0.1 Representing and Simplifying Parsers

This is an INTERMEDIATE SYMBOLIC REPRESENTATION (?) more specialised than general-purpose scala ast
This section is about simplifying in our semantic domain (parsers)

Scalafix runs at the meta-level, outside of the phase distinction of compile- and run-time. Staged metaprogramming applies optimisations at compile-time, whereas these "optimisations" at applied post-compilation

For example, given two AST nodes `Term.Name("p")` and `Term.Name("q")` corresponding to named parsers `p` and `q`, suppose a transformation involves combining them with the *ap* combinator `<*>`. One may consider using quasiquotes to achieve this: `q"p <*> q"` would automatically expand to `Term.ApplyInfix(Term.Name("p"), Term.Name("<*>"), Type.ArgClause(Nil), Term.ArgClause(List(Term.Name("q")), None))`. However, this loses the static inspectability of the individual parsers `p` and `q` – although quasiquotes can be used as extractor patterns to recover the original AST nodes, their usage as such is discouraged as they can easily result in unintended match errors. The recommended approach is to pattern match on the AST nodes directly, which is obviously unergonomic even for this small example: to extract the RHS term `q`, one would have to perform a nested pattern match on the `Term.ApplyInfix` term and its `Term.ArgClause` node representing the arguments of the infix function application.

It is hopefully obvious that this would a very painful process for the rule author. It would be desirable to abstract away from the low-level syntactic AST representation, and instead treat these AST nodes as what they semantically represent – parsers.

Instead, fig. 1 shows how parser terms can be represented as an algebraic data type ADT, in the same way `parsley` itself uses a deep embedding to represent parsers as pure data objects. The reasoning behind this approach is the same as that for PARSLEY – this representation allows parsers to be easily inspected and analysed via pattern matching on constructors.

```scala
trait Parser
case class NonTerminal(ref: Symbol) extends Parser
case class Pure(f: Function) extends Parser
case object Empty extends Parser
case class Choice(p: Parser, q: Parser) extends Parser
case class Ap(p: Parser, q: Parser) extends Parser
...
```

Fig. 1: A subset of the core combinators in the `Parser` ADT.

This section explores the motivation behind this and the design choices made in the implementation. Use the left-recursion factoring (**??**) rule as a basis/context to demonstrate the utility of this representation.

Running example The left-recursion factoring rule (**??**) performs the most complex analyses and transformations on parsers in `parsley-garnish`. Thus, it is a good example to motivate the design requirements for the parser representation. The following left-recursive parser and its transformation into its `postfix` form will serve as a running example for this section:

```scala
lazy val expr: Parsley[String] = (expr, string("a")).zipped(_ + _) | string("b")
```

0.1.1 Detecting Named Parsers

Before any analysis on parsers can be performed, it is first necessary to identify which AST nodes correspond to parsers. `parsley-garnish` builds a map of all parsers defined within a source file, indexed by the unique symbol of its name.

Identifying these AST nodes of interest involves pattern matching on `val`, `var`, and `def` definitions with a type inferred to be some `Parsley[_]` – this information is accessed by querying the Scalafix semantic API for the node's symbol information. Consider the labelled `ast` structure of the `expr` parser:

```
Defn.Val(
  mods = List(Mod.Lazy()),
  pats = List(Pat.Var(Term.Name("expr"))),
  decltpe = Some(
    Type.Apply(Type.Name("Parsley"), Type.ArgClause(List(Type.Name("String"))))
  ),
  rhs = Term.ApplyInfix(...)
)
```

The qualified symbol `expr` is used as the key in the map, and the `rhs` term is lifted the intermediate parser representation for analysis. A reference to the original AST node is also kept so any lint diagnostics or code rewrites can be applied to the correct location in the source file. Thus, a full traversal through the source file builds a map of all named parsers, representing all non-terminals in the grammar defined within that file.

0.1.2 Converting Scalameta Terms to the Parser ADT

Having identified the AST nodes which represent parsers, they need to be transformed into the appropriate `Parser` representation. This involves pattern matching on the `scala.meta.Term` to determine which parser combinator it represents, and then constructing the appropriate `Parser` instance.

Each `Parser` defines a partial function, `fromTerm`, which creates an instance of that parser from the appropriate `scala.meta.Term`. These `fromTerm` methods are combined to define a `toParser` extension method on `scala.meta.Term` – this is where AST nodes are lifted to their corresponding `Parser` representation.

The top-level combinator that makes up `expr`'s definition is the choice combinator, `|`. Scalameta represents this infix application of the `|` operator as so:

```
Term.ApplyInfix(
  lhs = Term.Apply(...), // AST node for (expr, string("a")).zipped(_ + _)
  op = Term.Name("|"),
  targClause = Type.ArgClause(List()),
  argClause = Term.ArgClause(
    List(
      Term.Apply(
        Term.Name("string"),
        Term.ArgClause(List(Lit.String("b")), None)
      )
    ),
    None
  )
)
```

This structure therefore guides the implementation of the pattern match in `Choice.fromTerm`:

```
object Choice {
  val matcher = SymbolMatcher.normalized("parsley.Parsley.`|`", "parsley.Parsley.`<|>`")

  def fromTerm(implicit doc: SemanticDocument): PartialFunction[Term, Choice] = {
    case Term.ApplyInfix(p, matcher(_), _, Term.ArgClause(List(q), _)) =>
      Choice(p.toParser, q.toParser)
  }
}
```

The definition of this method is fairly self-explanatory: it matches on a `ApplyInfix` term where the operator is the `|` combinator, and recursively applies `toParser` to its LHS and RHS nodes. Finishing off, the `expr` parser is therefore converted to the following `Parser` instance:

```
Choice(
  Zipped(Function(_ + _), List(NonTerminal(expr), Str(a))),
  Str(b)
)
```

The exact representation of the `Function` is not important at this momenet – this is covered in the next ??. For brevity, the remaining code snippets in this section will simplify the function representations and continue to grey them out.

0.1.3 Building New Parsers From Existing Parsers

Now that raw AST terms can be lifted to the higher-level parser representation, it is easy to build new parsers from existing parsers. This is crucial for left-recursion factoring, which "unfolds" parsers into separate parsers representing the left-recursive and non-left-recursive parts. These are then recombined to form parsers which are free from left recursion.

Smart constructors are used to make manipulating parser terms resemble writing `parsley` code itself. These are defined as infix operators, which are written as extension methods on the `Parser` trait:

```
implicit class ParserOps(private val p: Parser) extends AnyVal {
  def <*>(q: Parser): Parser = Ap(p, q)
  def <|>(q: Parser): Parser = Choice(p, q)
  def map(f: Function): Parser = FMap(p, f)
}
```

Parser terms can now be manipulated in a manner that looks almost indistinguishable from writing `parsley` code. For example, the `unfold` method on the `Ap` parser contains this snippet, where `pl`, `ql`, and `q` are parsers (`pe` is not a parser, but rather an `Option` value):

```
val lefts = {
  val llr = pl.map(flip) <*> q
  val rlr = pe.map(f => ql.map(composeH(f))).getOrElse(Empty)
  llr <|> rlr
}
```

Other than the capitalised `Empty` constructor, this would be perfectly valid `parsley` code.

0.1.4 Simplifying Parsers Using Parser Laws

Recombining unfolded parsers during left-recursion factoring introduces many necessary, but extraneous "glue" combinators. Even though the transformed parser is semantically correct, it ends up very noisy syntactically. Consider the resulting parser from factoring out the left-recursion in `expr`:

```
lazy val expr: Parsley[String] = chain.postfix(
  empty | (empty.map(a => b => a + b) | empty <*> expr) <*> string("a")
    | string("b") | empty
)(
  (empty.map(flip) <*> expr | pure(identity).map(compose(a => b => a + b)))
    .map(flip) <*> string("a")
    | empty | empty
)
```

The intent of this parser is completely obfuscated – it would be unacceptable for the output of the transformation to be left in this form. For human readability, this parser term must be simplified as much as possible, using domain-specific knowledge about parser combinators. This is where the deep embedding approach comes to shine; simplifications are easily expressed by pattern matching on `Parser` constructors.

Willis, Wu, and Pickering [2020] note that parser combinators are subject to *parser laws*, which often form a natural simplification in one direction. In Haskell `parsley`, Willis [2023] uses these parser laws as the basis for high-level optimisations to simplify the structure of the combinator tree. `parsley-garnish` uses the same principles to simplify the parser term to become more human-readable. The two only differ in the purpose of the simplification: whereas Haskell `parsley` does this to produce an optimised AST to be compiled as code, `parsley-garnish` simplifies the parser AST to be pretty-printed as text.

Fig. 2 shows the subset of parser laws utilised by `parsley-garnish` for parser simplification. Most of the laws in fig. 2 have already been shown to hold for Parsley by Willis and Wu [2018]; an additional proof for eq. (8) can be found in ??.

$$\texttt{p.map(f).map(g) = p.map(g compose f)} \tag{1}$$
$$\texttt{pure(f) <*> pure(x) = pure(f(x))} \tag{2}$$
$$\texttt{pure(f) <*> x = x.map(f)} \tag{3}$$
$$\texttt{empty | u = u} \tag{4}$$
$$\texttt{u | empty = u} \tag{5}$$
$$\texttt{pure(x) | u = pure(x)} \tag{6}$$
$$\texttt{empty <*> u = empty} \tag{7}$$
$$\texttt{empty.map(f) = empty} \tag{8}$$

Fig. 2: Functor (1), Applicative (2, 3), and Alternative (4–8) laws.

In the previous example, it is evident that the most noise results from the `empty` combinators. These can be eliminated using eqs. (4), (5), (7), and (8):

```
lazy val expr: Parsley[String] = chain.postfix(string("b"))(
  (pure(identity).map(compose(a => b => a + b))).map(flip) <*> string("a")
)
```

The complicated term in the postfix operator can then be simplified as follows:

```
    (pure(identity).map(compose(a => b => a + b))).map(flip) <*> string("a")
=       { eqs. (2) and (3) }
    pure(compose(a => b => a + b)(identity)).map(flip) <*> string("a")
=       { eqs. (2) and (3) }
    pure(flip(compose(a => b => a + b)(identity))) <*> string("a")
```

= { eq. (3) }

```
string("a").map(flip(compose(a => b => a + b)(identity)))
```

This results in the most simplified form of the parser:

```scala
val f: Function = flip(compose(a => b => a + b)(identity))
lazy val expr: Parsley[String] = chain.postfix(string("b"))(string("a").map(f))
```

Encapsulating boilerplate Lawful simplifications are applied akin to peephole optimisations on the recursively defined `Parser` ADT. There are many instances of parsers, which inevitably leads to repetitive and error-prone boilerplate code which exists to simply recurse through each case. To avoid this, the recursive traversal itself is decoupled from the application of the transformation function. Although the traversal is still hand-written, the implementation is inspired by the generic traversal patterns offered by Haskell's `uniplate` library [Mitchell and Runciman 2007].

This is realised as a `transform` method on the `Parser` trait, which takes a partial function and applies it to nodes where it is defined. The transformation is applied via a bottom-up traversal:

```scala
def transform(pf: PartialFunction[Parser, Parser]): Parser = {
  val p = this match {
    case Ap(p, q)      => Ap(p.transform(pf), q.transform(pf))
    case Zipped(f, ps) => Zipped(f, ps.map(_.transform(pf)))
    case Pure(f)       => Pure(f)
    ...
  }
  if (pf.isDefinedAt(p)) pf(p) else p
}
```

A `rewrite` method can then be defined in terms of `transform`, applying the partial function everywhere and re-applying it until it no longer makes a change. This has the effect of applying a transformation exhaustively until a normal form is reached.

```scala
def rewrite(pf: PartialFunction[Parser, Parser]): Parser = {
  def pf0(p: Parser) = if (pf.isDefinedAt(p)) pf(p).rewrite(pf) else p
  this.transform(pf0)
}
```

Therefore, any transformation on parsers can be defined without having to worry about recursion boilerplate: the act of traversal itself is fully abstracted away and encapsulated within the `transform` method. Using `rewrite`, parser simplification can then be expressed in a clean and maintainable manner:

```scala
def simplify: Parser = this.rewrite {
  // p.map(f).map(g) == p.map(g compose f)
  case FMap(FMap(p, f), g) => FMap(p, composeH(g, f))
  // u <|> empty == u
  case Choice(u, Empty) => u
  // pure(f) <|> u == pure(f)
  case Choice(Pure(f), _) => Pure(f)
  ...
}
```

Further design considerations are made to ensure the extensibility and safety of this approach: the `Parser` trait is sealed, which enables compiler warnings if a new `Parser` case is added and the `transform` method is not updated.

Since the traversal is still written by hand rather than generically derived, it is still more prone to error The traversal could be generically derived rather than written by hand, but this would require the use of an external dependency such as `shapeless`[1], which is overkill for the complexity of the `Parser` ADT.

0.1.5 Converting Parsers Back to Scalameta Terms

After parsers have been transformed and simplified, the last step is to convert them back to a textual representation to be applied as a Scalafix patch. Parsers can be lowered back to `scala.meta.Term` nodes by the inverse of the original `fromTerm` transformation. The `Parser` trait defines this transformation as the method `term`, using quasiquotes to simplify the construction of the `scala.meta.Term` nodes.

```
case class Zipped(func: Function, parsers: List[Parser]) extends Parser {
  val term: Term = q"(..${parsers.map(_.term)}).zipped(${func.term})"
}
```

This term can then be pretty-printed into a string, and applied as a Scalafix patch.

Summary

[1] https://github.com/milessabin/shapeless