https://doi.org/10.1145/3331545.3342599.

Renggli, Lukas et al. (2010). "Domain-Specific Program Checking". en. In: *TOOLS'10: Proceedings of the 48th International Conference on Objects, Models, Components, Patterns*. Ed. by Jan Vitek. Vol. 6141. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, pp. 213–232. ISBN: 978-3-642-13953-6. DOI: 10.1007/978-3-642-13953-6_12. URL: https://doi.org/10.1007/978-3-642-13953-6_12.

Scala Center (Feb. 2024). *Scalafix: Developer Guide*. en. URL: https://scalacenter.github.io/scalafix/docs/developers/tutorial.html (visited on 03/25/2024).

Scalameta (Nov. 2023). *Scalameta: SemanticDB Specification*. en. URL: https://scalameta.org/docs/semanticdb/specification.html (visited on 03/25/2024).

Shabalin, Denys, Eugene Burmako, and Martin Odersky (Mar. 2013). *Quasiquotes for Scala*. Tech. rep. EPFL, p. 15. URL: http://infoscience.epfl.ch/record/185242.

Sherwany, Amanj, Nosheen Zaza, and Nathaniel Nystrom (2015). "A Refactoring Library for Scala Compiler Extensions". In: *Compiler Construction*. Ed. by Björn Franke. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 31–48. ISBN: 978-3-662-46663-6. DOI: 10.1007/978-3-662-46663-6_2. URL: https://doi.org/10.1007/978-3-662-46663-6_2.

Willis, Jamie and Nicolas Wu (Sept. 2018). "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, pp. 24–34. ISBN: 978-1-4503-5836-1. DOI: 10.1145/3241653.3241656. URL: https://doi.org/10.1145/3241653.3241656.

— (2022). "Design patterns for parser combinators in Scala". In: *Proceedings of the Scala Symposium*. Scala '22. Berlin, Germany: Association for Computing Machinery, pp. 9–21. ISBN: 9781450394635. DOI: 10.1145/3550198.3550427. URL: https://doi.org/10.1145/3550198.3550427.

Willis, Jamie, Nicolas Wu, and Matthew Pickering (Aug. 2020). "Staged selective parser combinators". In: *Proc. ACM Program. Lang.* 4.ICFP. DOI: 10.1145/3409002. URL: https://doi.org/10.1145/3409002.

Appendix A

Proofs of Parser Laws

The approach taken to prove the following parser laws for `parsley` is via equational reasoning on `gigaparsec` semantics, under the assumption that their semantics are equivalent. While there is no formal proof of this equivalence at the present, `gigaparsec` was designed to have semantics equivalent to `parsley`'s.

A.1 Left absorption for `fmap`

```
    f <$> empty
=     { applicative functor law }
    pure f <*> empty
=     { definition of <*> }
    liftA2 ($) (pure f) empty
=     { semantics of liftA2 }
    Parsec $ λst ok err →
      let ok' x st' = (unParsec empty) st' (ok . (x $)) err
      in (unParsec $ pure f) st ok' err
=     { semantics of empty }
    Parsec $ λst ok err →
      let ok' x st' = (unParsec $ raise (`emptyErr` 0)) st' (ok . (x $)) err
      in (unParsec $ pure f) st ok' err
=     { semantics of raise }
    Parsec $ λst ok err →
      let ok' x st' = (unParsec $ Parsec $ λst'' _ bad →
        useHints bad (emptyErr st'' 0) st') st' (ok . (x $)) err
      in (unParsec $ pure f) st ok' err
=     { β-reduction }
    Parsec $ λst ok err →
      let ok' x st' = useHints err (emptyErr st' 0) st'
      in (unParsec $ pure f) st ok' err
=     { semantics of pure }
    Parsec $ λst ok err →
      let ok' x st' = useHints err (emptyErr st' 0) st'
      in (unParsec $ Parsec $ λst'' ok'' _ → ok'' f st'') st ok' err
=     { β-reduction }
    Parsec $ λst ok err →
      let ok' x st' = useHints err (emptyErr st' 0) st'
```

```
    in ok' f st
=   { inline ok' }
  Parsec $ λst ok err → useHints err (emptyErr st 0) st
=   { rearrange and α-conversion }
  Parsec $ λst _ bad → useHints bad ((`emptyErr` 0) st) st
=   { fold definition of raise }
  raise (`emptyErr` 0)
=   { fold definition of empty }
  empty
```