Chapter 1
Related Work

The work in this project draws inspiration from a variety of areas, which are briefly summarised in this chapter.

1.1 Library-Specific Linters

Renggli et al. [2010] acknowledge that generic language-level linters and static analysers fail to capture best practices and common errors particular to specialised domains. This is a sentiment shared by Gregor and Schupp [2006], who discuss how statically checking library semantics is very different from checking language semantics, arguing that this necessitates new program representation at the appropriate level of abstraction. The work in this project corroborates these claims, as the work in **??** demonstrates that the generic Scala AST is not at a sufficient level of abstraction to perform complex parser transformations. Gregor and Schupp [2006] also use a similar terminology to `parsley-garnish` of "lifting" their static checker from language to library level.

The situation that Renggli et al. [2010] face is similar to the one in this project, as they work with an embedded DSL, but in Smalltalk. For their static analyser, the authors also have access to a Scalafix-like tool which reuses the existing `ast` of the host language. However, they opt to define their own separate DSL to specify lint and rewrite rules. In contrast, this project argues that the most ergonomic approach is to model the linter AST on the original AST of the DSL itself – this allows `parsley-garnish` to manipulate ASTs of parsers in a high-level declarative manner comparable to writing standard `parsley` code.

1.2 Left-Recursion Removal

The need to remove left-recursion from parsers is not a new problem, since many popular parser-writing tools such as ANTLR [Parr 2013] produce recursive-descent parsers. Classical left-factoring algorithms such as Paull's algorithm [Moore 2000] and the left-corner transform [Rosenkrantz and Lewis 1970] have been around for decades. The idea of using the `chain` family to idiomatically write left-associative parsers using parser combinators is also rather established [Fokker 1995], and the left-recursion transformation algorithm in `parsley-garnish` is itself based on work from 2004 [Baars and Swierstra 2004]. However, the context in which this transformation is applied is relatively novel, as it is implemented as a code rewrite rule for a linter. Since the output of the transformation is user-facing, rather than generated code, it must be readable and understandable to the end user – this is one of the key challenges faced in this project.

1.3 Metaprogramming and Domain-Specific Languages

The work in **??** draws many parallels to techniques in the field of metaprogramming, especially in its intersection with DSLs. Squid [Parreaux, Shaikhha, and Koch 2017b; Parreaux, Shaikhha, and Koch 2017a; Parreaux, Voizard, et al. 2017] implements quasiquotes for Scala 2.11 and 2.12 that, unlike Scalameta's quasiquotes, are type-safe and hygienic. These quasiquotes also have rudimentary function inlining features, similar to the β-reduction and normalisation techniques used in this project. Squid quasiquotes were used to perform multi-staged rewrite-based optimisations on a linear-algebra DSL [Shaikhha and Parreaux 2019], generating optimised code using the same flavour of approach as `parsley` Haskell [Willis, Wu, and Pickering 2020; Willis 2024]. This DSL, similar to parser combinators, is highly based on algebraic laws, so there are many similarities in the optimisation techniques used in both projects. By the observation in **??**, therefore, this work is dual to the simplification work achieved in `parsley-garnish` – it is possible to adapt the work used by the authors to develop a linter for their DSL.

Scala 3's macro system [Stucki, Brachthäuser, and Odersky 2021] has similar type-safety and hygiene guarantees as Squid, and supports β-reduction on staged `Expr` trees. These techniques supersede the work in **??**, which was necessitated only because Scalameta trees do not have the same functionalities. Squid and Scala 3 macros are superior to the expression representation in this project in the sense that they formalise their type-safety and hygienic properties.

Finally, as already noted, the implementation of the `Expr` AST also shares similarities with the `Defunc` and `Lam` infrastructures for code optimisation in `parsley` Haskell.

Chapter 2
Conclusion

This project has presented a number of lint rules for the `parsley` parser combinator library:

- *Ambiguous Implicit Conversions* (??) alerts the user when multiple conflicting implicit conversions are in scope for their parsers, providing further context why the Scala compiler rejects their program.
- *No Explicit Implicit Conversions* (??) automatically removes unnecessary explicit calls to the implicit conversion methods in `parsley`, which is a code smell.
- *Factor Left-Recursion* (?? and ??) rewrites unproductive left-recursive parsers into an idiomatic form that `parsley` supports. In cases where this cannot be automatically fixed, the rule warns the user that their parser will cause an infinite loop at runtime.
- *Simplify Parsers* (??) suggests and applies simplifications to parsers based on the rules of parser laws.
- *Avoid Parser Redefinitions* (??) suggests and applies rewrites to parsers that accidentally redefine existing higher-level combinators in the `parsley` API.

Implementing these rules necessitated the development of intermediate AST representations to improve the static inspectability of parser and expression terms. The `Parser` AST (?? and ??) models the `parsley` DSL, allowing *code* of parsers to be manipulated in a high-level manner, as if they were parsers themselves. It provides a high-level interface for writing syntax-directed transformations on parser terms, and most importantly handles the burden of simplifying parser terms into a readable form, based on similar optimisation techniques used in `parsley` itself. The `Expr` AST (?? models an extension of the λ-calculus, allowing static representation of expressions to be partially evaluated and normalised to a simpler form.

2.1 Future Work

Separate into practical (improvements to parsley-garnish) and theoretical (cool research things)?

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.

2.1.1 Expression AST

Eta reduction – this is more complicated than in Haskell since Scala has special syntax Proper 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)

2.1.2 Resugaring

https://dl.acm.org/doi/10.1145/2858949.2784755