

## 0.1 Function Representation

This section is about simplifying in the general domain, so really Squid can do all of this? Still an interesting approach ig And shows a shortcoming of scalameta quasiquotes

?? showed that it is useful to lift AST nodes corresponding to parser terms into an intermediate `Parser` representation. Crucially, ?? demonstrates how this representation lends itself to reducing parsers to a simpler form. This section extends this idea to representing functions in a similar, statically-inspectable manner, and explores why this machinery is not just desirable but necessary for parsley-garnish.

### 0.1.1 Representing Functions

Utilising parser laws as a basis for term simplification for the `Parser` ADT has managed to tame the jumble of parser combinators resulting from the left-recursing factoring transformation ???. However, the functions passed to lifting combinators such as `map` have so far been treated as opaque, and have not been subject to the same simplifications as parsers.

Continuing from the running example from the previous ??, the function passed to the `map` combinator has ended up in an unnecessarily complex form:

```
val f: Function = flip(compose(a => b => a + b)(identity))
```

This is because the transformations on parsers during left-recursion factoring involve mapping over parsers with higher-order functions such as `flip` to reverse the order of arguments, and `compose` to compose functions. These are used to ensure that unfolded parsers are recombined in a lawful manner, but results in a lot of syntactic noise. Yet again, leaving the transformed terms in this form would be unacceptable for the linter. It is vital that these functions are partially evaluated and *normalised* into a semantically equivalent but visually simpler form. To achieve this, a representation of functions is needed that is amenable to the same kind of static inspection as the `Parser` ADT.

#### Defunctionalisation

In the same way that representing parsers as datatypes allows them to be statically inspectable, the aforementioned higher-order functions can also be rendered as datatypes to allow them to be analysed in a similar manner. This is an example of *defunctionalisation*, a technique to eliminate higher-order functions by transforming them into specialised first-order equivalents [Reynolds 1972; Danvy and Nielsen 2001].

Ideally for type safety, this could be represented as a generalised algebraic data type (GADT) [Cheney and Hinze 2003] with a type parameter documenting the type that the function term represents:

```
trait Function[+T]
case class Identity[A]() extends Function[A => A]
case class Flip[A, B, C]() extends Function[(A => B => C) => B => A => C]
case class Compose[A, B, C]() extends Function[(B => C) => (A => B) => A => C]
```

#### Lambda Calculus

The defunctionalised constructs alone are not enough to fully represent the functions passed to combinators. There needs to be a way to apply them to arguments, and to represent the arguments themselves – the core language to do so is the lambda calculus [Church 1936]. In the  $\lambda$ -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*. In Scala, functions are allowed to have multiple parameter lists to make them curried, although 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, fig. 1 shows the  $\lambda$ -calculus extended to represent multi-argument functions using  $n$ -ary abstraction and application.

$M, N ::= x$	(variable)
$\lambda(x_1, \dots, x_n).M$	( $n$ -ary abstraction)
$M(N_1, \dots, N_n)$	( $n$ -ary application)

Fig. 1: The untyped  $\lambda$ -calculus extended with  $n$ -ary abstraction and application.

### The Function ADT

The syntactic representation of functions is thus represented as the uni-typed `Function` ADT in fig. 2. The `Lambda` trait is a subset of the entire `Function` trait, representing  $\lambda$ -terms, with the addition of a `Translucent` term to encapsulate open terms holding a `scala.meta.Term` which cannot be evaluated further.

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<sup>1</sup>. 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.

### 0.1.2 Achieving Normalisation

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 first explains the traditional notion of normalisation by rewriting, before introducing normalisation by evaluation as a more elegant and efficient alternative.

#### Reduction-Based Normalisation

In the  $\lambda$ -calculus, terms are reduced in size by applying  $\beta$ -reduction: fig. 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  $\mathbf{x} = x_1, \dots, x_n$  are equal to the number of arguments in  $\mathbf{N} = N_1, \dots, N_n$ ; otherwise, the term is left as is. This is therefore a *directed* notion of reduction, which can be implemented as a term-rewriting system, in a similar way to how `Parser` terms are simplified.

The goal is to achieve beta normal form ( $\beta$ -NF) by allowing  $\beta$ -reduction to occur deep inside  $\lambda$ -terms, in all redexes of a term, until no more reductions can be made. Normally in the untyped  $\lambda$ -calculus,  $\beta$ -reduction as a rewriting rule does not manage to be strongly normalising as desired, and may lead to non-terminating rewrites. However, despite the `Lambda` representation itself being untyped, all `Lambda` terms are converted from valid typed Scala expressions. Scala 3 is modelled on the theoretic foundations of the Dependent Object Types (DOT) calculus, which has been shown to be strongly normalisable [Wang and Rompf 2017]. Although not proven for Scala 2, it is reasonable to assume that the same result holds, at least for the subset of expressions that are represented in the `Lambda` representation.

---

<sup>1</sup><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) extends Lambda
case class Translucent(t: Term, env: Map[VarName, Function]) extends Lambda

// Defunctionalised higher-order functions
case object Identity extends Function
case object Flip extends Function
case object Compose extends Function
...

```

Fig. 2: The `Function` ADT for representing functions.

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

Fig. 3: The  $\beta$ -reduction rule for the  $n$ -ary lambda calculus.

In fig. 3, the syntax  $M[\mathbf{N}/\mathbf{x}]$  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]. Capture-avoiding substitution is a notoriously tricky and subtle problem. There are a plethora of different approaches to solve this, mostly boiling down to how variable names get represented:

**Naïve substitution** Representing variable names as strings is the most straightforward approach in terms of understandability. 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 term tree multiple times.

**Barendregt’s convention** Renaming all bound variables to be unique satisfies **Barendregt’s convention** [Barendregt 1984], which removes the need to check for variable capture during substitution. However, variables still have to be renamed during substitution – this administrative renaming has relatively high performance overhead and chews through a scarily large number of fresh variable names. This approach can be massively optimised though: the Haskell GHC compiler uses a technique dubbed “the Rapier” [Peyton Jones and Marlow 2002], maintaining invariants to avoid renaming on substitution when unnecessary. Unfortunately, maintaining the invariants to keep this transformation correct becomes very difficult [Maclaurin, Radul, and Paszke 2023].

**Nameless and hybrid representations** Nameless representations like *De Bruijn indices* [de Bruijn 1972] eschew names entirely, instead representing variables as the number of binders between the variable and its binding site. Although an elegant representation, De Bruijn terms are notoriously difficult to work with, as they are not easily human-readable. Furthermore, performing substitutions with De Bruijn terms has an overhead as variable positions have to be shifted. To avoid this, hybrid representations combining named and nameless representations exist [McBride and McKinna 2004; Charguéraud 2012], but they become rather complex solutions for what should be a relatively simple  $\lambda$ -calculus implementation for parsley-garnish’s needs.

**Higher-order abstract syntax** *Higher-order abstract syntax* (HOAS) [Pfenning and Elliott 1988] is a representation that uses the meta-language’s own functions to represent the object language’s terms. **we will use this but also with nbe as shown in the next subsection**

## 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) evaluates syntactical terms into a semantic model, which is then *reified* back into the syntactic model [Filinski and Korsholm Rohde 2004]. Normalisation is then just defined as the composition of these two operations, as illustrated in fig. 4.

The denotational model (denoted by  $\llbracket - \rrbracket$ ) is specifically constructed to be *residualising*, meaning that terms can be extracted out into the original syntactic representation.

**Represent semantic model with HOAS**

```
trait HOAS
case class HAbs(n: Int, f: List[HOAS] => HOAS) extends HOAS
case class HApp(f: HOAS, xs: List[HOAS]) extends HOAS
case class HVar(name: VarName) extends HOAS
case class HTranslucent(t: Term, env: Map[VarName, HOAS]) extends HOAS
```

Abs represents n-ary  $\lambda$ -abstraction; App is n-ary function application; Var represents opaque free variables. Translucent is treated as opaque during normalisation, but contains normalisable terms in an environment, where they are substituted back in after normalisation.

### 0.1.3 Converting Scalameta Terms to the Function ADT

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

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

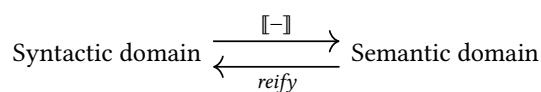
```
val asciiCode: Parsley[Int] = item.map(char => char.toInt)
```

These lambda expressions are represented in the Scalameta AST as `Term.Function` nodes, which are recursively traversed to collect all parameter lists. This is folded into a chain of *n*-ary abstractions, with the final term being the body of the lambda, which is wrapped into a `Translucent` term. To ensure that the parameter names in the `Translucent` body term are unique, the parameters are  $\alpha$ -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:

```
 $\lambda(_l1). \lambda(_l2, _l3). \text{Translucent}(_l2 + _l3, \text{env} = \{\_l1 \rightarrow \_l1, \_l2 \rightarrow \_l2, \_l3 \rightarrow \_l3\})$ 
```



$$\text{normalise} = \text{reify} \circ \llbracket - \rrbracket$$

Fig. 4: Normalisation by evaluation in a semantic model.

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<sup>2</sup> – this is filled with the freshly generated variable names in order of their occurrence in the expression body.

**Eta-Expansion** If the term is not a lambda expression, parsley-garnish attempts to  $\eta$ -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.

## 0.1.4 Normalising Function Terms

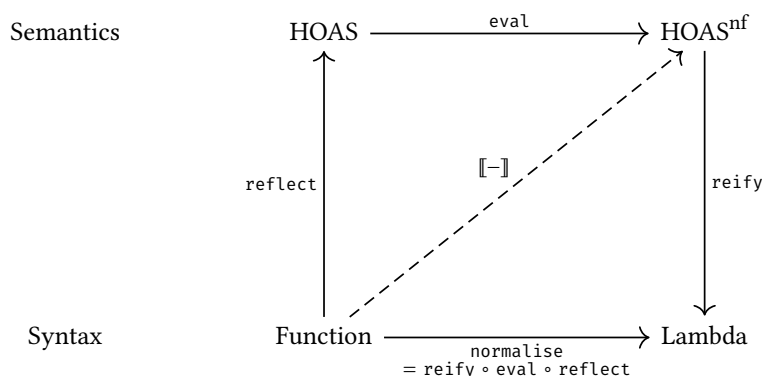


Fig. 5: Normalisation by evaluation for the `Function` datatype.

the code below is outdated

---

<sup>2</sup><https://www.scala-lang.org/files/archive/spec/2.13/06-expressions.html#anonymous-functions>

## 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))
}
}
```

## 0.1.5 Converting Functions back to Scalameta Terms

Surprise bitches same shit again, quasiquotes ftw

### 0.1.6 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. idk, 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