

## 0.1 Function Representation

The work in ?? showed that it is possible to lift AST nodes corresponding to parser terms into a more statically inspectable `Parser` datatype. Furthermore, ?? demonstrated that this representation allows parsers to be transformed into a simpler form. However, so far, the functions passed to lifting combinators have been treated in an opaque manner, and have not been subject to the same simplifications as parsers. This is unsatisfactory, as these functions are essential to combining and sequencing parsers to build more complex parsers.

Additionally, 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. Looking back at the running example in 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)(ID))
```

It would be ideal to make functions inspectable in a similar manner to parsers, so that they can be *normalised* into a semantically equivalent but visually simpler form.

### 0.1.1 Motivation

Where are functions used in parsley? Lowest-level: `map` and `family`

### 0.1.2 Approach

It may be useful to view this as statically evaluating as much as possible – however, not all inputs are known so we cannot fully evaluate the result value. Normalisation is therefore just evaluation, but in the presence of unknown terms.

Assume functions passed to parsers are pure, because we assume sane users. So it is reasonable to model functions as a lambda calculus, and this can also be achieved with an ADT. As an aside, this is fine – Scala 3 has been formalised to the DOT calculus, and the subset that we are interested in is equivalent to the lambda calculus. In the simple lambda calculus all functions are curried i.e. each function only takes one argument, and multi-argument functions are represented as a chain of single-argument functions. In Scala, uncurried functions are preferred for performance reasons, but it allows currying as functions can have multiple parameter lists. We could desugar all this and curry the functions, but we want to be able to easily transform our output back into Scala code. So we extend the lambda calculus with n-ary abstraction and application.

```
trait Function

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

case object Identity extends Function
```

```
case object Flip extends Function
case object Compose extends Function
```

## 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). Attempted a barendregt's but the optimised form (not freshening on every substitution) might be subtly incorrect, not sure tbh.

## 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
```

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

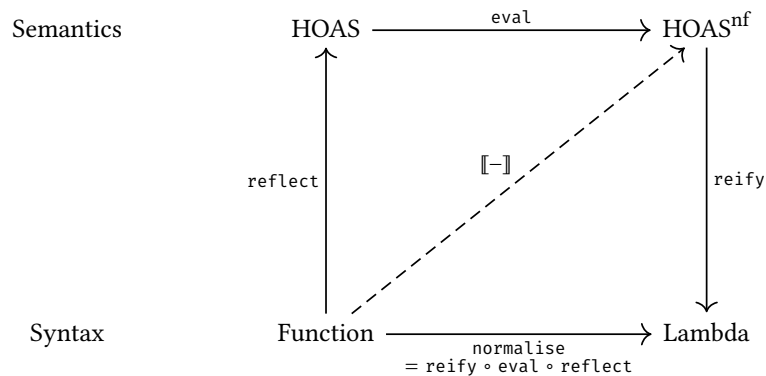


Fig. 1: Normalisation by evaluation for the `Function` datatype.

```
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.map { case (v, func) => v -> reflect0(func, 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 _ => 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.map { case (k, v) => k -> v.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.map { case (v, f) => v -> f.reify })
}
}
```

## 0.1.5 Converting Functions back to Scalameta Terms

Surprise bitches same shit again, quasiquotes ftw

## 0.1.6 Future Work?

Eta reduction – this is more complicated than in Haskell since Scala has special syntax

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