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 Simplify Parser

* Catch cases when user manually writes out a parser that is already defined in the library * Apply parser laws, re-using Parser and Func representations to do cool things prettify = resugar . normaliseFunctions . simplify

1.2 Ambiguous Implicit Conversions

Problem

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

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

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

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

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

```
//   are possible conversion functions from String("arnish") to Parsley[?]
//    val p = 'g' ~> "arnish"
//                   ^^^^^^^^
```

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

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

Solution

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

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

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

Example

Fig. 1.1a 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. 1.1b.

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

* 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)‘

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

```scala
object parser {
  import parsley.syntax.character.stringLift
  import lexer.implicits._

  val p = "garnis" <~ 'h'
}

object lexer {
  import parsley.token.Lexer, parsley.token.descriptions.LexicalDesc

  private val lexer = new Lexer(LexicalDesc.plain)
  val implicits = lexer.lexeme.symbol.implicits
}
```

(a) 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._
  ^^^^^^^^^^^^^^^^^^^^^^^^
```

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

Fig. 1.1: Example of the *Ambiguous Implicit Conversions* lint rule in action.

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

1.3 Remove Explicit Usage of Implicit Conversions

1.4 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

1.5 Left Recursion Factoring

Chapter 2
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 (§2.1) and functions (§2.2). Show that terms must be simplified to a normal form Demonstrate equivalence to dsl optimisations in staged metaprogramming

2.1 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(Nil), Term.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. 2.1 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. 2.1: 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 (§1.5) rule as a basis/context to demonstrate the utility of this representation.

Running example The left-recursion factoring rule (§1.5) 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")
```

2.1.1 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.

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 `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 momenet – 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.

2.1.3 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.

2.1.4 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 `Parser` 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. 2.2 shows the subset of parser laws utilised by `parsley-garnish` for parser simplification. Most of the laws in fig. 2.2 have already been shown to hold for Parsley by Willis and Wu [2018]; an additional proof for eq. (2.8) can be found in **??**.

$$\texttt{p.map(f).map(g)} = \texttt{p.map(g compose f)} \tag{2.1}$$
$$\texttt{pure(f) <*> pure(x)} = \texttt{pure(f(x))} \tag{2.2}$$
$$\texttt{pure(f) <*> x} = \texttt{x.map(f)} \tag{2.3}$$
$$\texttt{empty | u} = \texttt{u} \tag{2.4}$$
$$\texttt{u | empty} = \texttt{u} \tag{2.5}$$
$$\texttt{pure(x) | u} = \texttt{pure(x)} \tag{2.6}$$
$$\texttt{empty <*> u} = \texttt{empty} \tag{2.7}$$
$$\texttt{empty.map(f)} = \texttt{empty} \tag{2.8}$$

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

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(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. (2.2) and (2.3) }
    pure(compose(a => b => a + b)(identity)).map(flip) <*> string("a")
=       { eqs. (2.2) and (2.3) }
    pure(flip(compose(a => b => a + b)(identity))) <*> string("a")
```

$=$ { eq. (2.3) }

```
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.

2.1.5 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

2.2 Function Representation

Section 2.1 showed that it is useful to lift Scala AST nodes to a specialised `Parser` AST, making it easier to manipulate and inspect parsers. Crucially, this allowed us to simplify parsers via term-rewriting rules based on parser laws. Section 2.1.4 demonstrated why this is necessary for `parsley-garnish`: transformations such as left-recursion factoring §1.5 result in complex parser terms that must be simplified to be readable.

However, so far, the functions passed to lifting combinators such as `map` have been treated as opaque, and have not been subject to the same simplifications as parsers. This is evident from where we left off in the example from the previous section:

```
val f = flip(compose(a => b => a + b)(identity))
// f is equivalent to (a => b => b + a)
```

This mess is an artefact of the left-recursion factoring transformation – recombination of unfolded parsers requires using higher-order functions such as `flip` and `compose`. Yet again, any user would find it unacceptable if `parsley-garnish` gave this as the output of a transformation. Therefore, these functions must be *normalised* into a semantically equivalent but syntactically simpler form.

This section explores how function term normalisation can be achieved.

2.2.1 The *n*-ary Lambda Calculus

Once again, to avoid the complexity of manipulating raw Scalameta AST nodes, we can build an intermediate AST representation for function terms to make them easier to work with.

Scala, as a functional programming language, uses an extension of the λ-calculus [Church 1936] as its theoretical foundations [Cremet et al. 2006; Amin et al. 2016]. The function terms that we want to normalise are equivalent to λ-terms, just with extra syntactic sugar. In the standard 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*. Scala supports curried functions using multiple parameter lists, but 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, our representation will be based on fig. 2.3, which extends the λ-calculus to support proper multi-argument functions using n-ary abstraction and application.

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

$$
\begin{array}{rll}
M, N ::= & x & \text{variable} \\
| & (\lambda \overline{\mathbf{x}}.\ M) & n\text{-ary abstraction, where } \overline{\mathbf{x}} = (x_1, \ldots, x_n) \\
| & (M\, \overline{\mathbf{N}}) & n\text{-ary application, wher } \overline{\mathbf{N}} = (N_1, \ldots, N_n)
\end{array}
$$

Fig. 2.3: Syntax for the untyped λ-calculus extended with n-ary abstraction and application.

β-Reduction and α-Conversion

In the λ-calculus, terms are evaluated via β-reduction: fig. 2.4 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 $\overline{\mathbf{x}}$ are equal to the number of arguments in $\overline{\mathbf{N}}$; otherwise, evaluation is stuck.

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

Fig. 2.4: The β-reduction rule for the n-ary lambda calculus.

The syntax $M[N/x]$ 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]. This requires performing α-conversion, which renames bound variables to avoid capture. In the λ-calculus, two terms are considered α-equivalent if they can be transformed into each other by renaming bound variables: the term $\lambda x.x$ is equivalent to $\lambda y.y$.

For example, substitution without α-conversion incorrectly β-reduces the following term:

$$
\begin{aligned}
(\lambda x.\lambda y.xy)y &\to_\beta (\lambda y.xy)\ [y/x] \\
&= \lambda y.yy
\end{aligned}
$$

The y we wanted to substitute was originally a free variable, distinct from the y bound in the lambda $\lambda y.xy$. However, after substitution, it became captured under the lambda, where the two y terms are now indistinguishable in the incorrect expression $\lambda y.yy$. The correct β-reduction with capture-avoiding substitution would instead proceed as follows:

$$
\begin{aligned}
(\lambda x.\lambda y.xy)y &\to_\beta (\lambda y.xy)\ [y/x] \\
&=_\alpha (\lambda z.xz)\ [y/x] \\
&= \lambda z.yz
\end{aligned}
$$

2.2.2 Representing Names

There exists a plethora of approaches to implementing the λ-calculus, mostly differing in how they represent variable names. This affects how variable capture is handled, and also how α-equivalence of two terms can be determined. For `parsley-garnish`, cheap α-equivalence is desirable to help check equivalence of parser terms, which is useful for some transformations.

Naïve capture-avoiding substitution Representing variable names as strings is the most straightforward approach in terms of understandability. The example below shows how the simply-typed λ-calculus can be represented as a generalised algebraic data type (GADT) in Scala:

```
type VarName = String

trait Lambda
```

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

// λf. λx. f x
val f = Var("f")
val x = Var("x")
val expr = Abs(f, Abs(x, App(f, x)))
```

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 whole 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, to maintain this invariant, variables must also be renamed during substitution – this administrative renaming has relatively high performance overhead and chews through a scarily large number of fresh variable names. The approach has been successfully optimised to very impressive performance, though: the Haskell GHC compiler uses Barendregt's convention with 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 Using *higher-order abstract syntax* (HOAS) [Pfenning and Elliott 1988] sidesteps variable binders entirely by borrowing substitution from the meta-language. This makes it the meta-language's responsibility to handle variable capture instead. In contrast, the previous techniques were examples of first-order abstract syntax, which represents variables and unknowns with identifiers (whether with names or indices). A HOAS approach does not name bound variables, instead representing them as bindings in the meta-language:

```
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]

// λf. λx. f x
val expr = Abs(f => Abs(x => App(f, x)))
```

Therefore, this representation performs substitution through Scala's function application, which makes it extremely fast compared to the other approaches. However, since lambda abstractions are represented as lambda expressions within Scala itself, the function body becomes wrapped under Scala's variable binding, making them difficult to work with.

2.2.3 Implementing Normalisation

Deciding on a concrete ADT representation for the λ-calculus is a surprisingly complex problem. This section explores the different approaches to normalisation, and the effect this has on how the λ-calculus may be represented as a datatype. We first explain the traditional notion of reduction-based normalisation, before introducing normalisation by evaluation as a more elegant alternative.

Reduction-Based Normalisation

This rule 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.

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) [Filinski and Korsholm Rohde 2004] achieves this by evaluating syntactical terms into a semantic model, then *reifying* them back into the syntactic model. The denotational model (denoted by $[\![-]\!]$) is specifically constructed to be *residualising*, meaning that terms can be extracted out into the original syntactic representation. Normalisation is then just defined as the composition of these two operations, as illustrated in fig. 2.5.

2.2.4 The Function ADT

Evaluating Performance of Normalisation Strategies

2.2.5 Representing Functions

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 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

$$\text{Syntactic domain} \underset{\textit{reify}}{\overset{[\![-]\!]}{\rightleftarrows}} \text{Semantic domain}$$

$$\textit{normalise} = \textit{reify} \circ [\![-]\!]$$

Fig. 2.5: Normalisation by evaluation in a semantic model.

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.

The Function ADT

The syntactic representation of functions is thus represented as the uni-typed `Function` ADT in fig. 2.6. `Defunc` is the higher-level representation of defunctionalised higher-order functions, which are desugared into `Lambda` terms for normalisation. `Lambda` represents n-ary λ-terms, extended with the following:

- Optional explicit type annotations for variables – these are not used for type-checking, but are there to preserve Scala type annotations originally written by the user.
- `Translucent` terms to encapsulate open terms holding a `scala.meta.Term` which cannot be normalised further. These carry an environment of variable bindings to substitute back in during pretty-printing – this is analogous to splicing into quoted expressions in the meta-programming world.

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 this[2]. 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.

2.2.6 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
```

[2] https://github.com/scalacenter/scalafix/issues/1583

```
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, displayType: Option[scala.meta.Type]) 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. 2.6: The `Function` ADT for representing functions.

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:

```
λ(_l1). λ(_l2, _l3). Translucent(_l2 + _l3, env = {_l1 → _l1, _l2 → _l2, _l3 → _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 list[3] – 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:

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

```
(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.

2.2.7 Normalising Function Terms

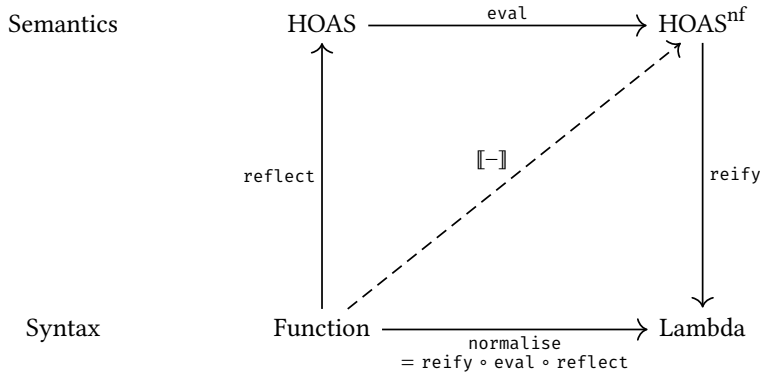

Fig. 2.7: Normalisation by evaluation for the `Function` datatype.

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) =>
        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))
  }
}
```

2.2.8 Converting Functions back to Scalameta Terms

Surprise bitches same shit again, quasiquotes ftw

2.2.9 Further Work?

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

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

* Extracting method arguments (alongside their types) is very painful * Need to unify information from signature (within symbolinformation) and synthetics * synthetics exist in certain cases: .apply methods, showing the con-

crete 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

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.