

parsley-garnish

A linter for the parsley parser combinator library

Rocco Jiang

June 8, 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 8, 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	5
2.2	Writing Linters for Scala	5
2.2.1	Choice of Tooling	6
2.2.2	A Crash Course on Scalafix	7
2.3	Parser Combinators	9
2.3.1	Parsley	11
2.3.2	Design Patterns for Parsley	11
2.3.3	Haskell Parsley	13
3	Lint Rules in parsley-garnish	14
3.1	Avoid Redefining Existing Parsers	14
3.2	Simplify Complex Parsers	14
3.3	Ambiguous Implicit Conversions	14
3.4	Remove Explicit Usage of Implicit Conversions	17
3.5	Refactor to use Parser Bridges	17
3.6	Left Recursion Factoring	17
4	Design and Implementation	18
4.1	Parser Representation	18
4.1.1	Detecting Named Parsers	19
4.1.2	Converting Scalameta Terms to the Parser ADT	19
4.1.3	Building New Parsers From Existing Parsers	20
4.1.4	Simplifying Parsers Using Parser Laws	20
4.1.5	Converting Parsers Back to Scalameta Terms	23
4.2	Function Representation	23
4.2.1	Representing Functions	23
4.2.2	Achieving Normalisation	25
4.2.3	Converting Scalameta Terms to the Function ADT	27
4.2.4	Normalising Function Terms	28
4.2.5	Converting Functions back to Scalameta Terms	30
4.2.6	Further Work?	30
4.3	Limitations	30
5	Evaluation	31
6	Related Work	32
7	Conclusion	33
7.1	Review	33
7.2	Future Work	33

Bibliography	34
A Proofs of Parser Laws	37
A.1 Left absorption for <code>fmap</code>	37

List of Figures

2.1	An example of a warning from the Haskell linter <code>hlint</code> , suggesting a fix that a user can choose to automatically apply.	3
2.2	An example of the <i>Extract Method</i> refactoring in IntelliJ IDEA.	4
2.3	A syntactic rule that makes all <code>val</code> definitions lazy.	8
2.4	Using Scalameta quasiquotes to construct a tree.	9
2.5	An example showing how semantic rules are semantic-aware on overloaded method names. . .	10
2.6	The grammar and AST for our simple expression language.	12
2.7	A parser for our simple expression language.	12
3.1	Example of the <i>Ambiguous Implicit Conversions</i> lint rule in action.	16
4.1	A subset of the core combinators in the <code>Parser</code> ADT.	18
4.2	Functor (4.1), Applicative (4.2, 4.3), and Alternative (4.4–4.8) laws.	21
4.3	The untyped λ -calculus extended with n -ary abstraction and application.	24
4.4	The <code>Function</code> ADT for representing functions.	25
4.5	The β -reduction rule for the n -ary lambda calculus.	25
4.6	Normalisation by evaluation in a semantic model.	27
4.7	Normalisation by evaluation for the <code>Function</code> datatype.	28

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, 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

¹<https://eslint.org/docs/latest/rules/>

²<https://doc.rust-lang.org/clippy/>

³<https://hackage.haskell.org/package/hlint>

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.

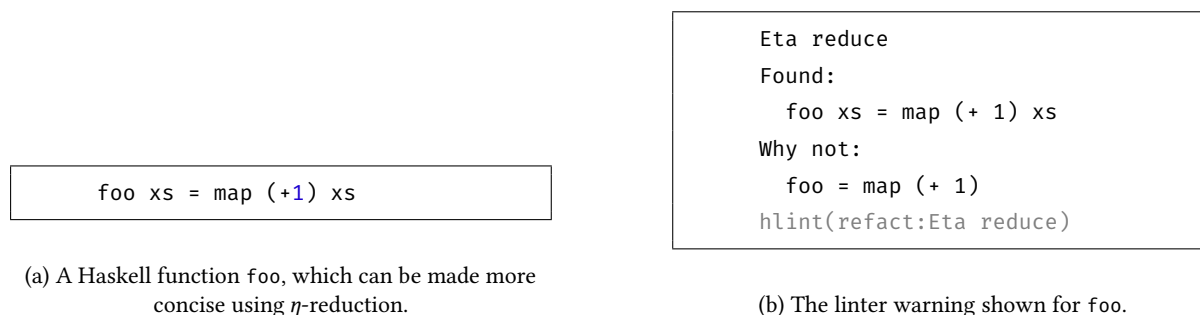


Fig. 2.1: An example of a warning from the Haskell linter `hlint`, suggesting a fix that a user can choose to automatically apply.

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, 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 IDEA⁵ IDE.

⁴<https://github.com/xunit/xunit.analyzers>

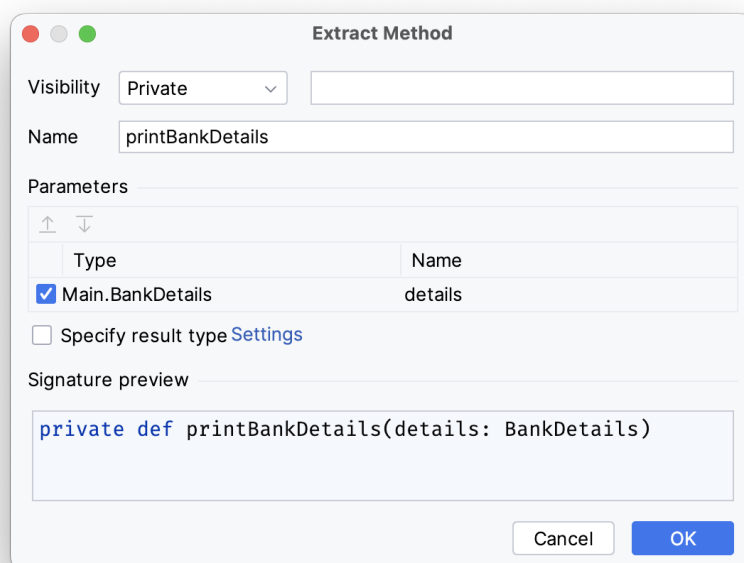
⁵<https://www.jetbrains.com/idea/>


```

1  object Main {
2      def main(args: Array[String]): Unit = {
3          val bankDetails = getBankDetails()
4          println(s"Account name: ${bankDetails.name}")
5          println(s"Account balance: ${bankDetails.balance}")
6      }
7  }

```

(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.

```

1  object Main {
2      def main(args: Array[String]): Unit = {
3          val bankDetails = getBankDetails()
4          printBankDetails(bankDetails)
5      }
6
7      private def printBankDetails(details: BankDetails): Unit = {
8          println(s"Account name: ${details.name}")
9          println(s"Account balance: ${details.balance}")
10     }
11 }

```

(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.

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 run to observe their actual runtime behaviour. Both techniques have their own advantages and disadvantages, and the choice between them depends on the desired properties of a tool. Dynamic analysers are *precise* in that they only observe what a program can actually do, but requires a whole system that can be executed. On the other hand, static analysers can be more lightweight and can be applied to incomplete systems, since they do not require the program to be run. However, the trade-off is that static analysis is generally more *imprecise* since it must approximate the possible behaviours of the program.

Linters should be fast and lightweight, as it is desirable for them to provide feedback to the user in real-time within the IDE. Furthermore, it is desirable for them to be able to provide warnings on incomplete programs and analyse even just fragments of code for suspicious code patterns. For these reasons, the vast majority of linters are static analysis tools.

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 – some linters opt to run during compilation as a compiler plugin, while others run as a separate tool after compilation. Lint-on-compile and lint-after-compile tools both have their own advantages and disadvantages, and the choice between them depends on the use case [Sadowski et al. 2018].

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 dot ty, 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

⁶<https://scalameta.org/>

⁷<https://scalameta.org/metals/>

⁸<https://scalacenter.github.io/scalafix/>

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 needing to invoke 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 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 GitHub⁹, 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 Plugin¹⁰. 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.

WartRemover¹¹ 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.

ScalaStyle¹² 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

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/>

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.
- `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`.
- `${...}` is the *splice* or *unquote* operation to insert a term 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: §4.2 in part discusses how this is handled in `parsley-garnish`.

```

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 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,

```
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.

```
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.

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.

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`.


```
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.

2.3.3 Haskell Parsley

TODO: Should I introduce the idea that haskell parsley uses staged metaprogramming for optimisation purposes here?

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

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.

3.1 Avoid Redefining Existing Parsers

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

3.2 Simplify Complex Parsers

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

3.3 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 conversions¹, 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.

Example

Fig. 3.1a 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 similar to that shown in fig. 3.1b.

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

¹<http://www.wartremover.org/doc/warts.html#implicitconversion>

```
1 object parser {  
2   import parsley.syntax.character.stringLift  
3   import lexer.implicit._  
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 implicit = lexer.lexeme.symbol.implicit  
13 }
```

(a) A minimal Parsley program which fails to compile due to ambiguous implicit in the parser object.

```
warning: [AmbiguousImplicitConversions] This import may cause clashing implicit conversions:  
* import parsley.syntax.character.stringLift at line 2  
* import lexer.implicit._ 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.implicit._  
^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
```

(b) The warning message reported at line 3 by the *Ambiguous Implicit Conversions* lint rule.

Fig. 3.1: Example of the *Ambiguous Implicit Conversions* lint rule in action.

* 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

3.4 Remove Explicit Usage of Implicit Conversions

3.5 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

3.6 Left Recursion Factoring

Chapter 4

Design and Implementation

Writing domain-specific lint rules unlocks the potential for more powerful and interesting transformations utilising specialised domain knowledge. Desirable: * inspectability for analysis (that's what we're here for!) and optimisation The purpose of this chapter is to describe the intermediate representations of parsers (§4.1) and functions (§4.2). Show that terms must be simplified to a normal form Demonstrate equivalence to dsl optimisations in staged metaprogramming

4.1 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)

Scalafix runs at the meta-level, outside of the phase distinction of compile- and run-time. Staged metaprogramming applies optimisations at compile-time, whereas these “optimisations” at applied post-compilation

For example, given two AST nodes `Term.Name("p")` and `Term.Name("q")` corresponding to named parsers `p` and `q`, suppose a transformation involves combining them with the `ap` combinator `<*>`. One may consider using quasiquotes to achieve this: `q"p <*> q"` would automatically expand to `Term.ApplyInfix(Term.Name("p"), Term.Name("<*>"), Type.ArgClause(List(Term.Name("q")), None))`. However, this loses the static inspectability of the individual parsers `p` and `q` – although quasiquotes can be used as extractor patterns to recover the original AST nodes, their usage as such is discouraged as they can easily result in unintended match errors. The recommended approach is to pattern match on the AST nodes directly, which is obviously unergonomic even for this small example: to extract the RHS term `q`, one would have to perform a nested pattern match on the `Term.ApplyInfix` term and its `Term.ArgClause` node representing the arguments of the infix function application.

It is hopefully obvious that this would a very painful process for the rule author. It would be desirable to abstract away from the low-level syntactic AST representation, and instead treat these AST nodes as what they semantically represent – parsers.

Instead, fig. 4.1 shows how parser terms can be represented as an algebraic data type ADT, in the same way parsley itself uses a deep embedding to represent parsers as pure data objects. The reasoning behind this approach is the same as that for PARSLEY – this representation allows parsers to be easily inspected and analysed via pattern matching on constructors.

```
trait Parser
case class NonTerminal(ref: Symbol) extends Parser
case class Pure(f: Function) extends Parser
case object Empty extends Parser
case class Choice(p: Parser, q: Parser) extends Parser
case class Ap(p: Parser, q: Parser) extends Parser
...
```

Fig. 4.1: A subset of the core combinators in the `Parser` ADT.

This section explores the motivation behind this and the design choices made in the implementation. Use the left-recursion factoring (§3.6) rule as a basis/context to demonstrate the utility of this representation.

Running example The left-recursion factoring rule (§3.6) performs the most complex analyses and transformations on parsers in parsley-garnish. Thus, it is a good example to motivate the design requirements for the

parser representation. The following left-recursive parser and its transformation into its postfix form will serve as a running example for this section:

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

4.1.1 Detecting Named Parsers

Before any analysis on parsers can be performed, it is first necessary to identify which AST nodes correspond to parsers. `parsley-garnish` builds a map of all parsers defined within a source file, indexed by the unique symbol of its name.

Identifying these AST nodes of interest involves pattern matching on `val`, `var`, and `def` definitions with a type inferred to be some `Parsley[_]` – this information is accessed by querying the Scalafix semantic API for the node’s symbol information. Consider the labelled ast structure of the `expr` parser:

```
Defn.Val(
  mods = List(Mod.Lazy()),
  pats = List(Pat.Var(Term.Name("expr"))),
  decltpe = Some(
    Type.Apply(Type.Name("Parsley"), Type.ArgClause(List(Type.Name("String"))))
  ),
  rhs = Term.ApplyInfix(...)
)
```

The qualified symbol `expr` is used as the key in the map, and the `rhs` term is lifted the intermediate parser representation for analysis. A reference to the original AST node is also kept so any lint diagnostics or code rewrites can be applied to the correct location in the source file. Thus, a full traversal through the source file builds a map of all named parsers, representing all non-terminals in the grammar defined within that file.

4.1.2 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 `scala.meta.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 `scala.meta.Term`. These `fromTerm` methods are combined to define a `toParser` extension method on `scala.meta.Term` – this is where AST nodes are lifted to their corresponding `Parser` representation.

The top-level combinator that makes up `expr`’s definition is the choice combinator, `|`. Scalameta represents this infix application of the `|` operator as so:

```
Term.ApplyInfix(
  lhs = Term.Apply(...), // AST node for (expr, string("a")).zipped(_ + _)
  op = Term.Name("|"),
  targClause = Type.ArgClause(List()),
  argClause = Term.ArgClause(
    List(
      Term.Apply(
        Term.Name("string"),
        Term.ArgClause(List(Lit.String("b")), None)
      )
    ),
    None
  )
)
```


This structure therefore guides the implementation of the pattern match in `Choice.fromTerm`:

```
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)
  }
}
```

The definition of this method is fairly self-explanatory: it matches on a `ApplyInfix` term where the operator is the `|` combinator, and recursively applies `toParser` to its LHS and RHS nodes. Finishing off, the `expr` parser is therefore converted to the following `Parser` instance:

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

The exact representation of the `Function` is not important at this moment – this is covered in the next section. For brevity, the remaining code snippets in this section will simplify the function representations and continue to grey them out.

4.1.3 Building New Parsers From Existing Parsers

Now that raw AST terms can be lifted to the higher-level parser representation, it is easy to build new parsers from existing parsers. This is crucial for left-recursion factoring, 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.

Smart constructors are used to make manipulating parser terms resemble writing parsley code itself. These are defined as infix operators, which are written as extension methods on the `Parser` trait:

```
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)
}
```

Parser terms can now be manipulated in a manner that looks almost indistinguishable from writing parsley code. For example, the `unfold` method on the `Ap` parser contains this snippet, where `pl`, `ql`, and `q` are parsers (`pe` is not a parser, but rather an `Option` value):

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

Other than the capitalised `Empty` constructor, this would be perfectly valid parsley code.

4.1.4 Simplifying Parsers Using Parser Laws

Recombining unfolded parsers during left-recursion factoring introduces many necessary, but extraneous “glue” combinators. Even though the transformed parser is semantically correct, it ends up very noisy syntactically. Consider the resulting parser from factoring out the left-recursion in `expr`:

```

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(identity).map(compose(a => b => a + b)))
    .map(flip) <*> string("a")
    | empty | empty
)

```

The intent of this parser is completely obfuscated – it would be unacceptable for the output of the transformation to be left in this form. For human readability, this parser term must be simplified as much as possible, using domain-specific knowledge about parser combinators. This is where the deep embedding approach comes to shine; simplifications are easily expressed by pattern matching on `Parsley` constructors.

Willis, Wu, and Pickering [2020] note that parser combinators are subject to *parser laws*, which often form a natural simplification in one direction. In Haskell `parsley`, Willis [2023] uses these parser laws as the basis for high-level optimisations to simplify the structure of the combinator tree. `parsley-garnish` uses the same principles to simplify the parser term to become more human-readable. The two only differ in the purpose of the simplification: whereas Haskell `parsley` does this to produce an optimised AST to be compiled as code, `parsley-garnish` simplifies the parser AST to be pretty-printed as text.

Fig. 4.2 shows the subset of parser laws utilised by `parsley-garnish` for parser simplification. Most of the laws in fig. 4.2 have already been shown to hold for `Parsley` by Willis and Wu [2018]; an additional proof for eq. (4.8) can be found in appendix A.

$$p.map(f).map(g) = p.map(g \circ f) \quad (4.1)$$

$$pure(f) <*> pure(x) = pure(f(x)) \quad (4.2)$$

$$pure(f) <*> x = x.map(f) \quad (4.3)$$

$$empty \mid u = u \quad (4.4)$$

$$u \mid empty = u \quad (4.5)$$

$$pure(x) \mid u = pure(x) \quad (4.6)$$

$$empty <*> u = empty \quad (4.7)$$

$$empty.map(f) = empty \quad (4.8)$$

Fig. 4.2: Functor (4.1), Applicative (4.2, 4.3), and Alternative (4.4–4.8) laws.

In the previous example, it is evident that the most noise results from the empty combinators. These can be eliminated using eqs. (4.4), (4.5), (4.7), and (4.8):

```

lazy val expr: Parsley[String] = chain.postfix(string("b"))(
  (pure(identity).map(compose(a => b => a + b))).map(flip) <*> string("a")
)

```

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

```

(pure(identity).map(compose(a => b => a + b))).map(flip) <*> string("a")
= { eqs. (4.2) and (4.3) }
pure(compose(a => b => a + b)(identity)).map(flip) <*> string("a")
= { eqs. (4.2) and (4.3) }
pure(flip(compose(a => b => a + b)(identity))) <*> string("a")

```

```
= { eq. (4.3) }
    string("a").map(flip(compose(a => b => a + b)(identity)))
```

This results in the most simplified form of the parser:

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

Encapsulating boilerplate Lawful simplifications are applied akin to peephole optimisations on the recursively defined `Parser` ADT. There are many instances of parsers, which inevitably leads to repetitive and error-prone boilerplate code which exists to simply recurse through each case. To avoid this, the recursive traversal itself is decoupled from the application of the transformation function. Although the traversal is still hand-written, the implementation is inspired by the generic traversal patterns offered by Haskell’s `uniplate` library [Mitchell and Runciman 2007].

This is realised as a `transform` method on the `Parser` trait, which takes a partial function and applies it to nodes where it is defined. The transformation is applied via a bottom-up traversal:

```
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
}
```

A rewrite method can then be defined in terms of `transform`, applying the partial function everywhere and re-applying it until it no longer makes a change. This has the effect of applying a transformation exhaustively until a normal form is reached.

```
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 recursion boilerplate: the act of traversal itself is fully abstracted away and encapsulated within the `transform` method. 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)
  ...
}
```

Further design considerations are made to ensure the extensibility and safety of this approach: the `Parser` trait is sealed, which enables compiler warnings if a new `Parser` case is added and the `transform` method is not updated. Since the traversal is still written by hand rather than generically derived, it is still more prone to error. The

traversal could be generically derived rather than written by hand, but this would require the use of an external dependency such as `shapeless`¹, which is overkill for the complexity of the `Parser` ADT.

4.1.5 Converting Parsers Back to Scalameta Terms

After parsers have been transformed and simplified, the last step is to convert them back to a textual representation to be applied as a Scalafix patch. Parsers can be lowered back to `scala.meta.Term` nodes by the inverse of the original `fromTerm` transformation. The `Parser` trait defines this transformation as the method `term`, using quasiquotes to simplify the construction of the `scala.meta.Term` nodes.

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

This term can then be pretty-printed into a string, and applied as a Scalafix patch.

Summary

4.2 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

Section 4.1 showed that it is useful to lift AST nodes corresponding to parser terms into an intermediate `Parser` representation. Crucially, §4.1.4 demonstrates how this representation lends itself to reducing parsers to a simpler form. This section extends this idea to representing functions in a similar, statically-inspectable manner, and explores why this machinery is not just desirable but necessary for `parsley-garnish`.

4.2.1 Representing Functions

Utilising parser laws as a basis for term simplification for the `Parser` ADT has managed to tame the jumble of parser combinators resulting from the left-recurring factoring transformation §3.6. However, the functions passed to lifting combinators such as `map` have so far been treated as opaque, and have not been subject to the same simplifications as parsers.

Continuing from the running example from 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)(identity))
```

This is because 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. These are used to ensure that unfolded parsers are recombined in a lawful manner, but results in a lot of syntactic noise. Yet again, leaving the transformed terms in this form would be unacceptable for the linter. It is vital that these functions are partially evaluated and *normalised* into a semantically equivalent but visually simpler form. To achieve this, a representation of functions is needed that is amenable to the same kind of static inspection as the `Parser` ADT.

Defunctionalisation

In the same way that representing parsers as datatypes allows them to be statically inspectable, the aforementioned higher-order functions can also be rendered as datatypes to allow them to be analysed in a similar manner. This

¹<https://github.com/milessabin/shapeless>

is an example of *defunctionalisation*, a technique to eliminate higher-order functions by transforming them into specialised first-order equivalents [Reynolds 1972; Danvy and Nielsen 2001].

Ideally for type safety, this could be represented as a generalised algebraic data type (GADT) [Cheney and Hinze 2003] with a type parameter documenting the type that the function term represents:

```
trait Defunc[+T]
case class Identity[A]() extends Defunc[A => A]
case class Flip[A, B, C]() extends Defunc[(A => B => C) => B => A => C]
case class Compose[A, B, C]() extends Defunc[(B => C) => (A => B) => A => C]
```

Facilitates more high-level parser simplifications

Lambda Calculus

The defunctionalised constructs alone are not enough to fully represent the functions passed to combinators. There needs to be a way to apply them to arguments, and to represent the arguments themselves – the core language to do so is the lambda calculus [Church 1936]. Furthermore, once the defunctionalised functions have exhausted their usefulness in high-level parser simplifications, they also need to be lowered to λ -terms to be normalised. In the λ -calculus, each function only takes one argument, and multi-argument functions are represented as a chain of single-argument functions: this is known as *currying*. In Scala, functions are allowed to have multiple parameter lists to make them curried, although uncurried functions are preferred for performance reasons. Since these functions will be transformed from Scala code and back, it is desirable to maintain a high-level equivalence between these two representations. Thus, fig. 4.3 shows the λ -calculus extended to represent multi-argument functions using n -ary abstraction and application.

The Function ADT

The syntactic representation of functions is thus represented as the uni-typed `Function` ADT in fig. 4.4. The `Lambda` trait is a subset of the entire `Function` trait, representing λ -terms, with the addition of a `Translucent` term to encapsulate open terms holding a `scala.meta.Term` which cannot be evaluated further.

The original intention was to represent `Function` as a type-parameterised ADT for improved type safety, where `Lambda` would use a variant of the simply typed λ -calculus. This would've also allowed `Parser` to be parameterised by the result type of the parser. However, attempting to implement this ran into two main hurdles:

- `Var` and `Translucent` terms would need to be created with concrete type parameters of their inferred types. Scalafix's semantic API is not powerful enough to guarantee that all terms can be queried for their inferred types – in fact, the built-in Scalafix rule *Explicit Result Types* calls the Scala 2 presentation compiler to extract information like this². This solution is not ideal as it is brittle and breaks Scalafix's cross-compatibility promises.
- Scala 2's type inference for GADTs is less than ideal, requiring extra type annotations and unsafe casts which ultimately defeat the original purpose of type safety. This situation is improved, although not completely solved, in Dotty [Parreaux, Boruch-Gruszecki, and Giarrusso 2019] – but Scalafix does not yet support Scala 3.

²<https://github.com/scalacenter/scalafix/issues/1583>

$M, N ::= x$	(variable)
$\mid \lambda(x_1, \dots, x_n).M$	(n -ary abstraction)
$\mid M(N_1, \dots, N_n)$	(n -ary application)

Fig. 4.3: The untyped λ -calculus extended with n -ary abstraction and application.

```

trait Function

// Core expression language
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

// Defunctionalised higher-order functions
trait Defunc extends Function
case object Identity extends Defunc
case object Flip extends Defunc
case object Compose extends Defunc
...

```

Fig. 4.4: The `Function` ADT for representing functions.

4.2.2 Achieving Normalisation

Partial evaluation and normalisation are related concepts: it is useful to view normalisation as statically evaluating as many terms as possible, but since not all terms have known values, the expression cannot be fully evaluated to a result value. Normalisation can thus be viewed simply as a process of evaluation, but in the presence of unknown terms. This section first explains the traditional notion of normalisation by rewriting, before introducing normalisation by evaluation as a more elegant and efficient alternative.

Reduction-Based Normalisation

In the λ -calculus, terms are reduced in size by applying β -reduction: fig. 4.5 shows how this can be defined for the n -ary λ -calculus. Unlike the standard λ -calculus, reduction will only take place if the expected number of arguments in $\mathbf{x} = x_1, \dots, x_n$ are equal to the number of arguments in $\mathbf{N} = N_1, \dots, N_n$; otherwise, the term is left as is. This is therefore a *directed* notion of reduction, which can be implemented as a term-rewriting system, in a similar way to how `Parser` terms are simplified.

The goal is to achieve beta normal form (β -NF) by allowing β -reduction to occur deep inside λ -terms, in all redexes of a term, until no more reductions can be made. Normally in the untyped λ -calculus, β -reduction as a rewriting rule does not manage to be strongly normalising as desired, and may lead to non-terminating rewrites. However, despite the `Lambda` representation itself being untyped, all `Lambda` terms are converted from valid typed Scala expressions. Scala 3 is modelled on the theoretic foundations of the Dependent Object Types (DOT) calculus, which has been shown to be strongly normalisable [Wang and Rompf 2017]. Although not proven for Scala 2, it is reasonable to assume that the same result holds, at least for the subset of expressions that are represented in the `Lambda` representation.

$$(\lambda \mathbf{x}.M)\mathbf{N} \rightarrow_{\beta} M[\mathbf{N}/\mathbf{x}] \quad (\text{if } |\mathbf{x}| = |\mathbf{N}|)$$

Fig. 4.5: The β -reduction rule for the n -ary lambda calculus.

In fig. 4.5, the syntax $M[\mathbf{N}/\mathbf{x}]$ denotes term substitution, where all free occurrences of x in M are replaced with N . Substitution must avoid *variable capture*, when N contains free variables that are bound in the scope where x is found [van Bakel 2022]. Capture-avoiding substitution is a notoriously tricky and subtle problem. There are a plethora of different approaches to solve this, mostly boiling down to how variable names get represented:

Naïve substitution Representing variable names as strings is the most straightforward approach in terms of understandability. Naïvely substituting these terms, however, seems logically simple but can be very tricky to get right. This approach requires calculating the free variables in a scope before performing substitution, renaming bound variables if it would lead to variable capture. This is not used in any real implementation of the λ -calculus due to its inefficiency, as it requires traversing the term tree multiple times.

Barendregt’s convention Renaming all bound variables to be unique satisfies **Barendregt’s convention** [Barendregt 1984], which removes the need to check for variable capture during substitution. However, variables still have to be renamed during substitution – this administrative renaming has relatively high performance overhead and chews through a scarily large number of fresh variable names. This approach can be massively optimised though: the Haskell GHC compiler uses a technique dubbed “the Rapier” [Peyton Jones and Marlow 2002], maintaining invariants to avoid renaming on substitution when unnecessary. Unfortunately, maintaining the invariants to keep this transformation correct becomes very difficult [Maclaurin, Radul, and Paszke 2023].

Nameless and hybrid representations Nameless representations like *De Bruijn indices* [de Bruijn 1972] eschew names entirely, instead representing variables as the number of binders between the variable and its binding site. Although an elegant representation, De Bruijn terms are notoriously difficult to work with, as they are not easily human-readable. Furthermore, performing substitutions with De Bruijn terms has an overhead as variable positions have to be shifted. To avoid this, hybrid representations combining named and nameless representations exist [McBride and McKinna 2004; Charguéraud 2012], but they become rather complex solutions for what should be a relatively simple λ -calculus implementation for parsley-garnish’s needs.

Higher-order abstract syntax *Higher-order abstract syntax* (HOAS) [Pfenning and Elliott 1988] sidesteps variable binders entirely by borrowing binding and substitution from the meta-language.

```
trait FOAS
case class Abs[A, B](x: Var[A], f: FOAS[B]) extends FOAS[A => B]
case class App[A, B](f: FOAS[A => B], x: FOAS[A]) extends FOAS[B]
case class Var[A](name: VarName) extends FOAS[A]

// x. y. x y
val expr = Abs(Var("x"), Abs(Var("y"), App(Var("x"), Var("y"))))

trait HOAS
case class Abs[A, B](f: HOAS[A] => HOAS[B]) extends HOAS[A => B]
case class App[A, B](f: HOAS[A => B], x: HOAS[A]) extends HOAS[B]
case class Var[A](name: VarName) extends HOAS[A]

// x. y. x y
val expr = Abs(x => Abs(y => App(x, y)))
```

is a representation that uses the meta-language’s own functions to represent the object language’s terms. **we will use this but also with nbe as shown in the next subsection** syntax - named, semantics - hoas, normalising by evaluation = automatically alpha converts terms by renaming

Normalisation by Evaluation

An interesting alternative approach stems from a notion of *reduction-free* normalisation, based on an undirected notion of term equivalence, rather than directed reduction. *Normalisation by Evaluation* (NBE) evaluates syntactical terms into a semantic model, which is then *reified* back into the syntactic model [Filinski and Korsholm Rohde 2004]. Normalisation is then just defined as the composition of these two operations, as illustrated in fig. 4.6.

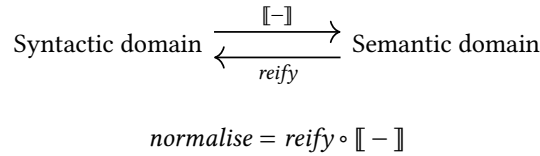


Fig. 4.6: Normalisation by evaluation in a semantic model.

The denotational model (denoted by $\llbracket - \rrbracket$) is specifically constructed to be *residualising*, meaning that terms can be extracted out into the original syntactic representation.

Represent semantic model with HOAS

```

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

```

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.

Evaluating Performance of Normalisation Strategies

4.2.3 Converting Scalameta Terms to the Function ADT

There are three cases to consider when converting `scala.meta.Term` nodes into `Function` terms.

Lambda Expressions Writing parsers often involves defining simple lambda expressions used to glue together parsers, or to transform the result of a parser:

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

These lambda expressions are represented in the Scalameta AST as `Term.Function` nodes, which are recursively traversed to collect all parameter lists. This is folded into a chain of n -ary abstractions, with the final term being the body of the lambda, which is wrapped into a `Translucent` term. To ensure that the parameter names in the `Translucent` body term are unique, the parameters are α -converted to fresh names. Take the following example to see why this is helpful:

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

Although any sane Scala programmer would not write this, this convoluted example shows how variable shadowing can occur – 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. The resulting `Function` term would then resemble the following λ -calculus expression:

```
 $\lambda(\_l1). \lambda(\_l2, \_l3). \text{Translucent}(\_l2 + \_l3, \text{env} = \{\_l1 \rightarrow \_l1, \_l2 \rightarrow \_l2, \_l3 \rightarrow \_l3\})$ 
```

Values shown in bold are `scala.meta.Term` nodes, so the lambda body's environment maps `Term.Name` nodes to their corresponding variable terms. When the term is pretty-printed, the `Term.Name` nodes are replaced with the corresponding `Function` terms – this is similar to the splicing operation on quasiquotes.

Placeholder Syntax Scala supports a placeholder syntax using underscores to make lambda expressions more concise, so the earlier parser can be rewritten as:

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


Scalameta differentiates between regular lambda expressions and those using placeholder syntax, representing the latter as `Term.AnonymousFunction` nodes. This makes it easy to identify which approach to be taken during conversion. To convert this case, each successive underscore in the expression body is replaced with a fresh variable name. Using placeholder syntax only results in a fully uncurried function with a single parameter list³ – this is filled with the freshly generated variable names in order of their occurrence in the expression body.

Eta-Expansion If the term is not a lambda expression, parsley-garnish attempts to η -expand the term if possible. For example, an idiomatic parser written using the *Parser Bridges* pattern [Willis and Wu 2022] could resemble the following:

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

When parsley-garnish converts `asciiCode` to a `Parser`, it desugars the bridge constructor into something resembling `item.map(AsciiCode.apply)`. The η -expanded form of `AsciiCode.apply` would be as follows:

```
(char: Char) => AsciiCode.apply(char)
```

To η -expand `scala.meta.Term` nodes, parsley-garnish attempts to look up the method signature of its symbol using Scalafix’s semantic API. This is not always possible – in that case, the term can’t be statically inspected any further and is just wrapped in a `Translucent` term.

4.2.4 Normalising Function Terms

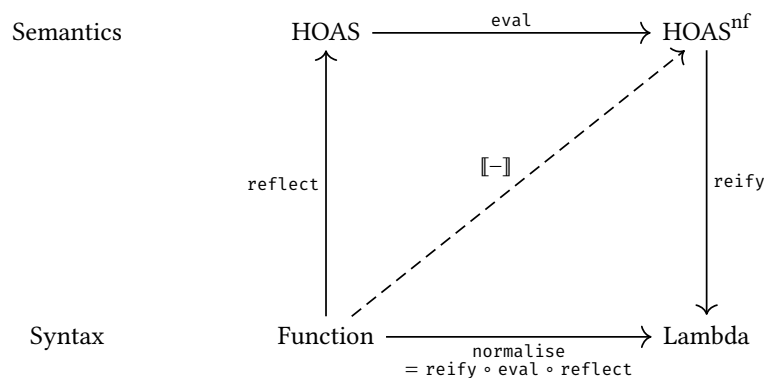


Fig. 4.7: Normalisation by evaluation for the `Function` datatype.

the code below is outdated

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) =>
```

³<https://www.scala-lang.org/files/archive/spec/2.13/06-expressions.html#anonymous-functions>

```

    HApp(reflect0(f, boundVars), xs.map(reflect0(_, boundVars)))
  case v @ Var(name) =>
    boundVars.getOrElse(v, HVar(name))
  case Translucent(term, env) =>
    HTranslucent(term, env.mapValues(reflect0(_, 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 HTranslucent(t, env) => HTranslucent(t, env.mapValues(_.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.mapValues(_.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.mapValues(_.reify))
  }
}

```

4.2.5 Converting Functions back to Scalameta Terms

Surprise bitches same shit again, quasiquotes ftw

4.2.6 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. idk, 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

4.3 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 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 Boogie 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

The left-recursion factoring transformation could be implemented in a staged version of `parsley` to eliminate left-recursion at compile-time with no run-time overhead. However, from a design perspective this is potentially controversial as it obfuscates the `PEG` semantics of the parser combinator library, allowing `CFG`-like parsers to be written directly. One could also argue (pedagogical perspective? not really) that the higher-level abstraction of the `chain` combinators is a more desirable form to write parsers for left-associative operations, rather than a lower-level left-recursive grammar rule.

7.2 Future Work

Bibliography

- Barendregt, Henk P. (1984). “Chapter 2 - Conversion”. In: *The Lambda Calculus*. Vol. 103. Studies in Logic and the Foundations of Mathematics. ISSN: 0049-237X. Elsevier, pp. 22–49. DOI: <https://doi.org/10.1016/B978-0-444-87508-2.50010-1>. URL: <https://www.sciencedirect.com/science/article/pii/B9780444875082500101>.
- Burmako, Eugene (Mar. 2017). “Unification of Compile-Time and Runtime Metaprogramming in Scala”. Ph.D. Thesis. Lausanne: EPFL. DOI: [10.5075/epfl-thesis-7159](https://doi.org/10.5075/epfl-thesis-7159). URL: <https://doi.org/10.5075/epfl-thesis-7159>.
- Charguéraud, Arthur (Oct. 2012). “The Locally Nameless Representation”. In: *Journal of Automated Reasoning* 49.3, pp. 363–408. ISSN: 1573-0670. DOI: [10.1007/s10817-011-9225-2](https://doi.org/10.1007/s10817-011-9225-2). URL: <https://doi.org/10.1007/s10817-011-9225-2>.
- Cheney, James and Ralf Hinze (July 2003). *First-class phantom types*. Tech. rep. Cornell University. URL: <https://hdl.handle.net/1813/5614>.
- Church, Alonzo (1936). “An Unsolvable Problem of Elementary Number Theory”. In: *American Journal of Mathematics* 58.2, pp. 345–363. ISSN: 00029327, 10806377. DOI: [10.2307/2371045](https://doi.org/10.2307/2371045). URL: <https://doi.org/10.2307/2371045> (visited on 06/06/2024).
- Danvy, Olivier and Lasse R. Nielsen (2001). “Defunctionalization at work”. In: *Proceedings of the 3rd ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming*. PPDP ’01. Florence, Italy: Association for Computing Machinery, pp. 162–174. ISBN: 158113388X. DOI: [10.1145/773184.773202](https://doi.org/10.1145/773184.773202). URL: <https://doi.org/10.1145/773184.773202>.
- de Bruijn, N.G (1972). “Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem”. In: *Indagationes Mathematicae (Proceedings)* 75.5, pp. 381–392. ISSN: 1385-7258. DOI: [https://doi.org/10.1016/1385-7258\(72\)90034-0](https://doi.org/10.1016/1385-7258(72)90034-0). URL: <https://www.sciencedirect.com/science/article/pii/1385725872900340>.
- Filinski, Andrzej and Henning Korsholm Rohde (2004). “A Denotational Account of Untyped Normalization by Evaluation”. In: *Foundations of Software Science and Computation Structures*. Ed. by Igor Walukiewicz. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 167–181. ISBN: 978-3-540-24727-2. DOI: [10.1007/978-3-540-24727-2_13](https://doi.org/10.1007/978-3-540-24727-2_13). URL: https://doi.org/10.1007/978-3-540-24727-2_13.
- 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).
- Gregor, Douglas and Sibylle Schupp (2006). “STLint: lifting static checking from languages to libraries”. In: *Software: Practice and Experience* 36.3, pp. 225–254. DOI: [10.1002/spe.683](https://doi.org/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](https://doi.org/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](https://doi.org/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](https://doi.org/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](https://doi.org/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](https://doi.org/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](https://doi.org/10.48550/arXiv.2310.11738). URL: <https://doi.org/10.48550/arXiv.2310.11738> (visited on 01/25/2024).
- Maclaurin, Dougal, Alexey Radul, and Adam Paszke (2023). “The Foil: Capture-Avoiding Substitution With No Sharp Edges”. In: *Proceedings of the 34th Symposium on Implementation and Application of Functional Languages*. IFL ’22. <conf-loc>, <city>Copenhagen</city>, <country>Denmark</country>, </conf-loc>. Association for Computing Machinery. ISBN: 9781450398312. DOI: [10.1145/3587216.3587224](https://doi.org/10.1145/3587216.3587224). URL: <https://doi.org/10.1145/3587216.3587224>.
- McBride, Conor and James McKinna (2004). “Functional pearl: i am not a number–i am a free variable”. In: *Proceedings of the 2004 ACM SIGPLAN Workshop on Haskell*. Haskell ’04. Snowbird, Utah, USA: Association for Computing Machinery, pp. 1–9. ISBN: 1581138504. DOI: [10.1145/1017472.1017477](https://doi.org/10.1145/1017472.1017477). URL: <https://doi.org/10.1145/1017472.1017477>.
- 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](https://doi.org/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.
- Parreaux, Lionel, Aleksander Boruch-Gruszecki, and Paolo G. Giarrusso (2019). “Towards improved GADT reasoning in Scala”. In: *Proceedings of the Tenth ACM SIGPLAN Symposium on Scala*. Scala ’19. London, United Kingdom: Association for Computing Machinery, pp. 12–16. ISBN: 9781450368247. DOI: [10.1145/3337932.3338813](https://doi.org/10.1145/3337932.3338813). URL: <https://doi.org/10.1145/3337932.3338813>.
- Peyton Jones, Simon and Simon Marlow (July 2002). “Secrets of the Glasgow Haskell Compiler inliner”. In: *J. Funct. Program.* 12.5, pp. 393–434. ISSN: 0956-7968. DOI: [10.1017/S0956796802004331](https://doi.org/10.1017/S0956796802004331). URL: <https://doi.org/10.1017/S0956796802004331>.
- Pfenning, F. and C. Elliott (June 1988). “Higher-order abstract syntax”. In: *SIGPLAN Not.* 23.7, pp. 199–208. ISSN: 0362-1340. DOI: [10.1145/960116.54010](https://doi.org/10.1145/960116.54010). URL: <https://doi.org/10.1145/960116.54010>.
- 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](https://doi.org/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](https://doi.org/10.1007/978-3-642-13953-6_12). URL: https://doi.org/10.1007/978-3-642-13953-6_12.
- Reynolds, John C. (1972). “Definitional interpreters for higher-order programming languages”. In: *Proceedings of the ACM Annual Conference - Volume 2*. ACM ’72. Boston, Massachusetts, USA: Association for Computing

- Machinery, pp. 717–740. ISBN: 9781450374927. DOI: [10.1145/800194.805852](https://doi.org/10.1145/800194.805852). URL: <https://doi.org/10.1145/800194.805852>.
- Sadowski, Caitlin et al. (Mar. 2018). “Lessons from building static analysis tools at Google”. In: *Commun. ACM* 61.4, pp. 58–66. ISSN: 0001-0782. DOI: [10.1145/3188720](https://doi.org/10.1145/3188720). URL: <https://doi.org/10.1145/3188720>.
- 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](https://doi.org/10.1007/978-3-662-46663-6_2). URL: https://doi.org/10.1007/978-3-662-46663-6_2.
- van Bakel, Steffen (2022). *Lecture Notes on Type Systems for Programming Languages*. URL: <https://www.doc.ic.ac.uk/~svb/TSfPL/>.
- Wang, Fei and Tiark Rompf (2017). “Towards Strong Normalization for Dependent Object Types (DOT)”. In: *31st European Conference on Object-Oriented Programming (ECOOP 2017)*. Ed. by Peter Müller. Vol. 74. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 27:1–27:25. ISBN: 978-3-95977-035-4. DOI: [10.4230/LIPIcs.ECOOP.2017.27](https://doi.org/10.4230/LIPIcs.ECOOP.2017.27). URL: <https://doi.org/10.4230/LIPIcs.ECOOP.2017.27>.
- Willis, Jamie (Aug. 2023). “Parsley: Optimising and Improving Parser Combinators”. Ph.D. Thesis. London: Imperial College London. DOI: [10.25560/110313](https://doi.org/10.25560/110313). URL: <https://doi.org/10.25560/110313>.
- 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](https://doi.org/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](https://doi.org/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](https://doi.org/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 $ \st ok err →
  let ok' x st' = (unParsec empty) st' (ok . (x $)) err
  in (unParsec $ pure f) st ok' err
=   { semantics of empty }
Parsec $ \st 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 $ \st ok err →
  let ok' x st' = (unParsec $ Parsec $ \st'' _ bad →
    useHints bad (emptyErr st'' 0) st') st' (ok . (x $)) err
  in (unParsec $ pure f) st ok' err
=   {  $\beta$ -reduction }
Parsec $ \st ok err →
  let ok' x st' = useHints err (emptyErr st' 0) st'
  in (unParsec $ pure f) st ok' err
=   { semantics of pure }
Parsec $ \st ok err →
  let ok' x st' = useHints err (emptyErr st' 0) st'
  in (unParsec $ Parsec $ \st'' ok'' _ → ok'' f st'') st ok' err
=   {  $\beta$ -reduction }
Parsec $ \st ok err →
  let ok' x st' = useHints err (emptyErr st' 0) st'
  in ok' f st
=   { inline ok' }
Parsec $ \st ok err → useHints err (emptyErr st 0) st
=   { rearrange and  $\alpha$ -conversion }
Parsec $ \st _ bad → useHints bad ((`emptyErr` 0) st) st
=   { fold definition of raise }
raise (`emptyErr` 0)
=   { fold definition of empty }
empty

```