# 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.

In the previous example, it is evident that the most noise results from the `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")
)
```

The complicated term in the postfix operator can then be 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)))
```

This results in the most simplified form of the parser:

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

### 1.1.2 Implementing Rewrites on the Parser AST

Lawful simplifications are applied akin to peephole optimisations on the recursively defined `Parser` AST. 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.

### Summary

## 1.2 Representing and Normalising Expressions

**??** 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 1.1 demonstrated why this is necessary for `parsley-garnish`: transformations such as left-recursion factoring **??** 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 **??**:

```
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 raw Scalameta AST nodes can be avoided by building a new intermediate AST representation for function 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 function 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 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. 1.2, which extends the $\lambda$-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. 1.2: Syntax for the untyped $\lambda$-calculus extended with $n$-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.

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

Fig. 1.3: The $\beta$-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 $\alpha$-conversion, which renames bound variables to avoid capture. In the $\lambda$-calculus, two terms are considered $\alpha$-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 $\alpha$-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 $y$ 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 $\beta$-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.

**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)))
```

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 $\lambda$-calculus due to its inefficiency, as it requires traversing the whole term tree multiple times. 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 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].

`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.

**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 Implementing Normalisation

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 alternative.

#### 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 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 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.

### 1.2.4 The Function ADT

`parsley-garnish` opts to implement normalisation with NBE, which results in a two-staged representation of function terms:

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

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

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

1. Scalameta terms are lifted to the `Function` ADT, which represents the syntax of terms using a simple named approach.
2. `Sem` uses HOAS to leverage Scala's semantics as the denotational model. During normalisation, `Function` terms are evaluated into `Sem`, then reified back into `Function`.

This helps achieve the following:

- The syntactic `Function` 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 allows for extremely efficient normalisation, and easier guarantees of correctness with respect to variable capture.
- Reifying `Sem` terms back into syntactic `Function` terms automatically $\alpha$-converts names, giving us $\alpha$-equivalence for free.

#### Lambda Calculus

The syntactic representation of functions is thus represented as the uni-typed `Function` ADT in fig. 1.5. `Defunc` is the higher-level representation of defunctionalised higher-order functions, which are desugared into `Lambda` terms for normalisation. `Lambda` represents $n$-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 – 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 $\lambda$-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.

#### Evaluating Performance of Normalisation Strategies

### 1.2.5 Converting Scalameta Terms to the Function ADT

There are three cases to consider when converting `scala.meta.Term` nodes into `Function` terms.

[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. 1.5: The `Function` ADT for representing functions.

**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 $\alpha$-converted to fresh names. Take the following example to see why this is helpful:

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

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 $\lambda$-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.

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

**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 Function Terms

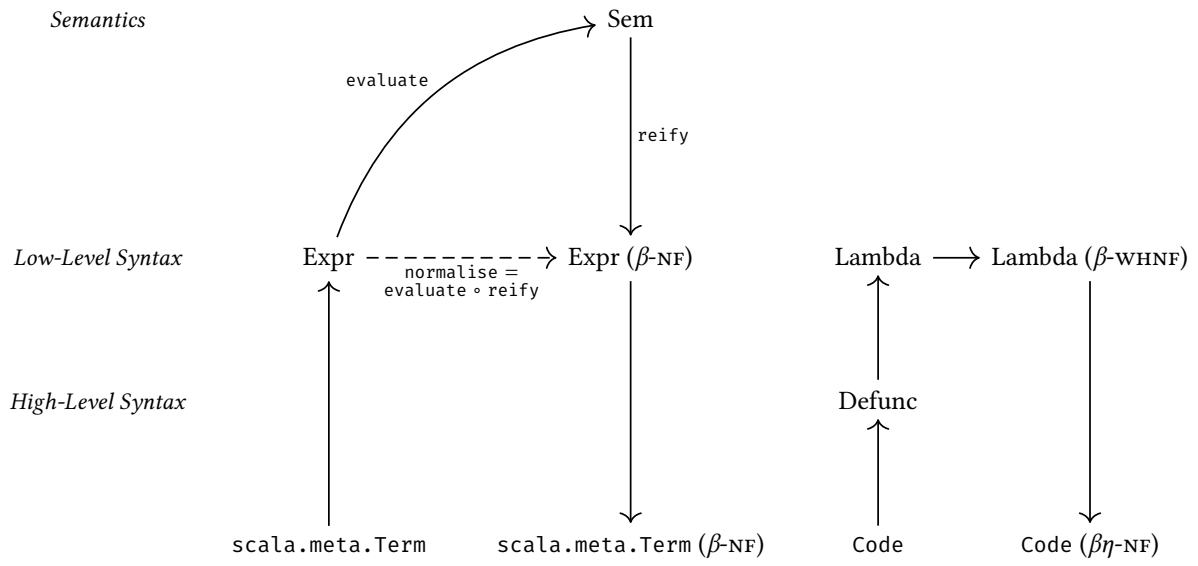


Fig. 1.6: Normalisation by evaluation for the `Expr` datatype.

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))
  }
}
```

### 1.2.7 Converting Functions back to Scalameta Terms

Surprise bitches same shit again, quasiquotes ftw

### 1.2.8 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 concrete type of a generic argument, implicit conversions * from https://scalacenter.github.io/scalafix/docs/developers/semantic-tree.html: SemanticTree is a sealed data structure that encodes tree nodes that are generated by the compiler from inferred type parameters, implicit arguments, implicit conversions, inferred .apply and for-comprehensions.

* 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