## Chapter 1

# Simplifying Parsers and Expressions

Shit output from previous section. This motivates:

- §1.1 discusses how parser terms can be simplified via domain-specific optimisations based on parser laws.
- §1.2 discusses how expressions can be partially evaluated. This is achieved using another intermediate AST, this time based on the $\lambda$-calculus.

## 1.1 Simplifying Parsers

This is where the deep embedding approach comes to shine: simplifications are easily expressed by pattern matching on `Parser` constructors.

- `parsley` performs rewrites on the parser AST to produce more optimised *code*.
- `parsley-garnish` performs rewrites on the parser AST to produce a more readable *textual representation of code*.

### 1.1.1 Parser Laws

Willis, Wu, and Pickering [2020] note that parser combinators are subject to *parser laws*, which often form a natural simplification in one direction. Both `parsley` Scala [Willis and Wu 2018] and `parsley` Haskell [Willis 2023] use these laws as the basis for high-level optimisations to simplify the structure of deeply-embedded parsers. These same principles can be used by `parsley-garnish` to simplify parser terms to be more human-readable.

Fig. 1.1 shows the subset of parser laws utilised by `parsley-garnish` for parser simplification. Most of the laws in fig. 1.1 have already been shown to hold for Parsley by Willis and Wu [2018]; an additional proof for eq. (1.8) can be found in ??.

```
p.map(f).map(g) = p.map(g compose f)      (1.1)
  pure(f) <*> pure(x) = pure(f(x))        (1.2)
      pure(f) <*> x = x.map(f)            (1.3)
           empty | u = u                  (1.4)
           u | empty = u                  (1.5)
        pure(x) | u = pure(x)             (1.6)
         empty <*> u = empty              (1.7)
        empty.map(f) = empty              (1.8)
```

Fig. 1.1: Functor (1.1), Applicative (1.2, 1.3), and Alternative (1.4–1.8) laws.

#### Simplifying the Example Parser

This section provides a worked example of how the parser in ?? is simplified using parser laws. Most of the noise in ?? comes from the large number of `empty` combinators. These can be eliminated using eqs. (1.4), (1.5), (1.7), and (1.8):

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

This already looks a lot better, but the second parameter to `postfix` can be further simplified as follows:

```
    (pure(identity).map(compose((_ + _).curried)).map(flip) <*> string("a")
 =     { eqs. (1.2) and (1.3) }
    pure(compose((_ + _).curried)(identity)).map(flip) <*> string("a")
 =     { eqs. (1.2) and (1.3) }
    pure(flip(compose((_ + _).curried)(identity))) <*> string("a")
 =     { eq. (1.3) }
    string("a").map(flip(compose((_ + _).curried)(identity)))
```

The most simplified form of the parser is then:

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

The parser has now been expressed in a much simplified form, in a similar style to how it would be written by hand. The remaining challenge is to simplify the contents of the expression `f`, which is tackled in §1.2.

### 1.1.2 Implementing Rewrites on the Parser AST

Lawful simplifications are applied by a bottom-up transformation over the recursively defined `Parser` AST. Since there are many parser cases, this inevitably leads to repetitive and error-prone boilerplate code which simply exists to recursively propagate the transformation through each case. To avoid this, the recursive traversal itself can be decoupled from the definition of the transformation function. Although the traversal is still hand-written, this implementation is inspired by the generic traversal patterns offered by Haskell's `uniplate` library [Mitchell and Runciman 2007].

The traversal 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, compose(g, f))
  // pure(f) <*> pure(x) == pure(f(x))
  case Pure(f) <*> Pure(x) => Pure(App(f, x))
  // u <|> empty == u
  case u <|> Empty => u
  // pure(f) <|> u == pure(f)
  case Pure(f) <|> _ => Pure(f)
  ...
}
```

**Extensibility and Safety** Further design considerations are made to ensure the extensibility 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. Although this formulation of the traversal is inspired by generic traversals, it still manually defines the traversal for each case: a safer approach would be to generically derive this. In Scala, this would require the use of an external dependency such as `shapeless`[1], which is frankly overkill given the relative simplicity of the `Parser` ADT.

## 1.2 Representing and Normalising Expressions

Currently, parsers such as `pure` and `map` take expressions as `scala.meta.Term` values, which have been treated as black boxes. No steps have been taken to improve the static inspectability of these values. This is evident from where we left off in the example from the previous **??**:

```
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.

### 1.2.1 The $n$-ary Lambda Calculus

Once again, the complexity of manipulating the generic Scalameta AST can be avoided by building a new intermediate AST representation for expression terms.

Scala, as a functional programming language, uses an extension of the $\lambda$-calculus [Church 1936] as its theoretical foundations [Cremet et al. 2006; Amin et al. 2016]. The expression terms that we want to normalise are equivalent to $\lambda$-terms, just with extra syntactic sugar. In the standard $\lambda$-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

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

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, the expression AST will be based on fig. 1.2, which extends the $\lambda$-calculus to support proper multi-argument functions using $n$-ary abstraction and application.

#### $\beta$-Reduction and $\alpha$-Conversion

In the $\lambda$-calculus, terms are evaluated via $\beta$-reduction: fig. 1.3 shows how this can be defined for the $n$-ary $\lambda$-calculus. Unlike the standard $\lambda$-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.

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]. Avoiding capture is achieved by performing $\alpha$-conversion, which is the process of renaming bound variables. In the $\lambda$-calculus, two terms are considered $\alpha$-equivalent if they can be transformed into each other by only renaming bound variables: the term $\lambda x.x$ is equivalent to $\lambda y.y$.

**Illustrating variable capture** For example, substitution without $\alpha$-conversion incorrectly $\beta$-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 $y$ that was substituted 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 $\beta$-reduction with capture-avoiding substitution would instead proceed as follows:

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

### 1.2.2 Representing Names

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

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

Fig. 1.2: Syntax for the untyped $\lambda$-calculus extended with $n$-ary abstraction and application.

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

Fig. 1.3: The $\beta$-reduction rule for the $n$-ary lambda calculus.

**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 $\lambda$-calculus can be represented as a generalised algebraic data type (GADT) [Cheney and Hinze 2003] 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)))
```

Although naïvely substituting these terms seems logically simple, it 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. Due to the inefficiency of having to traverse the whole term tree multiple times, this approach is not used in any real implementation of the $\lambda$-calculus. Furthermore, checking $\alpha$-equivalence is also tedious, requiring another full traversal of the term tree to compare variable names.

**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 incurs a 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 further 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. This makes $\alpha$-equivalence trivial to check, as it is just a matter of comparing the indices. 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 – this is undesirable given that the purpose of our ADT is to normalise $\lambda$-terms. 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 $\lambda$-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.

### 1.2.3 Normalisation Strategies

One remaining hurdle stands before deciding on an ADT representation: how normalisation will be implemented. 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 briefly explains the traditional notion of reduction-based normalisation, before introducing normalisation by evaluation as a more elegant and efficient strategy.

#### Reduction-Based Normalisation

The $\beta$-reduction rule is a *directed* notion of reduction, which can be implemented as a syntax-directed term-rewriting system, in a similar way to how `Parser` terms are simplified. The goal is to achieve beta normal form ($\beta$-NF) by allowing $\beta$-reduction to occur deep inside $\lambda$-terms, in all redexes of a term, until no more reductions can be made.

#### Normalisation by Evaluation

An interesting alternative strategy 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 domain. The denotational model (denoted by $[\![ - ]\!]$) generally requires implementing a separate datatype from the syntactic AST representation of functions. The semantics 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. 1.4.

$$\begin{tikzcd}
\text{Syntactic domain} \arrow[r, shift left, "{[\![ - ]\!]}"] & \text{Semantic domain} \arrow[l, shift left, "\mathit{reify}"]
\end{tikzcd}$$

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

Fig. 1.4: Normalisation by evaluation in a semantic model.

### 1.2.4 The Expression ADT

The final implementation of the `Expr` AST normalises terms with NBE, which results in a two-tiered representation of expression terms:

1. Scalameta AST nodes corresponding to expressions are lifted to the `Expr` ADT, which represents the syntax of lambda expressions using a simple named approach.
2. `Sem` uses HOAS to leverage Scala semantics as the denotational model for lambda expressions. During normalisation, `Expr` terms are evaluated into `Sem`, then reified back into `Expr`.

This helps achieve the following desired properties:

- The syntactic `Expr` ADT is represented in a simple manner, which is easy to construct and manipulate as opposed to a HOAS representation. This allows function terms to be pattern matched on, as part of parser simplifications.
- Lifting the syntactic constructs to Scala semantics with HOAS unlocks extremely efficient normalisation, and easier guarantees of correctness with respect to variable capture.
- Reifying `Sem` terms back into syntactic `Expr` terms automatically $\alpha$-converts names, granting $\alpha$-equivalence for free.

Fig. 1.5a shows the implementation of the untyped `Expr` ADT representing the abstract syntax of $n$-ary $\lambda$-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 – in a metaprogramming context, this is analogous to splicing into a quoted expression.

This structure is largely mirrored in the HOAS-based `Sem` ADT shown in fig. 1.5b, which allows it to be reified back into `Expr` terms.

`Expr` also implements some helper objects to make it easier to construct and deconstruct single-parameter abstractions and applications:

```
object Abs {
  def apply(x: Var, f: Expr) = AbsN(List(x), f)
  def unapply(func: AbsN): Option[(Var, Expr)] = func match {
    case AbsN(List(x), f) => Some((x, f))
    case _ => None
  }
}

object App {
  def apply(f: Expr, x: Expr) = AppN(f, List(x))
  def apply(f: Expr, xs: Expr*) = xs.foldLeft(f)(App(_, _))
}
```

Using these objects, fig. 1.6 shows how the higher-order functions necessary for left-recursion factoring can be implemented as constructors for `Expr` terms.

```
trait Expr
case class AbsN(xs: List[Var], f: Expr) extends Expr
case class AppN(f: Expr, xs: List[Expr]) extends Expr
case class Var(name: VarName, displayType: Option[scala.meta.Type]) extends Expr
case class Translucent(t: Term, env: Map[VarName, Expr]) extends Expr
```

(a) The Expr ADT for representing the abstract syntax of lambda expressions.

```
trait Sem
case class Abs(paramTypes: List[Option[scala.meta.Type]], f: List[Sem] => Sem) extends Sem
case class App(f: Sem, xs: List[Sem]) extends Sem
case class Var(name: VarName, displayType: Option[scala.meta.Type]) extends Sem
case class Translucent(t: Term, env: Map[VarName, Sem]) extends Sem
```

(b) The Sem ADT for representing the residualising semantics of lambda expressions.

Fig. 1.5: The intermediate AST for expressions.

```
/* id : A => A */
def id: Expr = {
  val x = Var.fresh()
  Abs(x, x)
}

/* flip : (A => B => C) => B => A => C */
def flip: Expr = {
  val (f, x, y) = (Var.fresh(), Var.fresh(), Var.fresh())
  Abs(f, Abs(x, Abs(y, App(f, y, x)))) // λf. λx. λy. f y x
}

/* compose : (B => C) => (A => B) => A => C */
def compose: Expr = {
  val (f, g, x) = (Var.fresh(), Var.fresh(), Var.fresh())
  Abs(f, Abs(g, Abs(x, App(f, App(g, x))))) // λf. λg. λx. f (g x)
}
def compose(f: Expr) = App(compose, f)
def compose(f: Expr, g: Expr) = App(compose, f, g)
```

Fig. 1.6: Constructors for higher-order functions represented as $\lambda$-expressions in Expr.

**Improved type safety** The originally intended design was to represent `Expr` as a type-parameterised GADT for improved type safety, where it would be based on a *typed* variant of the $\lambda$-calculus. This would've also allowed `Parser` to be represented as a GADT 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 complex and brittle due to its reliance on unstable compiler internals, which undermines Scalafix's goal of being a cross-compatible, higher-level abstraction over compiler details.
- 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.

#### Evaluating Performance of Normalisation Strategies

`parsley-garnish` originally used a named approach with Barendregt's convention, generating fresh variable names using an atomic counter. However, this required an extra $\alpha$-conversion pass to clean up variable names before pretty-printing the term, since the fresh variable names were very ugly. TODO: graphs of benchmarks and comparison (nbe is orders of magnitude faster lol)

### 1.2.5 Lifting to the Intermediate Expression AST

The `Parser` AST is amended to take `Expr` arguments where they used to take `scala.meta.Term` values. Take the `Pure` parser as an example: TODO: highlight changes like Jamie's PhD thesis? Can I get the latex source for the nicely configured tcolorboxes lol

```
case class Pure(x: Expr) extends Parser
object Pure {
  def fromTerm: PartialFunction[Term, Pure] = {
    case Term.Apply(matcher(_), Term.ArgClause(List(func), _)) => Pure(func.toExpr)
  }
}
```

The `toExpr` extension method on `scala.meta.Term` is used to lift `Term` AST nodes to `Expr` terms. As a high-level overview, there are three cases to consider during this conversion:

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

```
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 $\alpha$-converted to fresh names. The following example shows why this necessary:

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

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

Although no sane Scala programmer would write this, this lambda demonstrates how variable shadowing is possible – 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 `Expr` term would then resemble the following $\lambda$-calculus expression:

```
λ(x1). λ(x2, x3). Translucent(x2 + x3, env = {x1 → x1, x2 → x2, x3 → x3})
```

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 `Expr` terms – this is analogous 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. Placeholder syntax creates a fully uncurried function with a single parameter list[3]. Therefore, the converted `Expr` term is always a single $n$-ary abstraction, where the arguments are 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 $\eta$-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 $\eta$-expanded form of `AsciiCode.apply` would be as follows:

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

To $\eta$-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.

### 1.2.6 Normalising Expression Terms

Using NBE, normalisation therefore follows a two-step process: `Expr` values `evaluate` into `Sem` values, which are then `reify`ed back into `Expr`:

```
trait Expr {
  def normalise: Expr = this.evaluate.reify
}
```

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

**Evaluation** Evaluation proceeds by carrying an environment mapping bound variables to their semantic representations.

- Evaluating a variable looks up its name in the environment.
- Evaluating a lambda abstraction produces a closure using the current environment. HOAS allows these closures to be represented as native Scala closures.
- The interesting case is evaluating function application: the function and its arguments are evaluated separately at first. The magic of NBE takes place here – if the function evaluates to an abstraction, the arguments are passed to the Scala function `g: List[Sem] => Sem`, collapsing the structure by one step.
- Evaluating a translucent term is largely uninteresting, just propagating the environment through.

```
trait Expr {
  def evaluate: Sem = {
    def eval(func: Expr, boundVars: Map[Var, Sem]): Sem = func match {
      case v @ Var(name, displayType) =>
        boundVars.getOrElse(v, Sem.Var(name, displayType))
      case AbsN(xs, f) =>
        Sem.Abs(xs.map(_.displayType), vs => eval(f, boundVars ++ xs.zip(vs)))
      case AppN(f, xs) => eval(f, boundVars) match {
        case Sem.Abs(_, g) => g(xs.map(eval(_, boundVars)))
        case g => Sem.App(g, xs.map(eval(_, boundVars)))
      }
      case Translucent(term, env) =>
        Sem.Translucent(term, env.mapValues(eval(_, boundVars)))
    }

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

**Reification** Once the syntactic terms are fully evaluated into their semantics, the expression has been normalised to $\beta$-NF. Reification is then rather simple, just converting each level of the term back into its syntactic counterpart. When a lambda abstraction is reified, bound variables are assigned names from a fresh name supply. This step is what grants $\alpha$-equivalence for free, as the fresh name generator can be made deterministic: given two terms of the same semantic structure, reifying both will yield syntactic representations with the same names.

```
trait Sem {
  def reify: Expr = {
    def reify0(func: Sem)(implicit freshSupply: Fresh): Expr = func match {
      case Abs(tpes, f) =>
        val params = tpes.map(Expr.Var(freshSupply.next(), _))
        Expr.AbsN(params, reify0(
          f(params.map { case Expr.Var(name, tpe) => Sem.Var(name, tpe) } )
        ))
      case App(f, xs) => Expr.AppN(reify0(f), xs.map(reify0))
      case Translucent(t, env) => Expr.Translucent(t, env.mapValues(reify0))
      case Var(name, displayType) => Expr.Var(name, displayType)
    }
```

```
    reify0(this)(new Fresh)
  }
}
```

### 1.2.7 Lowering Back to the Scalameta AST

Surprise bitches same shit again, quasiquotes ftw

### 1.2.8 Discussion

TODO: comparison with parsley haskell

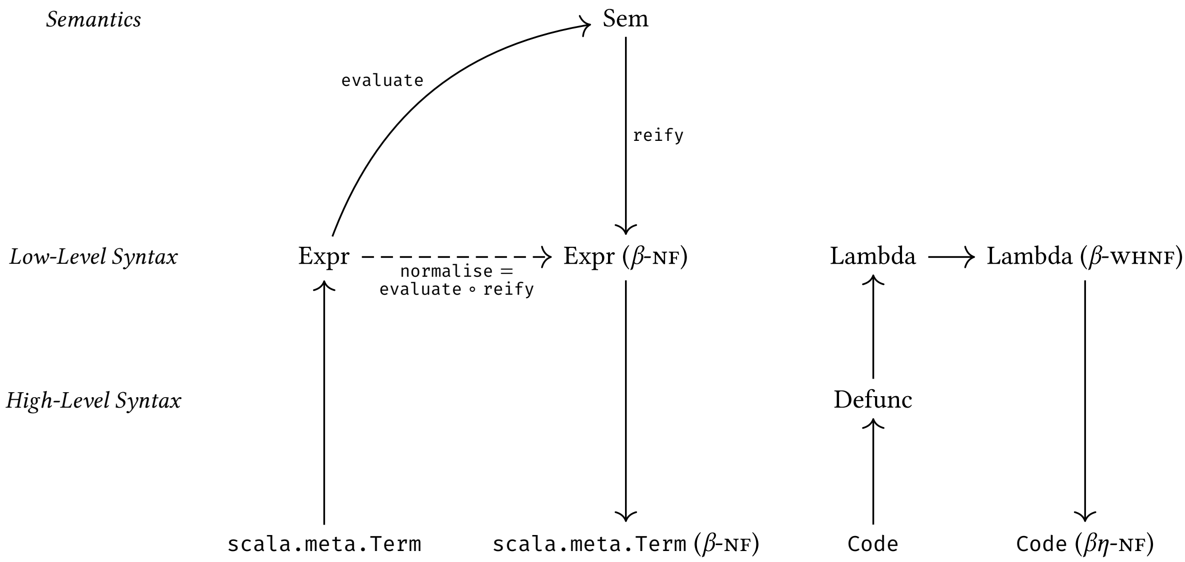


Fig. 1.7: Comparison of expression normalisation in `parsley-garnish` (left) and `parsley` Haskell (right).

### 1.2.9 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: 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).