0.1 Simplifying Parsers

Reusing a similar abstract syntax representation as `parsley` itself unlocks some interesting insights for `parsley-garnish`. Gibbons and Wu [2014] note that a deep-embedded DSL consists of two components:

1. A representation of the language's abstract *syntax*, in the form of the aforementioned datatype.
2. Some traversals over the datatype, which gives *semantics* to that syntax.

A deep-embedded DSL and a linter for that DSL can thus be viewed as two different semantic interpretations over the same abstract syntax:

- The DSL semantics are *evaluation*. The syntactic structure may be transformed for optimisation purposes before generating code to be evaluated.
- The linter's semantics are two-fold for lint diagnostics and code rewrites:
  - *Emitting side-effects* in the form of diagnostics, based on patterns of interest within the syntactic structure.
  - *Pretty-printing* a transformation over the syntactic structure, as a rewrite action – crucially, unlike evaluation, the transformed output is not converted into code but rather a textual representation to be rewritten over the original source file. The output of this transformation may benefit from *the same optimisation transformations as with the DSL semantics* to simplify the pretty-printed textual output.

This section shows that this is indeed the case for `parsley-garnish`: the same optimisation transformations apply for both `parsley` (the DSL) and `parsley-garnish` (the linter). The only difference lies in the purpose of performing these transformations:

- `parsley-garnish` needs to perform simplifications on the `Parser` AST to produce output of hand-written quality, or else the resulting parser would be unreadable.
- `parsley` performs simplifications on its combinator tree to produce output of hand-written quality, in order to deliver excellent parser performance.

0.1.1 Parser Laws

Willis, Wu, and Pickering [2020] note that parser combinators are subject to *parser laws*, which often form a natural simplification in one direction. Both `parsley` Scala [Willis and Wu 2018] and `parsley` Haskell [Willis 2023] use these laws as the basis for high-level optimisations to simplify the structure of deeply-embedded parsers. These same principles are used by `parsley-garnish` to simplify parser terms to resemble the natural style that a human would write by hand.

Fig. 1 shows the subset of parser laws utilised by `parsley-garnish` for parser simplification. Most of these laws have already been shown to hold for Parsley by Willis and Wu [2018]; an additional proof for eq. (8) can be found in **??**.

Simplifying the Example Parser

It is useful to illustrate how these laws are used to simplify a parser term, by starting with the parser in **??**. First of all, most of the noise in `example` comes from the large number of `empty` combinators. These can be eliminated using eqs. (4), (5), (7), and (8):

```
p.map(f).map(g) = p.map(g compose f)                    (1)
  pure(f) <*> pure(x) = pure(f(x))                      (2)
      pure(f) <*> x = x.map(f)                          (3)
            empty | u = u                               (4)
            u | empty = u                               (5)
        pure(x) | u = pure(x)                           (6)
         empty <*> u = empty                            (7)
        empty.map(f) = empty                            (8)
```

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

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

This already looks a lot better, but the second parameter to `postfix` can be further simplified as follows:

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

The final simplified form of the parser is then:

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

The parser is now expressed in a much simplified form, in a similar style to how it would be written by hand. The remaining challenge is to simplify the contents of the expression `f`, which is tackled in **??**.

0.1.2 Implementing Rewrites on the Parser AST

Lawful simplifications are applied by a bottom-up transformation over the recursively defined `Parser` AST. Since there are many parser cases, this inevitably leads to repetitive and error-prone boilerplate code which simply exists to recursively propagate the transformation through each case. To avoid this, the recursive traversal itself can be decoupled from the definition of the transformation function. Although the traversal is still hand-written, this implementation is inspired by the generic traversal patterns offered by Haskell's `uniplate` library [Mitchell and Runciman 2007].

The traversal 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:

```
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.

```
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:

```
def simplify: Parser = this.rewrite {
  case FMap(FMap(p, f), g) => FMap(p, compose(g, f))
  case Pure(f) <*> Pure(x) => Pure(App(f, x))
  case u <|> Empty => u
  case Pure(f) <|> _ => Pure(f)
  ...
}
```

Extensibility and Safety Further design considerations are made to ensure the extensibility 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. Although this formulation of the traversal is inspired by generic traversals, it still manually defines the traversal for each case: a safer approach would be to generically derive this. In Scala, this would require the use of an external dependency such as `shapeless`[1], which is frankly overkill given the relative simplicity of the `Parser` ADT.

0.1.3 Discussion

The design of the parser simplification process for `parsley-garnish` was not intended to closely follow the methods used in `parsley`, so it is remarkable that the two approaches have ended up being so similar. However, in retrospect, this resemblance is not surprising given that the act of parser simplification and optimisation are fundamentally the same transformation. Since both `parsley` and `parsley-garnish` represent the parser AST as a deep-embedded structure, it is natural that this transformation is implemented similarly in both cases as a bottom-up traversal over the abstract syntax.

This insight can be extended to any deep embedded DSL based on an algebra, where constructs within the DSL are subject to algebraic laws and operations. It would be interesting to see an EDSL and linter pair that shares a unified data structure for its abstract syntax, in order to take full advantage of this duality.

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