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 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 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. 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.

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

Alternative, newer approach that doesn't require any rewriting at all. Instead, evaluate In NbE, a term is first evaluated in some 'semantic' model of language. The resulting value is reified back into a term. This gives a practical method for deciding definitional equality even in complicated dependent type theories, where rewriting becomes prohibitively complicated.

0.1.2 Converting Scalameta Terms to the Function ADT

Eta-expansion.

0.1.3 Normalising Function Terms

$$\begin{tikzcd}
\text{Semantics} & \text{HOAS} \arrow[r, "\texttt{eval}"] & \text{HOAS}^{\text{nf}} \arrow[d, "\texttt{reify}"] \\
\text{Syntax} & \text{Function} \arrow[u, "\texttt{reflect}"] \arrow[ur, dashed, "[\![-]\!]"] \arrow[r, "\texttt{normalise}"', "= \texttt{reify} \circ \texttt{eval} \circ \texttt{reflect}"'] & \text{Lambda}
\end{tikzcd}$$

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

0.1.4 Future Work?

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

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.

* map, lift (implicit and explicit), zipped, (.as perhaps?) – these should surely boil down into two cases: (x, y).xxx(f) and xxx(f, x, y) * named function literals (val) * named method literals (def) * anonymous functions i.e. lambdas * functions with placeholder syntax * apply methods of case classes - symbol will tell its a class signature so we use this as a clue to look at synthetics??? * generic bridges – I reckon the information will probably show up in synthetics again

* 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