Chapter 1
Evaluation

The project was initially intended to mostly focus on implementing a variety of parser lints, however implementing these rules revealed an unanticipated level of complexity in the infrastructure needed to support them. This ended up necessitating a lot more work on the intermediate `Parser` and `Expr` ASTs than expected, which has led to a robust internal machinery that allows `parsley-garnish` to be easily extended with more domain-specific rules in the future. However, the time spent on this infrastructure work meant that the variety of implemented linting rules were not as comprehensive as initially desired. The vast majority of work was spent on the left-recursion transformation, which motivated much of the work on the intermediate ASTs. Therefore, evaluating the outputs of the left-recursion transformation also evaluates the success of the intermediate machinery.

This chapter evaluates the overall project by assessing the performance of its expression normalisation infrastructure, which represents some of the most complex and critical parts of the system. Then, it evaluates the success of the left-recursion transformation by examining its ability to handle different forms of left-recursion.

1.0.1 Performance of Expression Normalisation

Originally, the implementation of `Expr` used a named approach with Barendregt's convention, generating fresh variable names using an atomic counter. However, this required an extra α-conversion pass to clean up variable names before pretty-printing the expression term, since fresh name generation resulted in undesirably long names like `x4657476`.

The implementation was then switched to a HOAS approach, which made the correctness of the normalisation process more easy to reason about. However, this lost the ability to easily show α-equivalence of terms, so the next logical step was to implement the final two-tiered NBE approach, where a syntactic representation allowed for easy α-equivalence checking, and a semantic HOAS-based representation was used for normalisation.

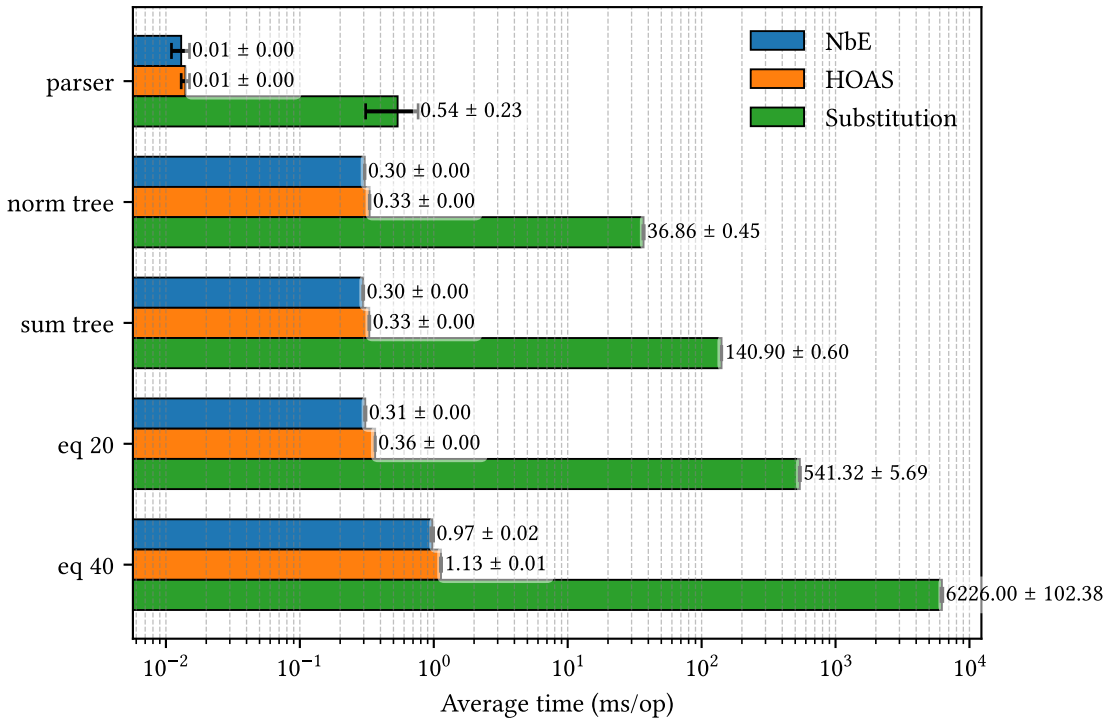

Fig. 1.1: Benchmark results for the expression normalisation process.

1.1 Removing Left-Recursion

Broadly speaking, left-recursive grammars can be classified into three categories: direct, indirect, and hidden. The left-recursive transformation implemented by `parsley-garnish` is able to handle all three of these cases, however

this does not necessarily mean that the output is of high quality. This section evaluates parsley-garnish's ability to handle each case, based on a mostly qualitative set of evaluation criteria:

- Was the instance of left-recursion detected?
- If an auto-fix was performed, was it correct? Can it be proven to be correct?
- How clear was the output? How does it compare to an idiomatic, manually fixed version?
- Does the output compile?

The following examples assume the existence of a number parser, defined the same way as earlier in **??**:

```
val number: Parsley[Int] = digit.foldLeft1(0)((n, d) => n * 10 + d.asDigit)
```

1.1.1 Direct Left-Recursion

Direct left-recursion is the simplest and most obvious form of left-recursion, where a parser directly refers to itself in its definition. Thus, it is the easiest form of left-recursion to detect and handle, regardless of the transformation technique used. This section evaluates parsley-garnish's handling of direct left-recursion in a few different scenarios.

1.1.1.1 Unary Postfix Operator

The following minimal grammar for a postfix unary incrementing operator is directly left-recursive:

$$\langle inc \rangle ::= \langle inc \rangle \text{ `+'} \mid \langle number \rangle$$

A parser for this grammar, written in left-recursive form, would be as follows:

```
enum Expr {
  case Num(n: Int)
  case Inc(x: Expr)
}
val incs: Parsley[Expr] = Inc.lift(incs) <~ "+" | Num.lift(number)
```

In this instance, parsley-garnish detects the left-recursion and transforms the parser into the form shown below; this is compared with a hand-written version.

```
// Transformed by parsley-garnish
val incs = chain.postfix[Expr](number.map(x1 => Num(x1)))
                               (string("+").map(x1 => x2 => Inc(x2)))
// "Optimal" version written by hand
val incsByHand = chain.postfix[Expr](number.map(Num(_)))("+" as Inc)
```

The output from parsley-garnish manages to compile and is clear and idiomatic, although the hand-written version is slightly more concise. These two versions can be shown to be equivalent via equational reasoning – the first argument to postfix is obviously the same, but with placeholder syntax instead of using explicit lambda arguments. The second argument can be derived as follows:

```
string("+").map(x1 => x2 => Inc(x2))
```

$=$ { Definition of as: `p.as(x) = p.map(_ => x)` }

```
string("+") as (x2 => Inc(x2))
```

$=$ { η-reduction on `Inc` }

```
string("+") as Inc
```

$=$ { Re-introduce *Implicit Conversions* pattern }

```
"+" as Inc
```

This example highlights some subtle points that parsley-garnish considers in order to improve the likelihood of producing compilable output:

- The type ascription `chain.postfix[Expr]` is not always necessary, although in this case it is actually required to help Scala correctly unify the types expected by the combinator. Since parsley-garnish cannot typecheck its outputs, it always includes this type ascription to boost Scala's type inference.
- The original parser used the *Implicit Conversions* pattern to elide the `string` combinator, but parsley-garnish re-introduces the explicit `string` combinator in its output. This is intentional – Scala 2 has trouble with implicit conversions in certain positions, such as in this case where `"+".map(...)` would not compile. parsley-garnish attempts to add the explicit combinator back in cases like this, although in general it will respect the original style the parser was written in.

1.1.1.2 Arithmetic Expression Language

As a larger-scale example, the left-associative arithmetic operators from **??** are also defined in a directly left-recursive manner. Recall how the hand-written version using `chain.left1` was presented:

```
lazy val expr: Parsley[Expr] = chain.left1(term)('+' as Add(_, _) | '-' as Sub(_, _))
```

Compared with the full transformed output from parsley-garnish:

```
lazy val expr = chain.postfix[Expr](term)(
  ('+' ~> term).map(x1 => x2 => Add(x2, x1)) | ('-' ~> term).map(x1 => x2 => Sub(x2, x1)))
lazy val term = chain.postfix[Expr](atom)(
  ('*' ~> atom).map(x1 => x2 => Mul(x2, x1)) | ('/' ~> atom).map(x1 => x2 => Div(x2, x1)))
lazy val atom = '(' ~> expr <~ ')' | number.map(Num(_))
```

This also manages to successfully compile, and is relatively clear to read. However, it does highlight a current shortcoming of the left-recursion rule: left-recursive parsers can only be transformed into the most generalised `postfix` form, which may not always be the most optimal choice of combinator. In this case, the `chain.left1` combinator would've been a more appropriate choice; even better would be the `precedence` combinator, since the grammar forms multiple layers of expression operators.

It is theoretically possible to automatically transform a `postfix` parser into one of its more specialised brethren, but this would require a more sophisticated ability to factor out common patterns in parsers and expressions. Willis [2024] demonstrates how `chain.left1` can be defined in terms of `postfix`, presented in Haskell as it illustrates the relationship more clearly. The only major syntactical difference of note is that Haskell uses `<$>` for the `map` combinator:

```
chainl1 p op  =  postfix p (flip <$> op <*> p)
```

Using this definition, the `postfix` version can be provably shown to be equivalent to the `chain.left1` version. This proof will again use Haskell syntax as it is more suitable for equational reasoning. The goal is to show that `('+' ~> term).map(x1 => x2 => Add(x2, x1))` can be rewritten in the form `op.map(flip) <*> p`, where `p` is `term` and `op` is shown to be `'+' as Add(_, _)`.

```
    (λx1 x2 → Add x2 x1) <$> ('+' *> term)
=     { Definition of flip }
    flip Add <$> ('+' *> term)
=     { Definition of *> (equivalent to ~> in Scala): p *> q = const id <$> p <*> q }
    flip Add <$> (const id <$> '+' <*> term)
=     { Re-association: u <*> (v <*> w) = pure (∘) <*> u <*> v <*> w }
    pure (∘) <*> flip Add <$> (const id <$> '+') <*> term
=     { Applicative fusion: pure f <*> pure x = pure (f x) }
    (flip Add .) <$> (const id <$> '+') <*> term
=     { Functor composition: fmap f ∘ fmap g = fmap (f ∘ g) }
```

```
    ((flip Add .) . const id) <$> '+' <*> term
=     { Point-free manipulation }
    (flip . const Add) <$> '+' <*> term
=     { Functor composition, in reverse }
    flip <$> (const Add <$> '+') <*> term
=     { Definition of $> (equivalent to .as in Scala) }
    flip <$> ('+' $> Add) <*> term
```

The same proof can be applied to the other operators, and the `map` combinator distributes over choice:

```
(u | v).map(f) = u.map(f) | v.map(f)
```

Therefore, the proof shows that the final term obtained from `parsley-garnish` can be rewritten in the form `op.map(flip) <*> p`, so the obtained `postfix` parser is equivalent to the hand-written version using `chain.left1`.

1.1.1.3 Evaluating the Arithmetic Expression Language

The evaluating parser variant of the same grammar, as presented in the introduction, has the same resulting form:

```
lazy val expr: Parsley[Float] = chain.postfix[Float](term)(
  ('+' ~> term).map(x1 => x2 => x2 + x1) | ('-' ~> term).map(x1 => x2 => x2 - x1))
```

However, in this case, the output unfortunately fails to compile. Scala's local type inference prevents the compiler from inferring the types of the `x2` parameters, as the arithmetic operators used in the lambdas are overloaded. This problem does not occur for the previous example, since the `Expr` constructors are explicitly and unambiguously typed. Users will have to manually fix `parsley-garnish`'s output by adding an explicit type annotation (`x2: Float`).

Resolving this issue is future work: the most likely solution is to refactor the parser so that they do not take curried functions, which would make it easier for Scala to infer their types. For example, if the parser was refactored into the `chain.left1` form utilising the `as` combinator, the addition function `_ + _` would be in a fully uncurried form.

Summary Direct left-recursion is the most straightforward form of left-recursion to detect, so it is unsurprising that `parsley-garnish` handles it well. It is important that this case is handled well, however, since it is generally the most common form of left-recursion. The transformation on these test examples are provably correct, and the resulting parsers are relatively clear and idiomatic. `parsley-garnish` also takes care to improve the likelihood of producing compilable output, although there is still some future work to be done in this area. The most significant weakness is the inability to specialise the `postfix` parser into a more specific form, however this is not a critical issue as the `postfix` form is still correct and idiomatic.

1.1.2 Indirect Left-Recursion

Instances of *indirect* left-recursion are harder to detect, since the parser's reference to itself takes more than one step to reach. Consider the following alternative grammar for arithmetic expressions, reduced to only addition for simplicity:

$$\langle expr \rangle ::= \langle add \rangle \mid \text{`(`} \langle expr \rangle \text{`)`} \mid \langle number \rangle$$
$$\langle add \rangle ::= \langle expr \rangle \text{ `+` } \langle expr \rangle$$

The indirect left-recursive cycle arises since `expr` firstly needs to parse `add`, which firstly needs to parse `expr`, and so on. This grammar can be naïvely translated in its left-recursive form in the following code segment. For variety, this example utilises the *Parser Bridges* pattern instead of using `lift` combinators:

```
enum Expr {
  case Num(n: Int)
  object Num extends ParserBridge1[Int, Num]
  case Add(x: Expr, y: Expr)
```

```
  object Add extends ParserBridge2[Expr, Expr, Add]
}
lazy val expr: Parsley[Expr] = add | '(' ~> expr <~ ')' | Num(number)
lazy val add: Parsley[Expr] = Add(expr, '+' ~> expr)
```

The indirect left-recursion is successfully detected by `parsley-garnish`, and it is able to offer an automated fix. For brevity, the parser type annotations will be omitted in subsequent examples, as they are not changed by `parsley-garnish`.

```
lazy val expr = chain.postfix[Expr]('(' ~> expr <~ ')' | number.map(x1 => Num(x1)))
                                   (('+' ~> expr).map(x1 => x2 => Add(x2, x1)))
lazy val add = Add(expr, '+' ~> expr) // unchanged and no longer referenced by expr
```

Although the output compiles and is functionally correct, there are a few areas for improvement:

- The definition of `expr` becomes somewhat cluttered, as the transformation operates by recursively visiting non-terminals and inlining their transformed definitions if they are left-recursive. In this example, the `add` parser was originally mutually left-recursive with `expr`, so it was inlined into `expr`. A possible solution to address this is to separate out these inlined parsers into new variables – they cannot overwrite the original definitions, as they may still be required by other parsers.
- Use of the parser bridge constructor for `Num` is not preserved – `parsley-garnish` was unable to resugar this back into its original form, instead leaving it in a more syntactically noisy form with `map`.

1.1.3 Hidden Left-Recursion

The final class of left-recursive grammars is *hidden* left-recursion, which in general are the most challenging to detect and handle. Hidden left-recursion occurs when a parser invokes itself after invoking other parsers that have not consumed any input. They generally have a form similar to the following:

$$\langle a \rangle ::= \langle b \rangle \, \langle a \rangle \, \cdots$$
$$\langle b \rangle ::= \epsilon$$

In this example, $\langle b \rangle$ is able to derive the empty string, so $\langle a \rangle$ is able to parse $\langle b \rangle$ without consuming input and then invoke itself, creating a left-recursive cycle. The following rather contrived example showcases a hidden left-recursive cycle in `a`, and the warning that `parsley-garnish` issues:

```
lazy val a: Parsley[Int] = b ~> a
// ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
// error: [FactorLeftRecursion] Left-recursion detected, but could not be removed from a.
// The resulting chain would be given a parser which consumes no input,
// causing it to loop indefinitely:
// chain.postfix[Int](some(digit) ~> a)(pure(x1 => x1))

lazy val b: Parsley[Int] = many(digit).map(_.mkString.toInt)
```

This parser is an instance of hidden left-recursion, since `many` repeatedly parses its given parser *zero* or more times, allowing it to succeed without consuming any input. `parsley-garnish` can detect cases of hidden left-recursion as long as it encounters combinators that it recognises, because as part of the unfolding transformation it must know about the empty-deriving semantics of parsers. However, it is unable to provide an automated fix since the `postfix` combinator encodes the behaviour of associative operators, which is not the case here.

Most cases of hidden left-recursion likely require manual intervention to fix, as it is unclear what the intended behaviour of the parser is. Therefore, although `parsley-garnish` is able to detect hidden left-recursion, it is unable to suggest a fix in any case. The mechanism in which it catches these cases is when the `leftRec` portion of the unfolded parser simplifies to `pure(x)`, which `parsley-garnish` rejects because it would cause an infinite loop. This can be seen in the error message, although in this context it is not particularly helpful as it leaks how `parsley-garnish` has desugared the parser while attempting to unfold it – for example, the `some` results from unfolding the `many` combinator. Future work could see improvements in the error messages to make them more helpful and informative in the case of hidden left-recursion.

Summary

The evaluation in this section shows that `parsley-garnish` is able to detect and handle all forms of left-recursion, although the quality of the output varies depending on the nature of the left-recursion. Revisiting the evaluation criteria:

- All classes of left-recursion can be detected, although hidden left-recursion cannot be automatically fixed. However, this is mostly due to the difficult nature and inherent ambiguity of grammars with hidden left-recursion.
- In all tested examples, the automated fixes were correct – some of these have been proven by equational reasoning. A natural extension of this work would be to formally prove the correctness of the transformation in the general case.
- The transformed output is generally clear and idiomatic, although there are areas for improvement in the preservation of syntactic sugar. A desirable next step would be to refactor `postfix` parsers into more specialised forms if possible for the given parser.
- In most tested cases, the transformed output is able to compile. However, since `parsley-garnish` tends to prefer creating curried functions, more complex parsers may require manual intervention to fix type inference issues. This is one of the major limitations of the current implementation, as it is undesirable for linters to suggest non-compiling fixes.