0.1 Function Representation

?? 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. ?? 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 ??:

```scala
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.

0.1.1 The *n*-ary Lambda Calculus

Once again, to avoid the complexity of manipulating raw Scalameta AST nodes, we can build an intermediate AST representation for function terms to make them easier to work with.

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 function terms that we want to normalise 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. Thus, our representation will be based on fig. 1, which extends the λ-calculus to support proper multi-argument functions using n-ary abstraction and application.

0.1.2 Normalisation

Deciding on a concrete ADT representation for the λ-calculus is a surprisingly complex problem. This section explores the different approaches to normalisation, and the effect this has on how the λ-calculus may be represented as a datatype. We first explain the traditional notion of reduction-based normalisation, before introducing normalisation by evaluation as a more elegant alternative.

Reduction-Based Normalisation

In the λ-calculus, terms are reduced in size by applying β-reduction: fig. 2 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

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

Fig. 1: Syntax for the untyped λ-calculus extended with n-ary abstraction and application.

arguments in $\overline{\mathbf{x}}$ are equal to the number of arguments in $\overline{\mathbf{N}}$; otherwise, the term is left as is.

This rule 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 (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.

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

Fig. 2: The β-reduction rule for the n-ary lambda calculus.

In fig. 2, 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]. 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. The example below shows how the simply-typed λ-calculus can be represented as a generalised algebraic data type (GADT) 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]


// 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 λ-calculus due to its inefficiency, as it requires traversing the whole 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, 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].

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 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. 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.

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 model. The denotational model (denoted by $[\![-]\!]$) 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. 3.

Evaluating Performance of Normalisation Strategies

0.1.3   Representing Functions

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

Facilitates more high-level parser simplifications

$$\text{Syntactic domain} \xrightarrow[\textit{reify}]{[\![-]\!]} \text{Semantic domain}$$
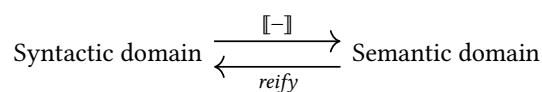
$$normalise = reify \circ [\![-]\!]$$

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

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]. Furthermore, once the defunctionalised functions have exhausted their usefulness in high-level parser simplifications, they also need to be lowered to λ-terms to be normalised.

The Function ADT

The syntactic representation of functions is thus represented as the uni-typed `Function` ADT in fig. 4. `Defunc` is the higher-level representation of defunctionalised higher-order functions, which are desugared into `Lambda` terms for normalisation. `Lambda` represents *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 – 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 λ-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[1]. 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.

0.1.4 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:

```scala
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. Take the following example to see why this is helpful:

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

---

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

```scala
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. 4: The `Function` ADT for representing functions.

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 λ-calculus expression:

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

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:

```scala
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[2] – 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 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:

```scala
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:

---

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

```
(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 can't be statically inspected any further and is just wrapped in a `Translucent` term.
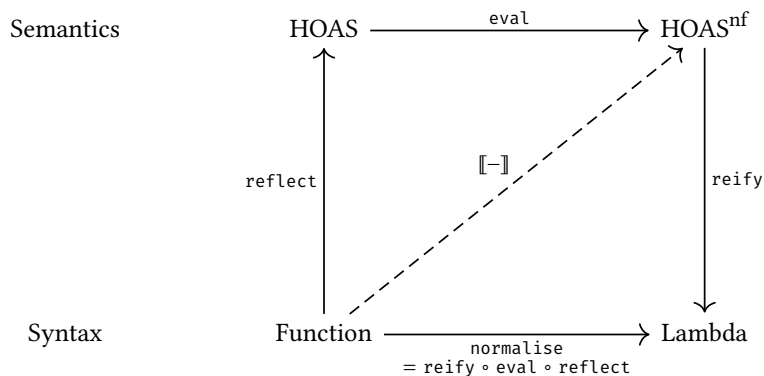
0.1.5 Normalising Function Terms

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

the code below is outdated

Reflection

```scala
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

```scala
trait HOAS {
  def eval: HOAS = this match {
    case HAbs(n, f) => HAbs(n, x => f(x).eval)
```

```scala
    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

```scala
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))
  }
}
```

0.1.6 Converting Functions back to Scalameta Terms

Surprise bitches same shit again, quasiquotes ftw

0.1.7 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 con-

crete 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