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 1

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.

1.1 Avoid Redefining Existing Parsers

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

1.2 Simplify Complex Parsers

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

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

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

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

Solution

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

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

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

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

Example

Fig. 1.1 extends the previous example to a full Scala source file following the *Implicit Lexer* pattern, but where the user has erroneously additionally imported the `stringLift` implicit from the *Implicit Conversions* pattern. This results in the Scala compiler throwing an error on line 6 due to ambiguous implicits. When run on this file, `parsley-garnish` will report a warning at line 3 similar to that shown in fig. 1.2.

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

Fig. 1.1: A minimal Parsley program which fails to compile due to ambiguous implicits in the `parser` object.

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

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

Fig. 1.2: The warning message produced by the `AmbiguousImplicitConversions` lint rule.

Implementation

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

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

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

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

1.4 Remove Explicit Usage of Implicit Conversions

1.5 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

1.6 Left Recursion Factoring

Chapter 2
Design and Implementation

2.1 Parser Representation

Several of the more complex lint rules, most notably §1.6, require manipulating parser combinators in a high-level manner. Scalafix works with code represented as data, in the form of a AST-like representation i.e. Scalameta `Tree` nodes. It would be possible, but extremely cumbersome, to work directly with these AST nodes.

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

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

This section explores the motivation behind this and the design choices made in the implementation. Use the §1.6 rule as a basis/context to demonstrate the utility of this representation.

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

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

2.1.1 Detecting Top-Level Parsers

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

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

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

The right hand side of the definition is a Scalameta `Term` node representing the definition of the parser. During traversal, the right hand side of the definition is converted to a `Parser` instance. This is then accumulated in a map,

with the symbol of the definition as the key. A full traversal through a source file obtains all the non-terminals in the grammar defined within that file.

2.1.2 Converting Scalameta Terms to the Parser ADT

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

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

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

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

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

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

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

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

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

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

2.1.3 Building New Parsers From Existing Parsers

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

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

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

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

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

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

2.1.4 Simplifying Parsers Using Parser Laws

Once all the unfolded parsers have been recombined, the raw output is very noisy and difficult to read. Again, ignore the functions, these will be covered in §2.2.

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

This is obviously unacceptable and completely obfuscates the intent of the parser. For human readability of the transformed output, it is therefore important to simplify the parser as much as possible. Now that the parsers are represented as objects, it is easy to pattern match on their constructors. This improved static inspectability allows us to perform simplifications using the laws that govern parser combinators – these often form a natural simplification in one direction. This is similar to the high-level optimisations performed in the Parsley backend as described by Willis, Wu, and Pickering [2020], using the same parser laws. Fig. 2.1 shows the subset of parser laws utilised by `parsley-garnish` for parser simplification.

```
p.map(f).map(g) = p.map(g compose f)                    (2.1)
  pure(f) <*> pure(x) = pure(f(x))                      (2.2)
      pure(f) <*> x = x.map(f)                          (2.3)
           empty | u = u                                (2.4)
           u | empty = u                                (2.5)
        pure(x) | u = pure(x)                           (2.6)
         empty <*> u = empty                            (2.7)
        empty.map(f) = empty                            (2.8)
```

Fig. 2.1: Functor (2.1), Applicative (2.2, 2.3), and Alternative (2.4–2.8) laws.

Most of the laws in fig. 2.1 have already been shown to hold for Parsley in [Willis and Wu 2018]. An additional proof for eq. (2.8) can be found in **??**.

In the previous example, it is evident that the most noise results from the `empty` combinators. These can be eliminated using eqs. (2.4), (2.5), (2.7), and (2.8):

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

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

```
    (pure(ID).map(COMPOSE(a => b => a + b))).map(FLIP) <*> string("a")
=       { eqs. (2.2) and (2.3) }
    pure(COMPOSE(a => b => a + b)(ID)).map(FLIP) <*> string("a")
=       { eqs. (2.2) and (2.3) }
    pure(FLIP(COMPOSE(a => b => a + b)(ID))) <*> string("a")
=       { eq. (2.3) }
    string("a").map(FLIP(COMPOSE(a => b => a + b)(ID)))
```

This results in the most simplified form of the parser:

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

Implementation

These simplifications are applied akin to peephole optimisations in a bottom-up traversal of the recursive `Parser` ADT. There are many instances of Parsers, so this leads to a lot of boilerplate code recursing through each

case, which is prone to error when we implement it. To avoid this, we decouple the application of a generic transformation function from the recursive traversal through the datatype. This is still a hand-written traversal, but heavily inspired by the generic traversal patterns in [Mitchell and Runciman 2007]. Bottom-up transformation takes a partial function, applying the transformation at nodes where it is defined. The resulting `transform` method on parsers resembles the following (only a few cases shown for brevity):

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

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

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

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

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

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

2.1.5 Converting Parsers Back to Scalameta Terms

After the transformations on parsers are complete, they need to be converted back to a textual representation to be applied as a Scalafix patch. It is actually rather trivial to do so, by borrowing the pretty-printing capabilities of

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

Scalameta terms. This transformation is thus the inverse of the `fromTerm` transformation. This can be written using Scalameta quasiquotes to construct the `Term` nodes. The `Parser` trait defines this transformation as the method `term`, for example:

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

2.2 Function Representation

By the end of §2.1.4, one may notice that it would also be beneficial to simplify functions for the same reasons that parsers needed to be simplified. The transformations on parsers during left-recursion factoring also involve mapping over parsers with functions such as `flip` to reverse the order of arguments, and `compose` to compose functions.

We want to convert functions to a semantically equivalent but visually simpler form. This is the idea of normalisation. It may be useful to view this as statically evaluating as much as possible – however, not all inputs are known so we cannot fully evaluate the result value. Normalisation is therefore just evaluation, but in the presence of unknown terms.

2.2.1 Approach

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

Normalisation by Rewriting

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

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

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

Normalisation by Evaluation

Alternative, newer approach that doesn't require any rewriting at all. Instead, evaluate In NbE, a term is first evaluated in some 'semantic' model of language. The resulting value is reified back into a term. This gives a practical method for deciding definitional equality even in complicated dependent type theories, where rewriting becomes prohibitively complicated.

2.2.2 Converting Scalameta Terms to the Function ADT

Eta-expansion.

2.2.3 Normalising Function Terms

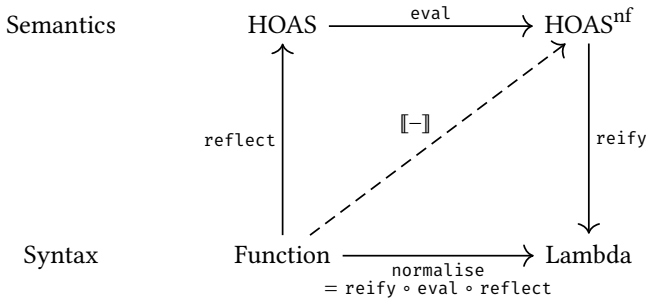

Fig. 2.2: Normalisation by evaluation for the `Function` datatype.

2.2.4 Future Work?

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

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

* Extracting method arguments (alongside their types) is very painful * Need to unify information from signature (within symbolinformation) and synthetics * synthetics exist in certain cases: .apply methods, showing the concrete type of a generic argument, implicit conversions * from https://scalacenter.github.io/scalafix/docs/developers/semantic-tree.html: SemanticTree is a sealed data structure that encodes tree nodes that are generated by the compiler from inferred type parameters, implicit arguments, implicit conversions, inferred .apply and for-comprehensions.

* map, lift (implicit and explicit), zipped, (.as perhaps?) – these should surely boil down into two cases: (x, y).xxx(f) and xxx(f, x, y) * named function literals (val) * named method literals (def) * anonymous functions i.e.

lambdas * functions with placeholder syntax * apply methods of case classes - symbol will tell its a class signature so we use this as a clue to look at synthetics??? * generic bridges – I reckon the information will probably show up in synthetics again

* 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

2.3 Limitations

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