Chapter 1
Simplifying Parsers and Expressions

At this stage, the left-recursion factoring transformation leaves a lot to be desired in terms of output quality. This chapter takes a step back from linting rules and focuses on ensuring how transformed terms can get pretty-printed in a human-readable form. The following ideas are explored:

- First, §1.1 discusses how parsers can be simplified via domain-specific optimisations based on parser laws.
- Afterwards, §1.2 discusses how expressions can be partially evaluated to some extent. This is achieved using another intermediate AST, this time based on the λ-calculus, which unlocks the idea of β-reduction and normalisation as tools to reduce the complexity of these terms.

1.1 Simplifying Parsers

Reusing a similar abstract syntax representation as `parsley` itself unlocks some interesting insights for `parsley-garnish`. Gibbons and Wu [2014] note that a deep-embedded DSL consists of two components:

1. A representation of the language's abstract *syntax*, in the form of the aforementioned datatype.
2. Some traversals over the datatype, which gives *semantics* to that syntax.

A deep-embedded DSL and a linter for that DSL can thus be viewed as two different semantic interpretations over the same abstract syntax:

- The DSL semantics are *evaluation*. The syntactic structure may be transformed for optimisation purposes before generating code to be evaluated.
- The linter's semantics are two-fold for lint diagnostics and code rewrites:
  - *Emitting side-effects* as diagnostics, based on patterns of interest within the syntactic structure.
  - *Pretty-printing* a transformation over the syntactic structure, as a rewrite action – crucially, unlike evaluation, the transformed output is not converted into code but rather a textual representation to be rewritten over the original source file. The output of this transformation may benefit from *the same optimisation transformations as with the DSL semantics* to simplify the pretty-printed textual output.

This section shows that this is indeed the case for `parsley-garnish`: the same optimisation transformations apply for both `parsley` (the DSL) and `parsley-garnish` (the linter). The only difference lies in the purpose of performing these transformations:

- `parsley-garnish` needs to perform simplifications on the `Parser` AST to produce output of hand-written quality, or else the resulting parser would be unreadable, as was the case in **??**.
- `parsley` performs simplifications on its combinator tree to produce output of hand-written quality, in order to deliver excellent parser performance.

1.1.1 Parser Laws

Willis [2024] notes 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, Wu, and Pickering 2020; Willis 2024] use these laws as the basis for high-level optimisations to simplify the structure of deeply-embedded parsers. These same principles are used by `parsley-garnish` to simplify parser terms to resemble the natural style that a human would write by hand.

Fig. 1.1 shows the subset of parser laws utilised by `parsley-garnish` for parser simplification. Most of these laws 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.

1.1.1.1 Simplifying the Example Parser

It is useful to illustrate how these laws are used to simplify a parser term, by starting with the parser in ??. In the following example, function terms will be greyed out as they are still currently uninspectable – this is work that is addressed in the next section.

First of all, most of the noise in `example` comes from the large number of `empty` combinators. These can be eliminated using eqs. (1.4), (1.5), (1.7), and (1.8):

```scala
lazy val example: 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 final simplified form of the parser is then:

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

The parser is now expressed in a much simplified form, in a similar style to the hand-written example in ??! 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].

Partial functions Scala supports *partial functions*, which are functions only defined for a subset of its possible input values. If invoked on an undefined input, this results in a runtime error. A useful idiom utilising partial functions is the `collect` method on collections, which combines functionality of `filter` and `map` in a clean manner. This method will attempt to apply a partial function to each element in the collection, and only include the result if the function is defined. In fact, `collect` has already been used in earlier sections to inspect and gather AST nodes of interest in the Scalameta AST:

```
doc.tree.collect { case Term.Name(name) => name }
//                 ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
// A partial function only defined for Term.Name nodes, not on any other type of AST node
```

Rewrite rules on parsers are most naturally expressed as a partial function, since they are only defined for certain parser cases. Therefore, the traversal method `transform` takes a partial function, applying 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 p <*> q       => p.transform(pf) <*> q.transform(pf)
    case Zipped(f, ps) => ps.map(_.transform(pf)).zipped(f)
    case Pure(f)       => Pure(f)
    ...
  }
  // Apply the partial function if defined, otherwise return the original parser
  pf.applyOrElse(parser, identity[Parser])
}
```

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)
}
```

With this, 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 {
  case FMap(FMap(p, f), g) => FMap(p, compose(g, f))
  case Pure(f) <*> Pure(x) => Pure(App(f, x))
  case u <|> Empty         => u
  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 overkill given the relative simplicity of the `Parser` ADT.

Discussion

The design of the parser simplification process for `parsley-garnish` was not intended to closely follow the methods used in `parsley`, so it is remarkable that the two approaches have ended up being so similar. However, in retrospect, this resemblance is not surprising given that the act of parser simplification and optimisation are

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

fundamentally the same transformation. Since both `parsley` and `parsley-garnish` represent the parser AST as a deep-embedded structure, it is natural that this transformation is implemented similarly in both cases as a bottom-up traversal over the abstract syntax.

This insight can be extended to any deep embedded DSL based on an algebra, where constructs within the DSL are subject to algebraic laws and operations. It would be interesting to see an EDSL and linter pair that shares a unified data structure for its abstract syntax, in order to take full advantage of this duality.

1.2 Representing and Normalising Expressions

The previous section demonstrated the process of simplifying the `Parser` AST, but this is not the only syntactic structure that requires simplification. So far, parsers such as `pure` and `map` still treat expressions as black boxes in the form of raw `scala.meta.Term` AST nodes. This is evident from where the example in §1.1.1.1 left off, where the parser itself is in a simplified form, but the function passed to `map` is not:

```scala
val f = flip(compose((_ + _).curried)(identity))
```

Therefore, this section explores the following:

- How expressions can be represented as another intermediate AST, so that they are statically inspectable enough to be simplified.
- The notion of *normalisation*, in order to reduce expressions into a semantically equivalent but syntactically simpler form.

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 λ-calculus [Church 1936] as its theoretical foundations [Cremet et al. 2006; Amin et al. 2016]. The expression terms present within parsers 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. As a result, the expression AST will be based on fig. 1.2, which extends the λ-calculus to support proper multi-argument functions using *n*-ary abstraction and application.

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

Fig. 1.2: Syntax for the untyped λ-calculus extended with *n*-ary abstraction and application.

1.2.1.1 β-Reduction and α-Conversion

In the λ-calculus, terms are evaluated via β-reduction: fig. 1.3 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. 1.3: 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 [Bakel 2022]. Avoiding capture is achieved by performing α-conversion, which is the process of renaming bound variables. In the λ-calculus, two terms are considered α-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 α-conversion incorrectly β-reduces the following term:

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

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 have become indistinguishable in the incorrect expression $\lambda y.yy$. Instead, the correct β-reduction using capture-avoiding substitution would proceed as follows:

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

1.2.1.2 Simplifying the Example Expression

The example from the beginning of the section can thus be evaluated by hand via β-reduction, representing the higher-order functions as λ-abstractions:

$$\begin{aligned}\texttt{flip(compose((\_ + \_).curried)(identity))} &= \texttt{flip}\,(\texttt{compose}\,(\lambda a.\, \lambda b.\, a + b)\;\texttt{identity}) \\ &= \texttt{flip}\,((\lambda f.\, \lambda g.\, \lambda x.\, f\,(g\,x))(\lambda a.\, \lambda b.\, a + b)(\lambda x.\, x)) \\ &\to_\beta^* \texttt{flip}\,(\lambda g.\, \lambda x.\, (\lambda b.\, g\,x + b)(\lambda x.\, x)) \\ &\to_\beta^* \texttt{flip}\,(\lambda x.\, \lambda b.\, x + b) \\ &= (\lambda f.\, \lambda x.\, \lambda y.\, f\,y\,x)(\lambda x.\, \lambda b.\, x + b) \\ &\to_\beta^* \lambda x.\, \lambda y.\, y + x \\ &= \texttt{(x, y) => y + x}\end{aligned}$$

This normalised expression has the same meaning as the original, but is now expressed in a clearer form! This is much more suitable to be placed back into the code rewrite, as it is easier to understand by the user. The rest of the section now explores how this process can be implemented in `parsley-garnish`.

1.2.2 Representing Names

There are 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) [Cheney and Hinze 2003] in Scala:

```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]

val (f, x) = (Var("f"), Var("x"))
val expr = Abs(f, Abs(x, App(f, x))) // λf. λx. 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 λ-calculus. Furthermore, checking α-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. For example, $\lambda x.x(\lambda y.xy)$ would be represented as $\lambda.0(\lambda.1\ 0)$: the variable y is replaced with 0 to signify that y is bound by the closest λ binder. The first occurrence of x is replaced with 0 in the same way, but its second occurrence is replaced with 1 since it is bound by the λ two levels up.

An advantage of De Bruijn indexing is that it makes α-equivalence trivial to check, as it is just a matter of comparing the indices. However, 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 the AST is to normalise λ-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 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, making 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:

```scala
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]

val expr = Abs(f => Abs(x => App(f, x))) // λf. λx. 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 closures within Scala itself, the function body becomes wrapped under Scala's variable bindings, making them difficult to inspect and work with.

1.2.3 Normalisation Strategies

One remaining hurdle stands before deciding on an ADT representation: how normalisation will be implemented. The ideas of 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.

1.2.3.1 Reduction-Based Normalisation

The β-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

(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.

1.2.3.2 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 involves 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 achieves the following desired properties for `parsley-garnish`'s use cases:

- 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 α-converts names, granting α-equivalence for free.

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

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

```scala
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.

```scala
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.

Constructing higher-order functions `Expr` also implements some helper objects to make it more convenient to construct and deconstruct single-parameter abstractions and applications:

```scala
object Abs {
  // Convenience factory method to create a single-parameter abstraction
  def apply(x: Var, f: Expr) = AbsN(List(x), f)
  // Extractor allows user to pattern match on Abs(x, f)
  // instead of the more tedious 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 {
  // Convenience factory method to apply a single argument to a function
  def apply(f: Expr, x: Expr) = AppN(f, List(x))
  // Apply multiple arguments in sequence, as f(x)(y)(z) instead of f(x, y, z)
  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.

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 λ-calculus. This would have 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.

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

```
/* 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 λ-expressions in `Expr`.

- 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.

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:

```
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. Expression lifting is invoked whenever a parser expects an expression (whether a function or simple value) as an argument. This section gives a high-level overview of the three cases that `toExpr` handles.

1.2.5.1 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 α-converted to fresh names. The body is also transformed to make sure references to these bound variables use their new names: this conversion is well-scoped as it compares terms using their unique Scalameta symbols. The following example illustrates when this is necessary:

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

Although no sane Scala programmer would write this, this lambda demonstrates how it is possible to shadow variables – the a in the function body refers only to the a in the second parameter list, as it shadows the a in the first parameter list. The lifted `Expr` term would then resemble the following λ-calculus expression, where **bold** values correspond to `scala.meta.Term` nodes as opposed to `Expr` values:

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

This shows how the lambda body's environment maps `Term.Name` nodes to their corresponding variable terms. When the term is pretty-printed, each `Term.Name` node is replaced with their corresponding `Expr` term – this is analogous to the splicing operation on quasiquotes:

```
q"x1 => (x2, x3) => $x2 + $x3"
```

1.2.5.2 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.

1.2.5.3 Eta-Expansion

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

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

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

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

To η-expand `scala.meta.Term` nodes, `parsley-garnish` attempts to look up the method signature of its symbol using Scalafix's semantic API. This is not always possible – in that case, the term cannot 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, while evaluating a lambda abstraction produces a closure using the current environment – using HOAS allows these closures to be represented as native Scala closures. The interesting case is evaluating function application: this allows β-reduction within the *semantic domain* at any point within the term, not just on the head term. The function and its arguments are first evaluated separately – then, if the function evaluates to an abstraction, the arguments are passed to the Scala closure `g: List[Sem] => Sem`, collapsing the term structure by one step.

```scala
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 The NBE semantics utilised by parsley-garnish are *intensional* [Lindley 2005], meaning that once syntactic terms are fully evaluated into their semantics, the expression is normalised to β-NF. Reification is then a simple process of 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 α-equivalence for free, as the fresh name generator can be made deterministic: given two terms that evaluate to the same semantic structure, reifying both will yield syntactic representations with the same names.

```scala
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

Once any transformations on the `Expr` terms are complete, they need to be converted back to strings to be placed in a Scalafix patch. This is achieved in the same way as parsers in **??**, by lowering them back as `scala.meta.Term` ast nodes and re-using Scalameta's pretty-printer to generate a syntactically well-formed string representation. These can also be achieved using quasiquotes:

- Lambda abstractions are transformed into a lambda expression of form `q"(...params) => body"`.
- Function application is transformed into method calls.
- Variables are simply `Term.Name` nodes with their syntactic names.
- Translucent terms splice their environment bindings back into their term body.

Discussion

`parsley` Haskell, as a *staged* parser combinator library, also has the ability to inspect and optimise the code of user-defined functions. The approach taken by `parsley-garnish` and parsley share many similarities, both using the λ-calculus as a core language to normalise expressions. In both cases, the need to reduce expression terms is motivated by how parser simplifications involve fusion, which results in function applications that can be partially evaluated.

However, the two have different motivations and requirements for normalising expressions, so their approaches differ in some ways – fig. 1.7 illustrates these differences.

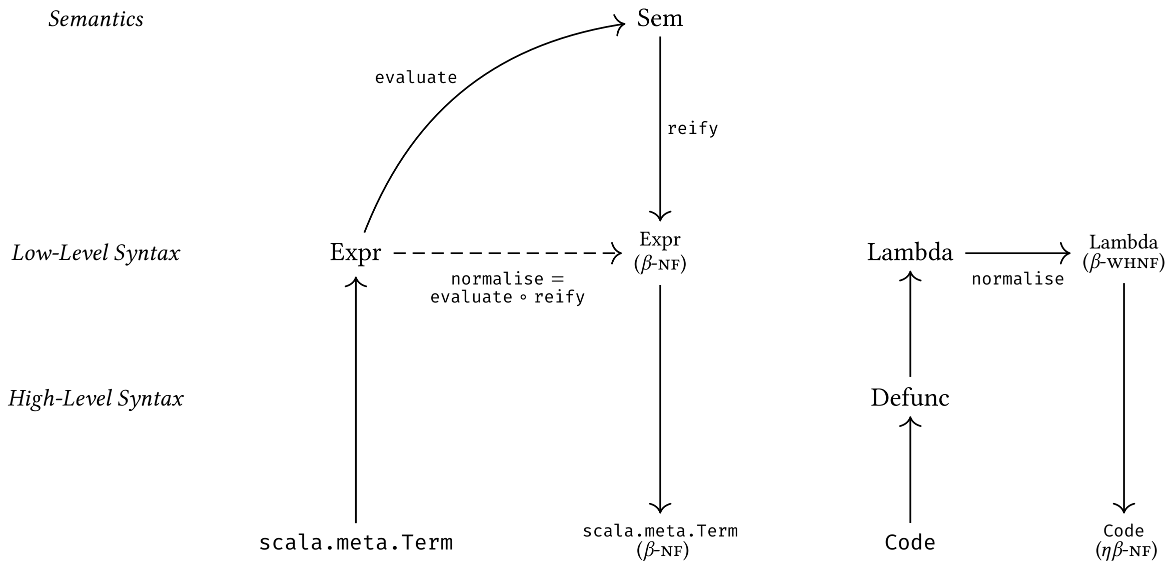

Fig. 1.7: Comparison of expression normalisation in `parsley-garnish` (left) and `parsley` Haskell (right).

Syntactic representation Unlike `parsley-garnish`, `parsley` has a two-level syntactic representation for expressions. `Defunc` is akin to a deep embedding of higher-order functions, representing them as a GADT: this process is known as *defunctionalisation* [Reynolds 1972; Danvy and Nielsen 2001]. This helps facilitate certain parser law optimisations which require pattern matching on functions as well as parsers, for example:

```
pure(identity) <*> u = u
```

After this step, `Defunc` values are then brought into the lower-level λ-calculus representation `Lambda`, to be normalised by β-reduction.

At the moment, `parsley-garnish` does not have a need to implement any parser simplifications based on these laws, although this may change in the future. Adding an extra defunctionalised layer to the expression AST would be fairly straightforward.

Normalisation strategy `parsley` normalises terms to full $\eta\beta$-NF, whereas `parsley-garnish` only normalises to β-NF. This is because η-reduction in Scala 2 is not as straightforward as in Haskell, and is not always possible – in most cases the appropriate reduction is instead to convert lambdas to placeholder syntax. Implementing this is left as future work.

In `parsley`, normalisation is implemented as a reduction-based approach over the HOAS `Lambda` datatype. Normalisation by β-reduction with Haskell function application brings this to β-WHNF. Then, code generation brings this further to β-NF as desired, as well as an extra step for η-reduction to put the term into full $\eta\beta$-NF.

The main reason why `parsley-garnish` takes a different normalisation approach is because unlike `parsley`, there is still a need for α-equivalence checking after normalisation. In `parsley`, the normalised forms are immediately utilised for code generation, so they can be kept as HOAS the entire time, without representing variables with any names. Conversely, in `parsley-garnish`, these normalised terms undergo further analysis before being transformed into code patches for pretty-printing.

Summary

This chapter introduced the idea of simplifying parsers and normalising expressions, by representing both as intermediate ASTs to improve their static inspectability. It also demonstrated how these processes are related to the optimisation techniques used in both `parsley` Scala and `parsley` Haskell.

With promising results applying these simplifications on the `example` parser from last **??**, the improved `Parser` AST unlocks the potential for more powerful and interesting transformations utilising specialised domain knowledge of parser combinators.