

## 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.

#### Normalisation by Rewriting

Essentially term-rewriting, similar to how Parsers are simplified. Strong reduction – allow beta reduction to occur deep inside lambda terms, in all redexes of a term, until no more reductions can be made. To beta normal form, not just beta-WHNF (weak head normal form)

Aside: Untyped lambda calculus is not strongly normalising, but these terms are converted from valid typed Scala programs. Although not formally proven, it is reasonable to assume that these terms are a subset of the simple lambda calculus that are strongly normalising – they won't be particularly complex.

`Function` has one binding construct – the  $n$ -ary abstraction. This introduces variables and the need for capture-avoiding substitution. Capture-avoiding substitution is hard, many approaches e.g. de Bruijn indices (but inefficient to open/close terms so much, so the correct choice is levels). \* There is a dizzying array of existing literature and techniques on to represent binders without names Attempted a barendregt's but the optimised form (not freshening on every substitution) might be subtly incorrect, not sure tbh.

---

<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.

### Normalisation by Evaluation

Fortunately, there is a cheat to completely circumvent the nightmare of capture-avoiding substitution. Unlike traditional normalisation techniques, NbE bypasses rewriting entirely, instead appealing to their denotational semantics. (fuck yeah bitches) Interpret the term into a denotational model which can then be evaluated, and reify the result back into a term. Use HOAS to leverage the meta-language (i.e. Scala) as our semantic model – blazingly fast! (theoretically lol) No rewriting = no worrying about avoiding capture. Shift that burden onto the scala compiler.

#### 0.1.3 Converting Scalameta Terms to the Function ADT

Three cases:

1. `Term.Function` nodes, representing lambda expressions
2. `Term.AnonymousFunction` nodes, representing lambda expressions using placeholder syntax
3. Any other term, which will be *eta*-expanded if possible

#### Lambda Expressions

Often found as relatively simple functions to glue together parsers, or transform the result of a parser:

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

Choose a particularly convoluted example:

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

$$\begin{array}{ccc} \text{Syntax} & \xrightleftharpoons[\text{reify}]{\llbracket - \rrbracket} & \text{Semantics} \\ & \text{normalise} = \text{reify} \circ \llbracket - \rrbracket & \end{array}$$

Fig. 3: Normalisation by evaluation in a semantic model.

In this example, 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.

Conversion “flattens” the lambda expression into a chain of n-ary abstractions, with the final term being the body of the lambda. Because we lose scoping information this way, perform  $\alpha$ -conversion on variables. The body becomes a `Translucent` term.

```
λ(_l1). λ(_l2, _l3). Translucent(_l2 + _l3, env = {_l1 → _l1, _l2 → _l2, _l3 → _l3})
```

Values in bold are Scalameta tree nodes, so the body term’s environment maps `Term.Name` nodes to their corresponding variable terms.

## Placeholder Syntax

Scala supports placeholder syntax for terseness, so the earlier parser can be rewritten as:

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

## Eta-Expansion

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

### 0.1.4 Normalising Function Terms

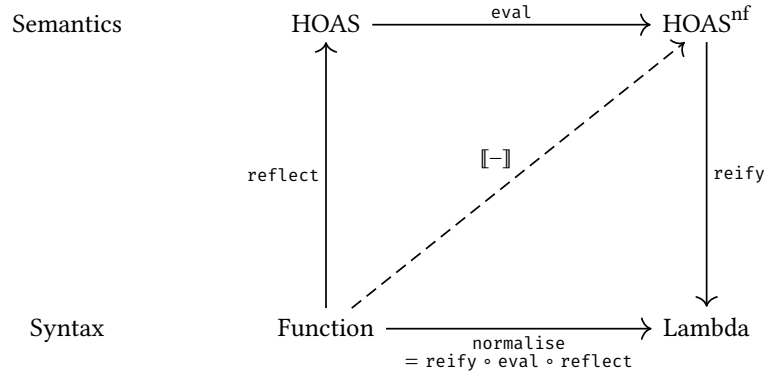


Fig. 4: Normalisation by evaluation for the `Function` datatype.

```
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.

## 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