Imperial College
London

Parsley

Optimising and Improving Parser Combinators

Jamie Hyde Willis

February 26, 2024

A dissertation submitted in fulfilment of the requirements for
the degree of Doctor of Philosophy

in the

Department of Computing
Imperial College London

Abstract

Parser combinators are a functional abstraction for parsing that abstracts hand-written recursive-descent parsers behind a high-level set of combinators. While these kinds of parsers are popular in the functional programming community, they have been historically criticised:

1. Parser combinator performance is sub-par compared with handwritten parsers.
2. The high-level grammar is obscured by the combinators compared with parser generators.
3. The error messages generated by parser combinators are not of fantastic quality.

This dissertation addresses each of these complaints with the work split across two libraries, both called `parsley`: one in Haskell and the other in Scala. Within these libraries, different issues are tackled.

Haskell `parsley` addresses the performance concerns of parser combinators by modeling them as a strongly-typed deep embedding, allowing for optimisations and analysis to be performed. To eliminate the overheads of interpretation and achieve high performance, `parsley` makes use of staged metaprogramming to convert the continuation-passing style automaton into Haskell code resembling hand-written recursive descent parsers; this is faster than contemporary parser combinator libraries in Haskell.

Writing parsers is often an ad-hoc exercise; this dissertation introduces parser combinator design patterns that help structure and standardise how these parsers should be written. These patterns focus on a handful of issues: cleanly handling precedence hierarchies and expression parsing; organising and distinguishing between low-level tokens and higher-level parsing; and abstracting away semantic actions and meta-data processing. This helps to make clean, maintainable, parsers.

Finally, Scala `parsley` has focused on improving the design of parser combinator error systems. The design of this improved system is explored as well as how it can be implemented efficiently, minimising impact on the "happy path." This gives rise to more parsing patterns as well as enriching existing patterns to incorporate errors. The new patterns help provide the tools to build bespoke, descriptive, errors.

Statement of Originality

Except when referenced by citation, I declare that all the work within this dissertation is my own work. Any work done in collaboration with colleagues is indicated, and the source declared, at the top of a chapter or section when applicable. The views expressed within are my own.

Jamie Hyde Willis

February 26, 2024

License

Dedicated to my grandpa, Robin Francis Rissone, from whom my love of engineering so clearly stems, despite us sadly never having met.

Acknowledgements

In the summer of 2016 my supervisor, Nicolas Wu, took me on board for an internship. He was absolutely determined to get me to love Haskell, and introduced me to Scala and parser combinators as a way of getting me on board; the influence this had on my life from that point cannot be overstated. This dissertation makes it crystal clear quite how much he succeeded (even if Scala is still my favourite: sorry!), and I will be forever grateful that he took me under his wing. Nick, thanks for all the support, guidance, and growth you've given me as both a researcher and a teacher over the last almost decade we've known each other, not to mention the tears of laughter; even if you sometimes forget my name isn't Jeremy. You're the best supervisor I could have asked for.

I also owe much to Matthew Pickering, who taught me everything I know about staged metaprogramming and helped open my eyes to a whole new way of thinking: without him, I'm not even sure what this project would have looked like. Matt, thanks for all the discussions we had, and helping me settle into the PhD life so well. To that end, I'd also like to thank Csongor and David, my other office mates, for making the office a fun place to work; I'm happy to have had you all around.

I'd like to thank all my colleagues, those with whom I worked directly, like Tom Schrijvers, and those who worked with me in the shadows: the ever-anonymous reviewers. The feedback on the drafts of my papers over the years has been instrumental in making me the writer I am today. I'd like to thank my examiners, Tony Field and Graham Hutton, specifically, for taking the time to read this: your comments were very helpful, and the viva was great fun. I would also like to thank Mark and Konstantinos too, for helping me find my place at Imperial: both of you have given me fantastic opportunities and made my time at Imperial so much more fun. I need to also thank all the undergraduates in the 19/20, 20/21, 21/22, and 22/23 second-year cohorts; without the leap of faith many of you took in using Scala `parsley` (even when it had next to no documentation or Maven releases!), I wouldn't have had the opportunity to really think deeply about how we should write parsers and make so many improvements. I can't forget those who chose me to supervise their final-year projects, either: it was a lot of fun being able to work with you all, and I'm glad you chose to do your projects with me and parser combinators.

Of course, I cannot forget my family and friends, who have been with me the entire journey. In particular, I need to thank my Mum, Dad, little sister Grace, and my step-mum Sophie for their love and support, as well as my grandma, Tessa, who encouraged me to come to London in the first place: I hope I've made you all proud. To my best friends Sam and Beth, thanks for always being there and cheering me on, your support has been invaluable. And of course, all my other friends too, for standing by my side, encouraging me, and keeping me grounded. I also want to thank Cosmin, who introduced me to Dark Souls, and served as my co-pilot: who would have guessed I needed *another* challenge on top of the PhD I'd already taken on?

Along the way, there are many people who have made an enormous impact on my life's trajectory, too many to mention. One of the most impactful, however, is Stéphanie Fleet, my AS-level French teacher and personal tutor. While I had been programming for a few years, somehow I hadn't put two-and-two together yet and realise it was something I could pursue as an actual career path – silly me; it was Steph who pointed that out to me and set me on the path I am on today. She was also a fantastic teacher, and has helped shape my own teaching today. Thank you so much: I wouldn't be where I am today without your gentle nudge.

Finally, I'd like to acknowledge Csongor's hat, for keeping it fresh.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In Computer Science, parsing is the process of extracting structure, described by some grammar, out of flat unstructured data, often text; a parser is an implementation of a parsing algorithm for a specific grammar. Parsers find practical use across a variety of different fields and disciplines, and are a core part of the architecture of most compilers and interpreters. The result of parsers, often referred to as its *semantic action*, can vary depending on the domain: for a compiler, this would be an *abstract syntax tree* (AST); for a light-weight data storage format like JSON, it would an object in some host language; for an interpreter or a calculator, it might be a direct evaluation into some result.

Parsers are usually constructed in one of two ways: either by constructing by-hand, often by *recursive descent*; or via some parsing framework. Parsing frameworks are general-purpose tools for writing parsers and are designed to abstract some or all the details of the underlying parsing task from the parser writer. Generally speaking, there are two kinds of parsing frameworks: the parser generator [Parr 2013; Deremer 1969a; Gill and Marlow 1995], a *domain-specific language* (DSL) [Fowler 2010] often taking the form of a high-level grammar, which can be compiled to produce the code for a parser in a general purpose language; and parser combinators [Swierstra 2009; Hutton 1992; Leijen and Meijer 2001], a functional-style *embedded* DSL [Hudak 1996] where the parser is constructed directly in the desired language using higher-order combinators to combine them.

There are many different flavours of parser combinator library, with varying underlying semantics, APIs, and performance characteristics. This ranges from fully non-deterministic implementations to industrial-strength continuation-passing style implementations. This dissertation advocates for the use of them for writing parsers.

Why Parser Combinators?

This dissertation is concerned with the real-world libraries such as those in the `parsec` [Leijen and Meijer 2001] family of libraries, with the following benefits compared with parser generators:

1. **Integrated** The source of the final parser is written as part of the larger program without pre-processing; this means that there are no extra build steps, and IDE integration is trivial.

2. **Flexible** By writing the parser in a high-level host language, parser combinators can:

   - factor out common, repeated, code.
   - leverage modularity and re-use parser fragments.

```
⟨prog⟩  ::= (⟨func⟩ ';')* ⟨expr⟩
⟨func⟩  ::= 'def' ⟨ident⟩ '(' ⟨ident⟩ (',' ⟨ident⟩)* ')' '=' ⟨expr⟩
⟨expr⟩  ::= ⟨ident⟩ | 'call' ⟨ident⟩ '(' ⟨expr⟩ (',' ⟨expr⟩) ')'
⟨ident⟩ ::= ('a' .. 'z')+
```

(a) The grammar in EBNF

```
data Prog = Prog [Func] Expr
data Func = Func String [String] Expr
data Expr = Ident String
          | Call String [Expr]
```

(b) The Haskell datatype to parse into.

Fig. 1.1: The grammar and AST for a Lambda Calculus-esque language.

- interact directly with desired semantic actions in a type-safe way.
- construct configurable parser fragments without limitations on how they can be used.

3. **Extensible** Often, missing functionality in a library can be implemented by the user, either as a composite combinator, or as new primitives if this does not require editing the underlying implementation type.
4. **Powerful** Most parser combinator libraries are capable of parsing context-sensitive grammars intuitively, with monadic parser combinators capable of generating new parsers on-the-fly.
5. **Transparent** Many parser combinator libraries operate exactly as written, meaning that their behaviour is easier to debug as few, if any, modifications are made to the operation of the parser.

Fig. 1.1a gives the definition of a more syntactically verbose form of the untyped Lambda Calculus, which has support for function definitions and expressions that may only call functions or return identifiers. To demonstrate some of the benefits of parser combinators in practice, implementations of this small language will be given using both a parser generator library, `Happy` [Gill and Marlow 1995], and parser combinators, parsing into the datatype given in fig. 1.1b.

Parser generators (`Happy`) The parser generator implementation of the grammar assumes the pre-existence of a lexer, that can produce a list of Tokens, which is what is referred to within the parser, via the `%token` mapping:

```
%name prog                       │ prog :: { Prog }
%tokentype   { Token }           │ prog  : funcs expr                      { Prog $1 $2 }
%token def   { TokenDef }        │ funcs : func ';' funcs                  { $1 : $3 }
       ident { TokenIdent $$ }   │       | {- empty -}                     { [] }
       call  { TokenCall }       │ func  : def ident '(' args ')' '=' expr { Func $2 $3 $6 }
       ';'   { TokenSemi }       │ args  : arg ',' args                    { $1 : $3 }
       ','   { TokenComma }      │       | arg                             { [$1] }
       '('   { TokenOpen }       │ expr  : ident                           { Ident $1 }
       ')'   { TokenClose }      │       | call ident '(' exprs ')'        { Call $2 $4 }
       '='   { TokenIs }         │ exprs : expr ',' exprs                  { $1 : $3 }
%%                               │       | expr                            { [$1] }
```

On the left is the start of the parser, which does the token description, giving names that can be used within the parser to each of the underlying tokens. On the right is the parser definition that follows: each line corresponds to a rule, and each has a pair of braces that describes how the components, numbered in order of appearance, should be combined into a semantic action. In this case, `Happy` does not have a way of describing the "zero or more" BNF extension, so funcs, args, and exprs manually encode separated lists. Other than this, it retains a very similar appearance to the original grammar in fig. 1.1a. Since this is not legal Haskell code by itself, it must first be compiled using `Happy` into a Haskell file that can then be referred to in the rest of the program.

Parser combinators The parser combinator implementation (given in a library with a similar style to `parsec` [Leijen and Meijer 2001]), on the other hand, does not assume the existence of a lexer. In fact, lexing and parsing are rarely decoupled by parser combinator implementations: lexers are built using the same underlying machinery. As such, the whitespace handling in this parser is explicit; a first attempt at the parser is:

```
ws = skipMany (satisfy isSpace)

prog  :: Parser Prog
prog  = Prog <$> many (func <* (string ";" *> ws)) <*> expr

func  = Func <$> ((string "def" *> ws) *> ident)
             <*> ((string "(" *> ws) *> args <* (string ")" *> ws))
             <*> ((string "=" *> ws) *> expr)
args  = (:) <$> arg <*> many ((string "," *> ws) *> arg)

expr  = Ident <$> ident
      <|> Call  <$> ((string "call" *> ws) *> ident) <*> ((string "(" *> ws) *> exprs <* (string ")" *> ws))
exprs = (:) <$> expr <*> many ((string "," *> ws) *> expr)

ident = some (oneOf ['a'..'z'])
```

This parser is a bit less clear, though this will be improved shortly, however the main differences are that it is written in plain Haskell (i.e., it is **integrated**), and it interleaves the semantic actions into the parser itself. Notice that the Haskell *arithmetic sequence* syntax [Marlow et al. 2010], ['a'..'z'], has been used to give a concise description of ident. It is worth explaining how to read this parser properly, but first it should be cleaned up a little. Notice that the pattern (string tok *> ws) appears in many places within this parser; since this is plain Haskell, this could be factored into its own function, called tok:

```
tok   :: String → Parser ()
tok t = string t *> ws where ws = skipMany (satisfy isSpace)
```

Substituting this function into the parser makes the definition much more readable:

```
prog  = Prog <$> many (func <* tok ";") <*> expr

func  = Func <$> (tok "def" *> ident) <*> (tok "(" *> args <* tok ")") <*> (tok "=" *> expr)
args  = (:) <$> arg <*> many (tok "," *> arg)

expr  = Ident <$> ident <|> Call <$> (tok "call" *> ident) <*> (tok "(" *> exprs <* tok ")")
exprs = (:) <$> expr <*> many (tok "," *> expr)

ident = some (oneOf ['a'..'z'])
```

It is now easier to see the underlying structure: the (*>) and (<*) *combinators* take two parsers and sequence them together, ignoring the result of the parser not being "pointed" to; semantic actions are applied to parsers by using (<$>) and (<*>); and alternatives are described with (<|>). This contrasts with the `$n` scheme `Happy` uses to decide how to combine results; as a benefit, this parser is type-checked in-file, whereas type errors from `Happy` only materialise when the generated file is compiled. There is still some obvious duplication of the logic for combining comma-delimited "things," though the many combinator serves as a first-class implementation of the "zero or more" BNF extension, returning a list of results. Unlike with `Happy`, parser combinators are **flexible** and **extensible** so this duplicated code can be factored out into a new polymorphic compound combinator:

```
commaSep :: Parser a → Parser [a]
commaSep p = (:) <$> p <*> many (tok "," *> p)
```

This combinator captures the shared logic between comma-separated expressions and arguments, combining them into a list without knowing p's result type. Using this, the parser can be simplified again:

```
prog  = Prog <$> many (func <* tok ";") <*> expr
func  = Func <$> (tok "def" *> ident) <*> (tok "(" *> commaSep arg <* tok ")") <*> (tok "=" *> expr)
expr  = Ident <$> ident <|> Call <$> (tok "call" *> ident) <*> (tok "(" *> commaSep expr <* tok ")")
ident = some (oneOf ['a'..'z'])
```

Compared with the original example, this parser much more closely resembles the grammar in fig. 1.1a, with the last remaining noise being the tok combinator itself, as well as the combinators to combine the semantic actions. Further work could be performed to continue to improve this parser. However, when done well, the ability to perform factoring and develop new combinators allows them to be even more concise than their parser generator equivalents. Instead of learning a new syntax, and limitations, of a new DSL, users need only familiarise themselves with the core set of combinators and otherwise leverage the host language that they already know. In this case, the combinators used in the parser are standard Haskell Applicative combinators, which users may already be familiar with.

Why *Not* Parser Combinators?

The example parser outlined previously has highlighted some of the concrete strengths of the parser combinator approach, but it also revealed some of its flaws:

1. **Verbose** Parser combinator libraries tend to require more implementation noise than parser generator libraries, as interaction with semantic actions must be performed throughout the parser, not separately.

2. **Unstandardised** Parser combinator libraries are not as portable as parser generator libraries, where libraries like ANTLR have backends for several different languages. Particularly across different host languages, parser combinator APIS vary a lot, and library knowledge is not as readily transferable.

3. **Slower** While many parser combinator libraries boast good performance, in general they are slower than their hand-written counterparts, and the best performing libraries, like `megaparsec`, require specific knowledge of specialised combinators to get the best performance.

4. **Uninformative** Compared with hand-written parsers, parser combinators have a less fine-grained control over error message generation. However, many parser combinators do have flexible error systems, and can produce high-quality errors out-of-the-box, usually better than their parser generator counterparts. Strategies to make high-quality error messages in common industrial-strength libraries, however, is relatively undocumented and unexplored.

5. **Undirected** While the flexibility of parser combinators can be considered a strength, it also opens the door to bad software engineering practices too: it is easy to write messy and unmaintainable parsers. Sadly, the art of writing parser combinators is not often talked about.

In particular, the previous example demonstrated the increased verbosity of the parser combinators, with the (<$>), (<*>), (<*), and (*>) combinators providing syntactic noise to the final parser. It also partially showed how the unrestrictive nature of parser combinators can result in messy implementations: the compulsory separation of lexing and parsing within `Happy` ensures that the noise of whitespace was already kept clear of the parser, whereas the initial attempt with parser combinators saw whitespace littered throughout. Very few industrial-strength compilers make use of either parser combinators or generators: often the compiler is prototyped with one of the techniques but is later ported to a hand-written parser for performance and to improve error messages.

> **Thesis statement** *Parser combinators are a viable means of writing efficient and maintainable parsers with good error messages.*

While historically they been considered to be too slow for practical use with poor error messages, they do have the potential for high-quality error messages, like hand-written parsers, and when constructed non-ambiguously can parse in $O(n)$ time. Leveraging host language abstractions allows them to be written in a clear way, and any associated overhead can be eliminated with staged metaprogramming.

Contributions

The main contributions of this dissertation to the development of parser combinator state-of-the-art are to:

- Demonstrate that the translation between regular expressions and non-deterministic finite automata arises from the Cayley transformation, producing a compiler. This is generalised to support user-directed results in a parametricity-preserving way, with recursive parsers, and the full semantics of *selective* `parsec`-like parser combinators added. This representation is then staged to remove overheads, leaving a fast implementation (CHAPTER 3).

  This consolidates the work of Willis, Wu, and Pickering [2020] and Willis, Wu, and Schrijvers [2022], where I was first author, generated the ideas, and wrote the implementations. Willis, Wu, and Pickering [2020] showcased the original staging, analysis, deep-embedded compiler of `parsley`, serving as the base of the full implementation; and Willis, Wu, and Schrijvers [2022] focused primarily on showing the relationship between regular expressions and automata, providing a basis for how the `parsley` representation arises – this presentation did not expand to full implementation, which is completed by this dissertation.

- Recover context-sensitive power in a non-monadic way by introducing threaded parsing state known as *references*. These have a fine-grained lifetime that cannot leak, without otherwise compromising the type of the parser. This is required since monadic (>>=) cannot be ahead-of-time compiled, but is needed for context-sensitive grammars and it is shown informally that the expressive power of *selectives* and *references* matches that of monads (CHAPTER 4). The improved ergonomics of the types of these *references* can be lifted out into its own library improving on the ST monad.

- Provide concrete patterns to help capture idiomatic use of parser combinators and avoid common user mistakes (CHAPTER 7). These patterns are presented in both Scala and Haskell, and have gained first-class support in the API of `parsley` and `gigaparsec`. They help improve the maintainability of parsers.

This consolidates the work of Willis and Wu [2021] and Willis and Wu [2022], where I was first author, the ideas of which were either inspired from folklore or as developments inspired by my own observations as a teacher of parser combinators at Imperial.

- Iterate on the error systems of `parsec` and `megaparsec`, outlining some quirks and showing how they can be refined with a new set of error primitives as well as how the system can be implemented efficiently (CHAPTER 8). Further patterns are presented for fine-tuning error messages.

  This builds on work of my master's dissertation [Willis 2018] and Willis and Wu [2018], integrating into the existing Scala `parsley` library, which serves as a easy to iterate base implementation. The error patterns refine the discussion of Willis and Wu [2021], which did not have the combinators necessary to express the patterns properly.

Outline

CHAPTER 2 starts by outlining some of the key background material necessary to understand the the rest of the dissertation. This includes discussion about classic parsing ideas; the hierarchy of functional abstraction from *functor* to *monad*; parser combinators themselves, and their history; principled metaprogramming in the form of *staging*; Cayley representations and how it forms an optimisation strategy; as well as structured recursion schemes, which are used for traversal. From there, the dissertation is broadly broken into three parts. The first part, Chapters 3 to 6, centre around the design, foundations, and implementation of the *staged selective* Haskell `parsley` library; this is the main body of work undertaken during this dissertation and the full implementation can be found at `https://github.com/j-mie6/ParsleyHaskell`.

However, during my time at Imperial, I have been involved heavily with the second-year undergraduate compilers project, compiling a language called WACC; students are free to choose any language and parsing tool they wish to undertake the project. Many students choose to use Scala and Scala `parsley` [Willis 2018; Willis and Wu 2018], the library that originally formed my master's dissertation. This provided ample opportunity to see first-hand how people write parsers and the general problems they encounter doing so. The result of this was the creation of several design patterns for parser combinators, for which I have seen a noticeable improvement on the quality of the parsers written now that they are being used. It is these patterns that are the focus of the second part of the thesis (CHAPTER 7), and they are presented in both Haskell and Scala.

Furthermore, good error messages are a key component of the assessment criteria for WACC, which has fuelled the innovation in `parsley`'s error system, allowing students to produce the kind of high-quality errors we expect and not be disadvantaged for their choice of library. These days, the `parsley` implementations of WACC routinely obtain the highest marks for error quality with significantly less effort than required for similar quality errors from students using parser generators. The current state of this error system is the topic of the third part (CHAPTER 8). The implementation of Scala `parsley` can be found at `https://github.com/j-mie6/parsley`.

Haskell `parsley`

The discussion of Haskell `parsley` begins in CHAPTER 3 with a deep-embedded representation of regular expressions and non-deterministic finite automata. These are evolved and developed to support user-directed heterogeneous results and full recursion, capturing applicative parser combinators, then adapted to support PEG-style backtracking and *selective* combinators. This aims to build up the library from a core set of behaviours and the initial conversion from regular expression to automaton is done as a calculation via the Cayley transformation. The interpretive overheads of this compilation and subsequent evaluation are then eliminated by making use of staged metaprogramming, resulting in generated code of near hand-written quality. Finally, the issue of representing recursion in an observable yet non-obtrusive way is explored, leveraging Haskell's StableNames to automatically detect let-bound parsers.

Selective parser combinators are not as powerful as their monadic counterparts, losing the ability to perform generic context-sensitive parsing on their own. CHAPTER 4 builds on the existing selective API to introduce individual pieces of threaded parsing state, known as *references*, to allow the parser writer to persist results cleanly. The chapter then highlights how so-called *iterative* combinators can be improved and optimised by making use of references. These references are then integrated into the existing system, and their staged evaluation semantics is given, and analysis required to work out what references must be threaded into each binding is discussed. Finally, the improved expressive power of selectives with references is informally reasoned about, by providing the implementation of a Turing-complete language as part of a reference-driven interpreter.

To facilitate further optimisation and analysis of parser combinators, CHAPTER 5 develops a data structure, called a RangeSet, which efficiently stores *semi-contiguous data*, where many of the elements within the set are in contiguous ranges, in a compact way. This structure supports a time- and memory-efficient set complement operation and is ideal for representing character predicates as inspectable data.

CHAPTER 6 concludes the discussion on Haskell `parsley` by describing various optimisations and analyses that are performed on the parser AST to improve performance and reduce workload for GHC. This involves discussing: how staged terms can be efficiently generated via a higher-order abstract syntax version of the Lambda Calculus; how iterative parsers can be optimised accounting for the idempotency of failure; how input consumption can be factored out and optimised; and how checks within failure can be eliminated based on static knowledge of what input has been consumed. With the system complete, the chapter concludes by evaluating the performance of Haskell `parsley`, and the effectiveness of its optimisations in a collection of benchmarks.

Design Patterns of Parser Combinators

The running example of the "verbose Lambda Calculus" parser (page 3) highlighted how easy it is to write messy or unmaintainable parsers. In fact, there are some underlying problems that remain in the parser. CHAPTER 7 rounds off this discussion, by presenting concrete patterns and conventions that can be used to guide the design of a parser written with parser combinators including: handling left-recursion; dealing with whitespace and lexing; and decoupling semantic actions and meta-data extraction – like position information. After presenting the main patterns on a running expression example, the considerations needed to integrate these patterns into a library like `parsley` are discussed, helping to provide a foundation to support these patterns in a more standardised way.

Scala `Parsley`'s Error System

CHAPTER 8 explores the ways that the error system of Scala `parsley` has iterated on top of its predecessors, revealing quirks of the original systems, and developing a set of simple primitive combinators that work together to facilitate high-quality error messages. This system is designed in a way that is well encapsulated, as to maximise the evolution that can occur without breaking binary backwards compatibility, a key concern within the wider Scala ecosystem. The implementation of such a system can be slow if implemented naïvely, and there is a balance to be struck between laziness to reduce unnecessary computation time, and growing space leaks that arise from unforced computation; the way this issue is handled is presented, allowing for an efficient system. Finally, further design patterns for generating more bespoke error messages are discussed, highlighting how the specific set of combinators in `parsley` enriches how these patterns can be implemented; these patterns are given first-class support in the form of new compound combinators.

Related Work Chapter 9 contrasts the work against other work in the field, including: different approaches for the underlying implementations of parser combinator libraries; discussions of alternative set-like data structures; how design patterns have been employed for parsing more generally; what work has been done surrounding staging parser combinators; different error generation and recovery strategies; and how the expressive power of *arrowised* parser combinators differ from that of *selective* parser combinators.

Conclusion While some of the performance evaluation occurs within sections 5.4 and 6.5, Chapter 10 reflects on how the development of both `parsley` libraries have helped to address the shortcomings introduced in this chapter and a critical reflection on how well the outlined advantages have been retained in the process. It then outlines the future work on the `parsley` libraries moving forward.

Chapter 2

Background

This dissertation is centred around two main ideas: parser combinators and their optimisation and use; and staged metaprogramming and how it facilitates the optimisation of parser combinator libraries. This chapter discusses the relevant background required for the entire dissertation. In addition to parser combinators (§2.3) and staged metaprogramming (§2.4) the following concepts are also covered:

- Classical parsing techniques and algorithms, where they form useful contextualisation of some of the discussion in the dissertation (§2.1).
- Basic background into *functors*, *applicative functors*, *selectives*, and *monads*. This will outline the key laws and properties that relate to parser combinators (§2.2).
- Cayley representations and transformations, a common optimisation strategy for functional programs (§2.5).
- Structured recursion schemes, which form the backbone of the `parsley` compiler (§2.6).

Working knowledge of Haskell and Scala is assumed, but not in equal measure – appendix A explains the relevant Scala specific concepts that Haskellers may be unaware of.

2.1 Classical Parsing Techniques

There are many different grammar types, language recognition, and parsing strategies in common use. This section briefly outlines those that are relevant for other discussion within the dissertation. Primarily, finite-state automata (§2.1.3) are needed for CHAPTER 3: EMBEDDING A PARSER COMBINATOR LIBRARY; and parser expression grammars are the grammar model for applicative parser combinators (§2.1.2).

2.1.1 LL Parsing

An $LL(k)$ parser [Lewis and Stearns 1968] is a top-down parser that parses an unambiguous *context-free grammar* (CFG) only requiring k or fewer tokens of lookahead [Aho et al. 2006]. The most constrained of these is the $LL(1)$ class of parsers, which parse unambiguous grammars by only considering the next token of input. The $LL(k)$ class of parsers is a subset of $LR(k)$, which use a bottom-up parsing strategy [Knuth 1965; Aho et al. 2006].

2.1.2 Parser Expression Grammars

Parser expression grammars, or PEGS, were introduced by Ford [2004]. PEGS allow for the description of a wide class of grammars – according to Ford [2004] this extends to all deterministic $LR(k)$ grammars as well as some context-sensitive grammars too – and, notably, supports a left-biased choice operation. Packrat parsers [Ford 2002] can be used to parse all PEG grammars in linear time [Ford 2004]. A PEG consists of rules of the form $T \leftarrow e$ for some non-terminal T, and a *parser expression* e, where a parser expression is defined inductively:

- the empty string ϵ is a parser expression.
- any terminal, including literals and character classes, is a parser expression.
- any non-terminal T is a parser expression so long as is part of the non-terminal set of the PEG.
- ordered choice e_1/e_2, where e_2 is only parsed if e_1 fails, is a parser expression.
- the sequence $e_1 e_2$ is a parser expression.
- the negative lookahead $!e$ is a parser expression.

For a PEG to be well-formed, it must not contain any left-recursion, indirect or otherwise. The biased choice operation follows the same semantics of the choice operator for the `parsec`-style of parser combinator libraries (§2.3), although the choice operator for PEGs will always backtrack on failure. In fact, every PEG can be recognised by a `parsec`-style library – semantically, they are equivalent, assuming the use of heavy backtracking in the combinator implementation.

Mascarenhas, Medeiros, and Ierusalimschy [2014] show that there is a correspondence between PEGs and CFGs, such that the two only differ by their choice operation: CFGs use an unbiased choice. They show that $LL(1)$ and strong-$LL(k)$ grammars can always be transformed into valid PEGs.

Ford [2004] identifies two properties of PEGs, applicable to parser combinators: given an expression repeated zero-or-more times $e*$, if e succeeds without consuming any input, then $e*$ will never terminate (called the $*$-loop condition); and whilst a PEG can be left-factored, it cannot be right-factored without altering its meaning.

2.1.3 Finite State Automata

Finite state automata are broadly a set of different kinds of automata that have a finite number of states, with transitions between states occurring on input consumption. Some automata may make state transitions based on the contents of a stack or may write out to an auxiliary tape. Each different kind of automaton can be used to *recognise* a particular class of languages: this means it does not necessarily produce a result but can answer "yes" or "no" to whether a string is in a language or not.

Non-Deterministic Finite Automata

A non-deterministic finite automaton [Rabin and Scott 1959; Sipser 2013; Thompson 1968], or NFA, is a machine with a finite set of states, where each state transitions to one or more states if a specific character is present next in the input. More concretely, and distinguishing between types and values, an NFA is an inhabitant of the following strict[1] Haskell datatype:

```
data NFA Q Σ = NFA { δ  :: Q → Maybe Σ → Set Q
                   , q₀ :: Q
                   , F  :: Set Q
                   }
```

[1]To avoid issues with $\bot$, assume the `Strict` GHC extension is enabled.

An NFA is parameterised by a state type Q, an alphabet type Σ, and requires three values: a transition function δ, which takes a state and potentially a symbol of the alphabet and produces a set of next states; a start state q_0, where execution begins; and a set of accept states F, denoting states on which termination implies success. Two aspects of the transition function δ contribute to the non-determinism: a symbol from the alphabet is optionally required for a transition (called an ϵ-transition); and each transition can map to multiple states. These can be used to parse *regular languages* [Chomsky 1956; Kleene et al. 1956].

Push-Down Automata

Compared to NFAs, push-down automata [Chomsky 1962] (PDA) are additionally equipped with a stack, where the top element can be manipulated during a state transition. This can be represented as follows:

$$
\begin{aligned}
&\textbf{data}\ \mathsf{PDA}\ Q\ \Sigma\ \Gamma = \mathsf{PDA}\ \{\ \delta\ :: Q \to \mathsf{Maybe}\ \Sigma \to \mathsf{Maybe}\ \Gamma \to \mathsf{Set}\ (Q, \mathsf{Maybe}\ \Gamma)\\
&\qquad\qquad\qquad\qquad\qquad\ ,\ q_0 :: Q\\
&\qquad\qquad\qquad\qquad\qquad\ ,\ F\ :: \mathsf{Set}\ Q\\
&\qquad\qquad\qquad\qquad\qquad\ \}
\end{aligned}
$$

In addition to the NFA type given in §2.1.3, the PDA type is parameterised by a stack alphabet Γ, and the transition function δ is altered to take an optional stack symbol, which must be popped from top of the stack for the transition to occur, as well as return an optional stack symbol, which should be pushed on transition. PDAs can recognise all regular languages and context-free languages [Chomsky 1956].

Moore Machine

Moore machines [Moore 1956] are a type of deterministic finite state transducer, which produces output as well as consuming input. A Moore machine is an inhabitant of the following strict Haskell type:

$$
\begin{aligned}
&\textbf{data}\ \mathsf{Moore}\ Q\ \Sigma\ \Lambda = \mathsf{Moore}\ \{\ \delta\ :: Q \to \Sigma \to Q\\
&\qquad\qquad\qquad\qquad\qquad\quad\ ,\ q_0 :: Q\\
&\qquad\qquad\qquad\qquad\qquad\quad\ ,\ F\ :: \mathsf{Set}\ Q\\
&\qquad\qquad\qquad\qquad\qquad\quad\ ,\ G :: Q \to \Lambda\\
&\qquad\qquad\qquad\qquad\qquad\quad\ \}
\end{aligned}
$$

Note that, in its original formulation, the Moore machine does not have accept states and runs forever, but, for parsing, accept states are assumed. The Moore machine differs from the NFA by having an output alphabet Λ, and a function G from state to a symbol to output – output is associated with the states themselves. Moore machines are deterministic, evidenced by the non-optional symbol in δ, which also only maps to a single state. Mealy machines [Mealy 1955] are similar, but instead of G they instead have a $\delta :: Q \to \Sigma \to (Q, \Lambda)$, so that output happens on edges and not states. For the purposes of this dissertation, both machines are indistinguishable, and Moore machine are chosen to create a better separation between state transition and outputting.

2.2 Functors, Applicative Functors, and Monads

Functors [Goguen and Thatcher 1974; Jones 1995], *applicative functors* [McBride and Paterson 2008], and *monads* [Wadler 1992; Wadler 1995; Spivey 1990] are a hierarchy of powerful abstractions for building effectful programs. Each typeclass is equipped with laws that govern the interactions between each of their operations.

2.2.1 Functors

The Functor typeclass is used to describe structures that contain values of a given parametric type, for which these values can be mapped into another parametric type using the operation fmap or its infix form (<$>).

```
class Functor (f :: ∗ → ∗) where fmap :: (a → b) → f a → f b
```

A lawful implementation must adhere to the laws outlined in fig. 2.1: mapping the identity function has no effect (eq. (2.1)), and the composition of maps can be re-written as a map of composition (eq. (2.2)). These laws mean that fmap cannot affect the shape or effects of a given value. Using fmap, some useful *combinators* can be defined:

```
(<$) :: Functor f ⇒ b → f a → f b
y <$ fx = fmap (const y) fx
void :: Functor f ⇒ f a → f ()
void fx = () <$ fx
```

Both, along with fmap, form valid and useful combinators for writing parsers: (<$) allows for the result of a parser to be replaced with a constant, of which void is a special case that replaces the result with ().

2.2.2 Applicative Functors

More constrained than Functor is Applicative, which provides a means to combine two effectful values whilst combining their effects; there is no ordering imposed on the execution of these effects. It is defined as:

```
class Functor f ⇒ Applicative (f :: ∗ → ∗) where
  pure :: a → f a
  (<*>) :: f (a → b) → f a → f b   -- pronounced "ap"
```

The function pure represents the lifting of a value into the type f without having an effect. The operator (<*>) applies functions produced by the execution of the left computation and to values produced by the right. In the context of parsers, this is a sequencing operation, used to combine results of two sequential parsers together. Applicative functors also adhere to laws (fig. 2.2): eqs. (2.3) to (2.5) demonstrate that pure has no effect[2]; and eq. (2.6) shows (<*>) is associative. The applicative laws can be used to prove the functor laws via eq. (2.7). The pure and (<*>) *combinators* can be used to provide a variety of other helpful combinators:

```
liftA2 :: Applicative f ⇒ (a → b → c) → f a → f b → f c   -- and so on for liftA3 etc
liftA2 f p q = f <$> p <*> q
```

[2] ($) :: (a → b) → a → b is an operator representing function application.

$$\text{fmap id} = \text{id} \quad (2.1) \qquad \text{fmap f} \cdot \text{fmap g} = \text{fmap (f} \cdot \text{g)} \quad (2.2)$$

Fig. 2.1: Functor laws

$$\text{pure id} <\!\!*\!\!> \text{u} = \text{u} \quad (2.3) \qquad \text{u} <\!\!*\!\!> \text{pure x} = \text{pure (\$ x)} <\!\!*\!\!> \text{u} \quad (2.5)$$

$$\text{pure f} <\!\!*\!\!> \text{pure x} = \text{pure (f x)} \quad (2.4) \qquad \text{u} <\!\!*\!\!> \text{(w} <\!\!*\!\!> \text{v)} = \text{pure} (\cdot) <\!\!*\!\!> \text{u} <\!\!*\!\!> \text{w} <\!\!*\!\!> \text{v} \quad (2.6)$$

$$\text{f} <\!\!\$\!\!> \text{fx} = \text{pure f} <\!\!*\!\!> \text{fx} \quad (2.7)$$

Fig. 2.2: Applicative functor laws

```
(<**>) :: Applicative f ⇒ f a → f (a → b) → f b   -- pronounced “reverse ap”
(<**>) = liftA2 (flip ($))
(*>)   :: Applicative f ⇒ f a → f b → f b         -- pronounced “then”
(*>)   = liftA2 (const id)
(<*)   :: Applicative f ⇒ f a → f b → f a         -- pronounced “then discard”
(<*)   = liftA2 const
(<:>)  :: Applicative f ⇒ f a → f [a] → f [a]     -- pronounced “cons”
(<:>)  = liftA2 (:)
sequence :: Applicative f ⇒ [f a] → f [a]
sequence = foldr (<:>) (pure [ ])
traverse :: Applicative f ⇒ (a → m b) → [a] → f [b]
traverse f = sequence · map f
```

The two combinators (<*) and (*>) are commonly used for a parser: they sequence two parsers together and return the result of one of them and ignore the other. The sequence and traverse combinators demonstrate how combinators can be built out of smaller components using the higher-order features of the host language.

Monoidal Functors

An honourable mention for the typeclass hierarchy is Monoidal, which represents monoidal functors.

```
class Functor f ⇒ Monoidal (f :: * → *) where
  unit :: f ()
  (<~>) :: f a → f b → f (a, b)   -- pronounced “product” or “zip”
```

This typeclass is equivalent to the Applicative typeclass, where (<~>) is a generalised cartesian product:

```
  -- Monoidal as Applicative              -- Applicative as Monoidal
(<~>) = liftA2 (, )                     p <*> q = uncurry ($) <$> (p <~> q)
unit  = pure ()                         pure x  = x <$ unit
```

The Monoidal typeclass provides a more natural API for uncurried languages, like Scala:

```
liftProduct :: Monoidal f ⇒ ((a, b) → c) → f a → f b → f c
liftProduct f p q = f <$> (p <~> q)
(~>) :: Monoidal f ⇒ f a → f b → f b   -- pronounced “then” or “right product”
(~>) = liftProduct snd
(<~) :: Monoidal f ⇒ f a → f b → f a   -- pronounced “then discard” or “left product”
(<~) = liftProduct fst
```

2.2.3 Alternative Functors

Alternative applicative functors extend the Applicative interface with a notion of choice and failure:

```
class Applicative f ⇒ Alternative (f :: * → *) where
  empty :: f a
  (<|>)   :: f a → f a → f a   -- pronounced “or”
```

There are many different possible interpretations for the (<|>) combinator: As such, which laws are part of Alternative are contested, with some instances obeying some laws and others obeying a slightly different set. The laws in fig. 2.3 reflect the general laws, as well as some of the controversial laws that do hold true for parser combinator libraries like `parsley`, but not for all Alternatives: eqs. (2.8) to (2.10) demonstrate associativity and that empty is a zero; eq. (2.11) demonstrates left-biased choice; and eq. (2.12) shows that failure terminates a branch. Some proposed alternative laws are rejected for parsers [Gibbons and Hinze 2011] and are not listed here. There are helpful combinators that result from these operations:

```
choice :: Alternative f ⇒ [f a] → f a
choice = foldr (<|>) empty
many :: Alternative f ⇒ f a → f [a]
many u = let go = u <:> go <|> pure [ ] in go
some :: Alternative f ⇒ f a → f [a]
some u = u <:> many u
```

In particular, many and some form an important pair of combinators for performing an effect zero- or one-or-more times, respectively. For parsing, this means that they correspond to the Kleene-star and Kleene-plus operations.

2.2.4 Selective Functors

Positioned between Applicative and Monad is the selective applicative functor [Mokhov et al. 2019], or selective functor for short. If Applicative provides a way of expressing function application at the effect level, then Selective provides a way of expressing case discrimination:

$$\mathsf{empty} \mathbin{\langle|\rangle} u = u \tag{2.8}$$

$$u \mathbin{\langle|\rangle} \mathsf{empty} = u \tag{2.9}$$

$$(u \mathbin{\langle|\rangle} v) \mathbin{\langle|\rangle} w = u \mathbin{\langle|\rangle} (v \mathbin{\langle|\rangle} w) \tag{2.10}$$

$$\mathsf{pure}\ x \mathbin{\langle|\rangle} u = \mathsf{pure}\ x \tag{2.11}$$

$$\mathsf{empty} \mathbin{\langle*\rangle} u = \mathsf{empty} \tag{2.12}$$

$$f \mathbin{\langle\$\rangle} (u \mathbin{\langle|\rangle} v) = (f \mathbin{\langle\$\rangle} u) \mathbin{\langle|\rangle} (f \mathbin{\langle\$\rangle} v) \tag{2.13}$$

Fig. 2.3: Alternative applicative functor laws

$$\text{branch u (pure f) (pure g)} = \text{either f g} \mathbin{<\$>} \text{u} \tag{2.14}$$

$$\text{select (pure x) (u} \ast\!> \text{v)} = \text{select (pure x) u} \ast\!> \text{select (pure x) v} \tag{2.15}$$

$$\text{u} <\!\ast\!> \text{v} = \text{select (Left} \mathbin{<\$>} \text{u) (flip (\$)} \mathbin{<\$>} \text{v)} \tag{2.16}$$

$$\text{branch (pure (Left x)) u v} = \text{u} <\!\ast\!> \text{pure x} \tag{2.17}$$

$$\text{branch (pure (Right y)) u v} = \text{v} <\!\ast\!> \text{pure y} \tag{2.18}$$

$$\text{branch u v w} = \text{branch (either Right Left} \mathbin{<\$>} \text{u) w v} \tag{2.19}$$

Fig. 2.4: Selective functor laws

```
class Applicative f ⇒ Selective (f :: ∗ → ∗) where
  branch :: f (Either a b) → f (a → c) → f (b → c) → f c
```

Though branch can be implemented as liftA3 (λx f g → either f g x), this would necessitate executing the effects of all three computations. However, selective functors often skip the execution of one of the last two arguments depending on the result returned by the first. The original presentation of Selective [Mokhov et al. 2019] presents the typeclass in terms of a different operator:

```
select :: f (Either a b) → f (a → b) → f b
select p q = branch p q (pure id)
```

This is mostly interchangeable with branch: branch is implementable with two selects, though this removes the possiblity of identifying mutual-exclusion [Mokhov et al. 2019]. A selective functor is known as *rigid* when eq. (2.16) holds true – other selective laws are given in fig. 2.4, and more specific ones that do not hold for all selectives are eqs. (2.17) to (2.19). One important combinator is defined in conjunction with Alternative:

```
filterS :: (Selective f, Alternative f) ⇒ (a → Bool) → f a → f a
filterS f mx = select (pure cond <∗> mx) empty
  where cond x = if f x then Right x else Left ()
```

The filterS f mx combinator allows a computation mx to fail if a predicate f does not hold true on its result. This finds use in parsing (see §7.1.2), and is subject to filter fusion:

$$\text{filterS f} \cdot \text{filterS g} = \text{filterS} (\lambda x \rightarrow \text{f x} \land \text{g x}) \tag{2.20}$$

The other combinators of interest are the whenS and whileS combinators, which find use in conjunction with *parsing references* (see CHAPTER 4) to enable context-sensitive parsing; and the ifS combinator, which represents a standard conditional expression:

```
ifS :: Selective f ⇒ f Bool → f a → f a → f a
ifS mb mt mf = branch ((λb → if b then Right () else Left ()) <$> mb) (const <$> mf) (const <$> mt)
whenS :: Selective f ⇒ f Bool → f () → f ()
whenS mb mx = ifS mb mx unit
```

```
whileS :: Selective f ⇒ f Bool → f ()
whileS mb = fix (whenS mb)
```

The whenS mb mx combinator allows for the execution of an effect mx only when mb returns True, and the whileS combinator repeatedly executes its argument until it returns False.

2.2.5 Monads

Monad is used to create dynamic sequencing of effects:

```
class Applicative m ⇒ Monad m where
  (»=) :: m a → (a → m b) → m b    -- pronounced “bind”
```

Unlike Applicative, which does not impose an ordering on the effects during composition, Monad naturally creates an ordering by ensuring that one computation must be fully evaluated before its result can be used to form the next. The (»=) combinator can be used to re-use a result, delay its use, or make dynamic choices. Compared with Selective these choices are not confined to one of two different known choices but are generated on demand dynamically. Like its counterparts, Monad is adherent to a selection of laws, though these are not relevant for this dissertation: as it will turn out, monads are not suitable for a *staged* parser combinator library like `parsley`. That said, there is one extra combinator of note:

```
join :: Monad m ⇒ m (m a) → m a
join mmx = mmx »= id
```

The combinator join allows for the flattening of a monadic structure, sequencing the inner effect after the outer one. This can be used to delay effects by temporarily wrapping them in a pure context before re-incorporating them at a later point.

2.3 Parser Combinators

Parser combinators [Hutton 1992; Hutton and Meijer 1996; Leijen and Meijer 2001; Swierstra and Duponcheel 1996; Wadler 1985] are a functional approach to writing top-down recursive descent parsers in a palatable way. All parser combinator libraries provide functionality in terms of simple base combinators, and usually expose increasing more complex composed combinators for higher-level development. The advantage of this is that it allows the parser writer to abstract away the finer details of parser writing in favour of expressing the overall structure of the grammar more concisely.

Parser combinators offer much lower-level control over the parsing process compared to parser generators. Obviously, however, parser combinators offer less control than a truly hand-written parser but are less onerous to write. Parser combinators are usually expressed as a first-class value in the language they are being written in as an embedded domain-specific language (EDSL) [Hudak 1996], allowing for the use of host-abstraction to construct parsers, and straightforward integration of semantic actions directly into the parser.

2.3.1 Origins of Parser Combinators

The first time parser combinators appeared in the functional programming literature was Wadler [1985], but Wadler points out that many of the techniques he outlined were already folklore known to the community, and Hutton notes that the basic ideas originated as far back as Burge [1975]. Wadler [1985] outlines the general structure of parser combinators for decades to come, albeit with non-modern naming, and his discussion only goes as far as (<|>), (<$>), many, some, liftA2, pure, and empty – though using different names. The presence of liftA2 and pure, however, means that the very first discussion of parser combinators was that of *applicative parser combinators*, and Wadler notes that this is useful for $LL(1)$ parsing. This work also introduces the *lit* combinator, which is often referred to as *char*, as the primitive combinator for parsing.

In 1992, Hutton [1992] fleshed out the ideas from Wadler [1985], interestingly adopting the Monoidal (<~>) combinator, which he calls seq, instead of liftA2 – he explicitly points out the use in using (<$>) to extract the desired results from the (<~>) combinator. Instead of the primitive lit, Hutton instead introduces the *satisfy* combinator, which generalises lit to take a predicate matching many possible characters: this combinator remains popular in modern libraries as the primitive, since it can be used to implement lit in a concise way (§2.3.2). The main contribution of his work is a discussion on how to construct parsers with combinators, presenting the classic example of simple expression parsing. The initial representation of parsers used is the "A Parser for Things is a function from Strings to Lists of Pairs of Things and Strings!" a rhyme that Hutton popularised [Hutton 2016] but that is attributed to Fritz Ruehr: this represents a fully non-deterministic parser that returns all possibly ambiguous results of the parser. However, Hutton goes on to mention that, for many practical parsing use-cases, unambiguous grammars are often used and the Maybe type may be better suited; he goes on to extend this with a three-valued datatype that also distinguishes between two different kinds of value to introduce error reporting into the paradigm (this introduces the nofail combinator, which is a cut-like combinator for helping refine errors).

In 1996, Hutton and Meijer [1996] presented monadic parser combinators, and their implementation. They demonstrate the application of monadic **do**-notation to parsers, showing how satisfy can be implemented in terms of the parser that reads any single character *item* and (>>=), and introduces the idea of chaining combinators, which are important for resolving left-recursion in an idiomatic way (§7.1.1). The introduction of Monad to parsing brings parser combinators out of context-free grammars and into general context-sensitive parsing.

Around the same time, Partridge and Wright [1996] discuss the error reporting aspect of parser combinators, noting that the three-valued approach to parser combinators for error-reporting has the drawback that the nofail combinator has to be annotated consistently around the parser for the errors to be effective: they demonstrate that for $LL(1)$ predictive parsers that this is unnecessary, and that this can be automated. They achieve this by refining the type of the parser to return a four-valued datatype, encoding each of the following states of a parser: the parser recognised the empty string and succeeded (*epsilon ok*); the parser consumed input and succeeded (*consumed ok*); the parser failed, but without consuming input (*epsilon error*); and the parser failed having already consumed input (*consumed error*). This introduced, for the first time, the idea that a parser p <|> q is not only left-biased, following the semantics of PEG (§2.1.2), but also only tries to parse its second alternative q when p fails having consumed no input. The claim is that this results in good quality error reporting without any need for manual annotation, but the parsers are necessarily non-ambiguous for this to work. At the end of the paper, they suggest that, while their presentation of this idea materialises as a four-valued datatype, the idea could be easily adapted to make use of continuation-passing style.

In 2001, the ideas of Hutton and Meijer [1996], Fokker [1995], and Partridge and Wright [1996] were picked up by Leijen and Meijer [2001], which culminates in the first industrial strength monadic parser combinator library in Haskell, `parsec`[3]. The main aims of this work were to address the performance issues of previous presentations of parser combinators and generate good error messages. While they acknowledge the work of Swierstra and Duponcheel [1996] in creating good error reporting and recovery, they note that this does not work with a monadic interface and necessitates a separate lexing pass. The underlying model for `parsec` builds directly on the off-hand comment at the end of Partridge and Wright [1996], adopting a *four-continuation* approach to parsing, with the four continuations `cok`, `eok`, `cerr`, and `eerr` aligning with the previously identified four-values required for good parsing errors. However, `parsec` was not limited to unambiguous parsers, introducing the *try* combinator for opt-in backtracking. It is the semantics and architecture of `parsec` underlying the design of `parsley` and the work in this dissertation. The remaining discussion about the basics of parser combinators will be rooted around the `parsec`-style of parser combinator library.

2.3.2 Parser Combinator Primitives

This section will outline: character consuming combinators and their derivatives; combinators that control lookahead, both positive and negative; and the combinator that controls backtracking, atomic.

Reading Characters

The most primitive combinator for any parsing library is the combinator that reads a single character. However, there are several candidates for this role, each with their own trade-offs:

```
item    :: Parser Char
char    :: Char → Parser Char
satisfy :: (Char → Bool) → Parser Char
oneOf   :: Set Char → Parser Char
```

Each of these consume a single character from the input if available. The simplest, item, reads any single character: this makes it is very straightforward to implement, but requires the filterS (§2.2.4) combinator to implement any of the others, reducing static inspectibility (§2.4.1). The char combinator reads a specific character, failing if it is not found next in the input: this is highly inspectable, and can be used to implement oneOf, but item is prohibitively expensive to implement using purely char. Instead, satisfy, which reads any character that matches the given predicate, can easily implement all the others, but its inspectibility is limited as it uses a function. The oneOf combinator is a generalised form of char, matching one of a specific set of characters, and suffers from the same limitation. All considered, the best choice for the primitive is the satisfy combinator, though it may be also worthwhile to make oneOf a primitive too:

```
item      = satisfy (const True)
char c    = satisfy (≡ c)
```

[3] https://hackage.haskell.org/package/parsec

```
oneOf cs   = satisfy (flip Set.member cs)
noneOf cs = satisfy (not · flip Set.member cs)
```

There are two notable laws for satisfy's interactions with the other basic combinators:

$$\text{satisfy f} \mathbin{<\!|\!>} \text{satisfy g} = \text{satisfy} (\lambda c \longrightarrow \text{f c} \vee \text{g c}) \tag{2.21}$$

$$\text{atomic (filterS g (satisfy f))} = \text{satisfy} (\lambda c \longrightarrow \text{f c} \wedge \text{g c}) \tag{2.22}$$

Eq. (2.21) shows that two satisfys in disjunction with each other can be merged by taking the disjunction of their predicates. Eq. (2.22) is similar to eq. (2.20) and the relationship is immediately clear when examining the definition of satisfy in terms of item: satisfy f = atomic (filterS f item) – here, the atomic combinator ensures that the filtering and character consumption happen together and is introduced in §2.3.2.

Parsing Strings The most primitive character combinators can be used to build a variety of helpful parsers, like those for reading digits or letters, along with the string combinator:

```
string :: String ⟶ Parser String
string = traverse char
```

This combinator tries to parse a string exactly as given at the front of the input stream. If this fails having read characters, this combinator will have failed having consumed input for the purposes of backtracking.

Controlling Backtracking

In §2.3.1, it was mentioned that Partridge and Wright [1996] introduced the idea that the (<|>) combinator should not backtrack if input was consumed along a branch. The rationale was to preserve the deepest point into the input that a parser had gotten, to report errors from there. This approach is effective but relies on the fact that the grammar must be unambiguous recursive-descent, or $LL(1)$. While this is a desirable property for a parser to have, it is not always practical to left-factor a parser completely, especially when the backtracking performed would not otherwise increase the asymptotic complexity of the parser. Historically, the try combinator counters the behaviour of the left-biased (<|>) combinator and allows a parser to fail as if it has not consumed input:

```
try :: Parser a ⟶ Parser a
```

If parsers p and q share a common prefix, then p <|> q cannot successfully read q, because p would have parsed the common prefix. Instead try p <|> q can successfully parse q. The improved error messages, coupled with the lack of a need to excessively annotate a parser with cut-points for performance, makes this well suited for a high-performance library like `parsec` or `parsley`.

Nomenclature The try combinator was first created in `parsec`, and this name has persisted in its direct successors like `attoparsec`[4] and `megaparsec`[5]. However, many people get confused about the combinator; with

[4] https://hackage.haskell.org/package/attoparsec
[5] https://hackage.haskell.org/package/megaparsec

some people putting them in the wrong places, or holding a misconception that trys is inherently inefficient. The meaning that many seem to associate with the combinator is "the combinator allows a parser to try another alternative in an $\langle|\rangle$ if it fails, by backtracking". However, a more precise description is "the combinator ensures that a parser is handled atomically: either it consumes all the required input or none of it;" it is the semantics of $\langle|\rangle$ itself that performs backtracking, and the try combinator facilitates this by preventing partial input consumption on failure. Anecdotally, explained like this, people seem to have a much better sense of how it interacts with the wider parser; moreover, it becomes clearer that the combinator is not tied explicitly to $\langle|\rangle$ and has other uses. As such, this combinator is renamed to atomic, which better reflects its actual meaning[6]. In this dissertation, except for specifically referencing the `megaparsec` or `parsec` try, the combinator atomic will be used.

Laws The atomic combinator interacts with other smaller primitive parsers and combinators:

$$\text{atomic (satisfy f)} = \text{satisfy f} \quad (2.23) \qquad \text{atomic empty} = \text{empty} \quad (2.25)$$

$$\text{atomic (pure x)} = \text{pure x} \quad (2.24) \qquad \text{atomic (f} \langle\$\rangle \text{ u)} = \text{f} \langle\$\rangle \text{ atomic u} \quad (2.26)$$

Eqs. (2.23) to (2.25) demonstrate that atomic has no effect when applied to things that only consume one or fewer characters; and eq. (2.26) shows that atomic distributes across mapping.

Positive and Negative Lookahead

Lookahead allows a parser to inspect future input without committing to consuming it. Parser combinator libraries often have some mechanism for performing consumptionless lookahead. This can be both positive, where a parser must succeed from the current input, or negative, where a parser must fail from the current input. This takes the form of two combinators:

$\text{look}_+ :: \text{Parser a} \to \text{Parser a}$

$\text{look}_- :: \text{Parser a} \to \text{Parser ()}$

The look_+ combinator can look forward at the input and try to execute a given parser: should it succeed, the combinator succeeds having consumed no input, but if it fails it may consume input. Since the lookahead is

[6]This has the benefit that in many languages, like Scala, `try` is a keyword, which is no longer an issue.

$$\text{look}_+ \text{ empty} = \text{empty} \quad (2.27) \qquad \text{look}_- \text{ p} *\rangle \text{ look}_+ \text{ q} = \text{look}_+ \text{ (look}_- \text{ p} *\rangle \text{ q)} \quad (2.34)$$

$$\text{look}_+ \text{ (pure x)} = \text{pure x} \quad (2.28) \qquad \text{atomic} \cdot \text{look}_- = \text{look}_- \cdot \text{atomic} = \text{look}_- \quad (2.35)$$

$$\text{look}_- \text{ empty} = \text{unit} \quad (2.29) \qquad \text{look}_- \cdot \text{look}_+ = \text{look}_+ \cdot \text{look}_- = \text{look}_- \quad (2.36)$$

$$\text{look}_- \text{ (pure x)} = \text{empty} \quad (2.30) \qquad \text{look}_+ \cdot \text{look}_+ = \text{look}_+ \quad (2.37)$$

$$\text{atomic} \cdot \text{look}_+ = \text{look}_+ \cdot \text{atomic} \quad (2.31) \qquad \text{look}_- \cdot \text{look}_- = \text{void} \cdot \text{look}_+ \cdot \text{atomic} \quad (2.38)$$

$$\text{fmap f} \cdot \text{look}_+ = \text{look}_+ \cdot \text{fmap f} \quad (2.32) \qquad \text{look}_- \text{ (atomic p} \langle|\rangle \text{ q)} = \text{look}_- \text{ p} *\rangle \text{ look}_- \text{ q} \quad (2.39)$$

$$\text{look}_+ \text{ p} \langle|\rangle \text{ look}_+ \text{ q} = \text{look}_+ \text{ (p} \langle|\rangle \text{ q)} \quad (2.33) \qquad \text{look}_- \text{ p} \langle|\rangle \text{ look}_- \text{ q} = \text{look}_- \text{ (look}_+ \text{ p} *\rangle \text{ look}_+ \text{ q)} \quad (2.40)$$

$$\text{look}_+ \text{ (satisfy f)} *\rangle \text{ satisfy g} = \text{satisfy } (\lambda\text{c} \to \text{f c} \land \text{g c}) \quad (2.41)$$

$$\text{look}_- \text{ (satisfy f)} *\rangle \text{ satisfy g} = \text{satisfy } (\lambda\text{c} \to \text{not (f c)} \land \text{g c}) \quad (2.42)$$

Fig. 2.5: Lookahead laws

positive, the result of the parser can be returned. The look_ combinator will never consume input and only succeeds if the given parser fails; as such result is (). Only look_+ needs to be a primitive combinator, as it is possible to implement negative-lookahead with the monadic combinator join, atomic, and look_+:

```
look_ p = join (atomic (look₊ p *> pure empty) <|> pure unit)
```

The idea is that the result of having successfully looked ahead at p is the empty parser itself, otherwise, if p cannot be parsed, any input consumed must be rolled back, and the unit parser is returned instead. Then, once the scope of the (<|>) has been left, the join combinator takes whichever parser was returned and executes it. It is important that the failure or success of the combinator is delayed until after having left the scope of (<|>), otherwise it would unconditionally recover for a parser that fails having consumed no input and the combinator would have no effect[7]. However, not all parsing libraries, like `parsley`, support monads . Instead, either the look_ combinator can be made primitive or the selective whenS combinator can be used instead:

```
look_ p = whenS (atomic (True <$ look₊ p) <|> pure False) empty
```

This version of look_ tries to parse p, and if it succeeds returns True, denoting success, or False otherwise. When this parser returns True the whenS combinator will execute the empty combinator and fail, otherwise it will do nothing, equivalent to executing unit. This serves as a good example of how selective combinators can be used to replace monadic combinators in situations where the possible outcomes of a parser are known: in this case empty or unit. To my knowledge, this selective implementation has not been presented before.

Checking for no more input The look_ can be used to define eof, which only succeeds if there is no input remaining. This is defined as eof = look_ item. This is useful to ensure that all input has been parsed.

Laws The two lookahead combinators interact with each other, and other combinators via laws (fig. 2.5):

- Eqs. (2.27) to (2.30) show a similarity with boolean logic, where pure is True, empty is False, and look_ is not.
- Eqs. (2.31) and (2.32) demonstrate that look_+ commutes with some operations, whereas eqs. (2.33) and (2.34) demonstrate that it distributes as well.
- Eqs. (2.35) and (2.36) show that look_ eliminates adjacent uses of both atomic and look_+.
- Eq. (2.37) shows that look_+ is idempotent, whereas eq. (2.38) shows that double negation can be transformed into positive lookahead with discarded results.
- Eqs. (2.39) and (2.40) are De Morgan's laws for look_, where (<|>) is treated like $\vee$ and (*>) is treated as $\wedge$. This is related to Kleene-algebras with tests [Kozen 1997], but where commutativity of $\wedge$ does not hold, since this would make sequencing parsers commutative, which is obviously broken.
- Eqs. (2.41) and (2.42) show how lookahead with a single satisfy can be fused with an adjacent satisfy by merging the predicates.

[7]this is a known bug with parsec: `https://github.com/haskell/parsec/issues/8`.

2.3.3 Implementations of Parser Combinators

The semantics of the library developed in this dissertation is one that parses ambiguous grammars in a left-biased way, where backtracking is not performed by default for branches that have consumed input. This type of parser must make use of the "four-values" approach [Partridge and Wright 1996; Leijen and Meijer 2001], which can distinguish between successes and failures as well as whether they have consumed input or not in either case. The base type for this might be:

$$\textbf{type}\ \mathsf{Parser\ a} = \mathsf{String} \rightarrow \mathsf{Either\ (Err, Maybe\ String)\ (a, Maybe\ String)}$$

The result of this type does indeed encode all four values: Left (err, Nothing) is error with no consumed input; Left (err, Just inp′) is error with input consumed; Right (x, Nothing) is success with no consumed input; and Right (x, Just inp′) is success with consumed input. This type, as it stands, would introduce a space-leak into the parser, since the old input must be retained by the (<|>) combinator to feed to the second branch. However, it does keep the discussion clear, so this is overlooked for now and rectified later. This datatype-based approach to encoding the four possible states of a parser has a more obvious cost in the amount of allocation it performs, and it is less amenable to inlining and optimisation by GHC. Instead, it is possible to refine it using *continuation-passing style* (CPS) [Sussman and Steele 1998; Reynolds 1993]. The CPS transformation is generally:

$$a \rightarrow b \Rightarrow \forall r.\, a \rightarrow (b \rightarrow r) \rightarrow r$$

This alone may improve performance, but the advantage comes from simplifying the resulting type into something more efficient. To do this, note that – ignoring ⊥ – two types are isomorphic if they have an equal number of inhabitants: this means that every value of a given type can be mapped uniquely and reversibly onto a value of another type with the same number of inhabitants. Types can now be thought of as numbers, and the laws of commutative rings with identity can be applied to types as shown in fig. 2.6.

By CPS transforming the type Parser and applying these laws, a more efficient implementation is derived:

$$\begin{aligned}
&\textbf{type}\ \mathsf{Parser\ a} = \mathsf{String} \rightarrow \mathsf{Either\ (Err, Maybe\ String)\ (a, Maybe\ String)}\\
&\qquad = \quad \{\ \text{CPS transform}\ \}\\
&\qquad\quad \forall \mathsf{r.String} \rightarrow (\mathsf{Either\ (Err, Maybe\ String)\ (a, Maybe\ String)} \rightarrow \mathsf{r}) \rightarrow \mathsf{r}\\
&\qquad = \quad \{\ \text{apply eq. (2.49)}\ \}\\
&\qquad\quad \forall \mathsf{r.String} \rightarrow ((\mathsf{Err, Maybe\ String}) \rightarrow \mathsf{r}, (\mathsf{a, Maybe\ String}) \rightarrow \mathsf{r}) \rightarrow \mathsf{r}\\
&\qquad = \quad \{\ \text{apply}\ \mathsf{Maybe\ a} \cong \mathsf{Either\ a\ ()}\ \text{(as Maybe a has a + 1 inhabitants)}\ \}
\end{aligned}$$

$$\begin{aligned}
\mathsf{Either\ a\ b} &\cong \mathsf{Either\ b\ a} && (2.43) & \mathsf{(a, Either\ b\ c)} &\cong \mathsf{Either\ (a,b)\ (a,c)} && (2.48)\\
\mathsf{(a,b)} &\cong \mathsf{(b,a)} && (2.44) & \mathsf{Either\ a\ b} \rightarrow \mathsf{c} &\cong (\mathsf{a} \rightarrow \mathsf{c}, \mathsf{b} \rightarrow \mathsf{c}) && (2.49)\\
\mathsf{Either\ a\ Void} \cong \mathsf{Either\ Void\ a} &\cong \mathsf{a} && (2.45) & \mathsf{a} \rightarrow (\mathsf{b} \rightarrow \mathsf{c}) &\cong \mathsf{(a,b)} \rightarrow \mathsf{c} && (2.50)\\
\mathsf{(a,())} \cong \mathsf{((),a)} &\cong \mathsf{a} && (2.46) & \mathsf{Void} \rightarrow \mathsf{a} &\cong \mathsf{()} && (2.51)\\
\mathsf{(a,Void)} \cong \mathsf{(Void,a)} &\cong \mathsf{Void} && (2.47) & \mathsf{()} \rightarrow \mathsf{a} &\cong \mathsf{a} && (2.52)
\end{aligned}$$

Fig. 2.6: Commutative ring with identity laws applied to types

$$
\begin{aligned}
& \forall r.\mathsf{String} \to ((\mathsf{Err}, \mathsf{Either\ String}\ ()) \to r, (a, \mathsf{Either\ String}\ ()) \to r) \to r \\
= \quad & \{ \text{ apply eq. (2.48) on both continuations } \} \\
& \forall r.\mathsf{String} \to (\mathsf{Either}\ (\mathsf{Err}, \mathsf{String})\ (\mathsf{Err}, ()) \to r, \mathsf{Either}\ (a, \mathsf{String})\ (a, ()) \to r) \to r \\
= \quad & \{ \text{ apply eq. (2.46) on both continuations } \} \\
& \forall r.\mathsf{String} \to (\mathsf{Either}\ (\mathsf{Err}, \mathsf{String})\ \mathsf{Err} \to r, \mathsf{Either}\ (a, \mathsf{String})\ a \to r) \to r \\
= \quad & \{ \text{ apply eq. (2.49) twice and associativity of product } \} \\
& \forall r.\mathsf{String} \to ((\mathsf{Err}, \mathsf{String}) \to r, \mathsf{Err} \to r, (a, \mathsf{String}) \to r, a \to r) \to r \\
= \quad & \{ \text{ apply eq. (2.50) five times } \} \\
& \forall r.\mathsf{String} \to (\mathsf{Err} \to \mathsf{String} \to r) \to (\mathsf{Err} \to r) \to (a \to \mathsf{String} \to r) \to (a \to r) \to r
\end{aligned}
$$

The result of these transformations is a type that has switched the four return values for four continuations: here no allocation or four-way case splits are necessary to implement the combinators, and this makes them more amenable to inlining and optimisation. In fact, this is roughly the type of the Parsec monad [Leijen and Meijer 2001], except that the lack of String in the non-consuming continuations introduces a space leak that can be fixed by adding in more String values, using a similar calculation using Either String instead of Maybe, gives:

$$
\begin{aligned}
\textbf{type}\ \mathsf{Parser}\ a \ = \ & \forall r.\mathsf{String} \to (\mathsf{Err} \to \mathsf{String} \to r) \to (\mathsf{Err} \to \mathsf{String} \to r) \\
& \to (a \to \mathsf{String} \to r) \to (a \to \mathsf{String} \to r) \to r
\end{aligned}
$$

Two-continuation approach In practice, the two success continuations, `cok` and `eok`, will usually be the same as each other, except for the specific onward continuations they call. Except for those that arise from ($\langle|\rangle$), the same is true for the failure continuations, `cerr` and `eerr`. This repeated code can cause problems for programs that inline heavily, where the code size would double every time a recursive call is made. This is a problem for `parsley` due to its use of metaprogramming (§2.4). Instead, by further manipulation, the two good paths and two bad paths can each be merged into a single continuation:

$$
\begin{aligned}
\textbf{type}\ \mathsf{Parser}\ a = \quad & \{ \text{ apply eq. (2.50) five times } \} \\
& \forall r.\mathsf{String} \to ((\mathsf{Err}, \mathsf{String}) \to r, (\mathsf{Err}, \mathsf{String}) \to r, (a, \mathsf{String}) \to r, (a, \mathsf{String}) \to r) \to r \\
= \quad & \{ \text{ apply } (a, a) \cong \mathsf{Bool} \to a \text{ twice (this follows from } xx = x^2) \ \} \\
& \forall r.\mathsf{String} \to (\mathsf{Bool} \to ((\mathsf{Err}, \mathsf{String}) \to r), \mathsf{Bool} \to ((a, \mathsf{String}) \to r)) \to r \\
= \quad & \{ \text{ apply eq. (2.50) three times } \} \\
& \forall r.\mathsf{String} \to (\mathsf{Bool} \to \mathsf{Err} \to \mathsf{String} \to r) \to (\mathsf{Bool} \to a \to \mathsf{String} \to r) \to r
\end{aligned}
$$

Here the Bool value for each of the two continuations represents whether input has been consumed during the execution of the parser. It is not necessary to pass this explicitly, however, if there is a cheap $O(1)$ way of establishing this from the input.

2.4 Principled Metaprogramming

Staged metaprogramming is a technique for performing principled code generation [Sheard and Peyton Jones 2002; Stucki, Biboudis, and Odersky 2020; Taha and Sheard 1997; Nielson and Nielson 1992]. In this model, code is a regular first-class value in the host language, Code a, and functions that produce Code a can be run at compile-time to synthesise a value of type a to place directly into the compiled code; this is a two-stage approach [Nielson and Nielson 1992] where the first stage is compile-time and the second is runtime. Multi-stage programming, where code can also be generated at runtime, is possible [Taha 2004; Taha and Sheard 1997], and is supported in Scala 3 [Stucki, Biboudis, and Odersky 2020]; however, it is broadly out of scope for this dissertation, as *Typed Template Haskell* [Sheard and Peyton Jones 2002] (TTH), the framework used to build `parsley`, does not support it, even if the facilities are theoretically available [Pickering 2021].

The "principled" means that the code written to perform staging is not only guaranteed to be *well-levelled*[8], but also well-typed. With *Untyped Template Haskell*, usually known as just *Template Haskell* (TH), it is possible to write functions that return ill-typed code, or even code that refers to variables that are not in scope – errors are produced from the generated code itself when it is compiled. While this does have its uses (see sections 3.2.4, 7.2.1, and 7.2.3), it is not an ideal environment to develop more intricate systems.

The primitives of staging TTH is a principled metaprogramming framework, and foregoes the use of raw combinators opting instead for a pair of two operators, which are well-typed:

$$(\$(\cdot)) :: \mathsf{Code\ a} \to \mathsf{a} \qquad\qquad ([\![\cdot]\!]) :: \mathsf{a} \to \mathsf{Code\ a}$$

The operation $([\![\cdot]\!])$, called "quote," represents the AST of some value of type a exactly as written: this is the way of creating code that can be then manipulated as a first-class value. The operation $(\$(\cdot))$, called "splice" or "anti-quote," is used to inject the AST for some value of type a into a surrounding context: this is usually a quote, where this will place an AST into the branches of a larger compound AST. However, $(\$(\cdot))$ can also be used once at the very outer-most part of the staging, where it signifies to the compiler that it should run the code inside to generate some value that is to be placed directly into the to-be-compiled code.

An example This can be difficult to visualise without an example. Traditionally, the "hello world" of staged metaprogramming is the power function for exponentiation: however, a more relevant example can be found with parser combinators, the exactly combinator, which can be naïvely written as follows:

```
exactly :: Int → Parser a → Parser [a]
exactly 0 _ = pure [ ]
exactly n p = p <:> exactly (n − 1) p
```

This combinator will perform a parser a specified number of times, collecting up all the results into a list. But suppose a parser writer wishes to parse something exactly twice, whilst they could use exactly 2 for this, there would be a small overhead from having to perform the recursion, subtraction, and pattern matching as the combinator is traversed. Instead, they might write this combinator instead:

```
twice :: Parser a → Parser [a]
twice p = p <:> p <:> pure [ ]
```

[8]This is not too important, since all the code in the dissertation is well-levelled, but effectively it means that a value is always used in the stage in which it was originally defined, also known as well-staged though Pickering [2021] points out that is a conflation of terminology.

What if they now want a thrice, and so on? Already they have had to duplicate the logic of this parser once: this is where staged metaprogramming comes in. The first step is to identify what makes it possible to specialise; in this case, if the number of times the parser should be performed is known up-front (*statically*), the combinator can be unrolled to specialise it, taking the specific parser later at runtime (*dynamically*). With staged code, *static* values are regular types, and *dynamic* values are wrapped in Code:

```
exactlyS₁ :: Int → Code (Parser a → Parser [a])
exactlyS₁ 0 = ⟦λ_ → pure [ ]⟧
exactlyS₁ n = ⟦λp → p <:> $(exactlyS₁ (n − 1)) p⟧
  -- in TTH, the top-level splice MUST be kept in a separate file
twice :: Parser a → Parser [a]
twice = $(exactlyS₁ 2)
      = λp₁ → p₁ <:> (λp₂ → p₂ <:> (λ_ → pure [ ]) p₂) p₁
```

The base case for exactlyS_1 shows a plain quote representing the function const (pure [ ]); in the recursive case, a splice is used to graft the function generated by the n − 1 case into an application of the (<:>) combinator to the parser – because it returns a function, the argument p must be applied to the spliced function explicitly. The twice function is now in terms of a top-level splice of the exactlyS_1 2 code, which results in the given code in the body of the function during the remainder of the compilation. Quotes and splices do not normalise: this makes the generated function body messy. Instead, the function can be refined by breaking apart the dynamic generated function into a static function on dynamic values, resulting in exactlyS_2:

```
exactlyS₂ :: Int → Code (Parser a) → Code (Parser [a])
exactlyS₂ 0 _  = ⟦pure [ ]⟧
exactlyS₂ n qp = ⟦$qp <:> $(exactlyS₂ (n − 1) qp)⟧
twice :: Parser a → Parser [a]
twice p = $(exactlyS₂ 2 ⟦p⟧)
        = p <:> (p <:> pure [ ])
```

By breaking apart the structure of the result function statically, the application of function to value is now a purely compile-time job, and the generated code is – for now – as simple as it gets. This highlights the key idea of staging: identify static information and exploit it to remove runtime work at compile-time.

Note that functions can be converted between dynamic and static representations as follows:

```
staFun ::  Code (a → b) → (Code a → Code b)
staFun qf qx = ⟦$qf $qx⟧
dynFun :: (Code a → Code b) → Code (a → b)
dynFun f = ⟦λx → $(f ⟦x⟧)⟧
```

These functions do not form an isomorphism however, because the dynFun direction will always introduce an extra lambda: the two functions compose to create η-expansion instead.

A note on syntax The presented syntax for the quote and splice operations is unfortunately overloaded. In fact, in real code, the syntax shown has been that of the untyped variant:

```
-- untyped quotes and splices                -- typed quotes and splices
($(·)) :: Q Exp -> a                         ($$(·)) :: Code Q a -> a
([|·|]) :: a -> Q Exp                        ([||·||]) :: a -> Code Q a
```

While this is unfortunate, the convention will be to use $(\llbracket\cdot\rrbracket)$ to represent `([||·||])` and $(\llbracket\cdot\rrbracket_e)$ to represent `([|·|])`, or more precisely the syntax `([e|·|])` for untyped expression quoting. Splice is overloaded but is clear from context. For simplicity Code Q a will be denoted as Code a: Q is not itself important.

2.4.1 The Distinction between *Static* and *Dynamic* Information

The exactly example introduced the idea of creating a separation between *static* and *dynamic* information. The difference between the two terms boils down to when the information becomes known and exploitable: *static* information is something that can be known and reasoned about just from looking at the program; *dynamic* information is something that can only be reasoned about whilst the program is running. The art of staging is being able to properly separate the static from the dynamic, allowing for the exploitation of static information ahead of time. The tricky bit is avoiding pathological interactions with static and dynamic information: static information within dynamic information is still useless to the metaprogrammer. A key consideration is improving the *binding-time* of data ensuring static information is freely manipulatable independent of dynamic information.

Anything expressible at the type-level is trivially static information, as all types are known at compile-time. The stronger the type, the more it can be exploited. Usually, dynamic information is the stuff that makes the program "interesting," providing freedom to the user, but takes power away from the metaprogrammer: a careful balance is needed to be most effective. Consider the exactly example, where knowing the number of iterations up-front allowed for the specialisation of the combinator, at the cost of flexibility for the user to determine this iteration number at runtime.

Exploiting Static Knowledge

Domain-specific knowledge is another kind of static information available to the metaprogrammer. This is something that the programmer always knows about a program, even if a compiler does not. As an example, consider again the generated result of exactlyS_2 2:

twice p = p ‹:› p ‹:› pure [ ]

Simplifying this expression using the definition of (‹:›), liftA2, and eq. (2.7) from §2.2.2 gives:

twice p = pure (:) ‹∗› p ‹∗› (pure (:) ‹∗› p ‹∗› pure [ ])

Repeated application of the laws in fig. 2.2 establishes a normal form for applicative combinators where there is precisely one pure in any stretch of linear applicative combinators, known as applicative fusion [Delbianco, Jaskelioff, and Pardo 2012; Kiss, Pickering, and Wu 2018]. This is not the case for the twice, so wasted work is still being done: by applying the applicative laws, a more efficient one is derived:

```
twice p = (pure (:) <*> p) <*> ((pure (:) <*> p) <*> pure [ ])
        =   { by eq. (2.5) on _ <*> pure [ ] }
          (pure (:) <*> p) <*> (pure ($ [ ]) <*> (pure (:) <*> p))
        =   { by eq. (2.2) via eq. (2.7) on right-hand parser }
          (pure (:) <*> p) <*> (pure (($ [ ]) · (:)) <*> p)
        =   { simplify }
          (pure (:) <*> p) <*> (pure (: [ ]) <*> p)
        =   { by eq. (2.6) }
          ((pure (·) <*> (pure (:) <*> p)) <*> pure (: [ ])) <*> p
        =   { by eq. (2.2) via eq. (2.7) }
          ((pure ((·) · (:)) <*> p)<*>) <*> p
        =   { by eq. (2.5) on _ <*> pure (: [ ])) }
          (pure ($ (: [ ])) <*> (pure ((·) · (:)) <*> p)) <*> p
        =   { by eq. (2.2) via eq. (2.7) }
          pure (($ (: [ ])) · ((·) · (:))) <*> p <*> p
        =   { simplify }
          pure (λx y → x : y : [ ]) <*> p <*> p
```

This definition is more efficiently creating the list in place with all the results of the parsers in one lambda. In fact, the general optimised shape of exactly n is an n argument lambda taking x_1 through x_n and making the list $[x_1 \mathinner{..} x_n]$. The application of the applicative and functor laws to derive this more efficient implementation is static: no knowledge was required about the nature of the dynamic parser p in the derivation. This means that, in principle, this information can be used to generate the more efficient term from the outset.

Very strong types Encoding an arity-*n* function using TTH involves significantly more advanced types. Compared with plain TH, the exact type of the code needs to be always known, and the statically known iteration count must be reflected at the type-level to help retain this: this kind of masochistic use of Haskell is lovingly known as *Hasochism* [Lindley and McBride 2013]:

```
data Nat = Z | S Nat
data SNat (n :: Nat) where
  Zero :: SNat Z
  Succ :: SNat n → SNat (S n)
type family Arity (n :: Nat) a r where
  Arity Z     _ r = r
  Arity (S n) x r = x → Arity n x r
```

The SNat type is what is known as a *singleton*, which mirrors type-level information at the value-level, in this case it is providing a runtime witness of a type-level natural number of *kind* Nat[9]. The *type family* [Schrijvers et al. 2008] Arity can be thought of as a function on types: Arity Z a r is a zero-argument function returning

[9]Here, the `DataKinds` extension is being used to make a value-level type become a type-level kind.

type r; Arity (S Z) a r is a function from a → r; and so on with repeated a parameters. This is the Hasochistic machinery required to encode the new staged exactly combinator. Most of the work is done by mkListFunc:

```
mkListFunc :: ∀a n.SNat n → Code (Arity n a [a])
mkListFunc = go [ ] where
  go :: ∀n.[Code a] → SNat n → Code (Arity n a [a])
  go args Zero     = foldl′ (λqacc qv → ⟦$qv : $qacc⟧) ⟦[ ]⟧ args
  go args (Succ n) = ⟦λx → $(go (⟦x⟧ : args) n)⟧
```

Given a singleton SNat n, mkListFunc will produce the code for an a → ... → [a] function of arity n: it threads a list of the representations of each argument generated by each lambda and then collapses that into a sequence of n (:) cells on the empty list for the arity Z function. With this in place, the exactlyS_3 combinator is:

```
exactlyS₃ :: ∀a.Int → Code (Parser a) → Code (Parser [a])
exactlyS₃ n qp = go n Zero
  where
    go :: ∀n.Int → SNat n → Code (Parser (Arity n a [a]))
    go 0 sn = ⟦pure $(mkListFunc @a sn)⟧
    go n sn = ⟦$(go (n − 1) (Succ sn)) <*> $qp⟧
twice p = $(exactlyS₃ 2 ⟦p⟧)
        = pure (λx₁ → λx₂ → x₁ : x₂ : [ ]) <*> p <*> p
```

The exactlyS_3 combinator works by constructing a singleton to witness the arity of the function as it recurses; this means that when the base case of 0 is reached, the singleton will have reflected the value-level Int into the type-level, allowing for the function of the right arity to be generated for the unwinding of the recursion. At the outer-most call, the result of go is Code (Parser (Arity Z a [a])) which reduces to the required Code (Parser [a]).

Cross-Stage Persistence: `Lift`

Usually, dynamic and static data must be kept separate, and static information cannot become dynamic and vice-versa. The exception is for so-called *liftable* types; these are types a for which there is a Lift a instance:

```
class Lift a where
  lift :: a → Code a
```

This is a simplified view of Lift, showing only the portions of the class that are relevant. The lift function is the function that can take static information and persist it until runtime: this is known as *cross-stage persistence* [Taha and Sheard 1997]. Many types can be persisted in this way, so long as they can be inspected: notably, this means functions cannot be lifted. An example instance for the Nat type defined in §2.4.1 would be:

```
instance Lift Nat where
  lift Z     = ⟦Z⟧
  lift (S n) = ⟦S $(lift n)⟧
```

This is very mechanical, so it is possible to use the `DeriveLift` language extension to have the compiler derive the instances. Often lift is applied implicitly by GHC when necessary to make the code well-levelled.

Continuation-Passing Style

One of the core ways of improving the *binding-time* of information within a staged program is to make use of *Continuation-Passing Style* (CPS) [Steele and Sussman 1976; Reynolds 1993; Steele 1978]: this allows static information that is locked underneath dynamic constructs to be passed forward, unlocking more potential for exploitation. Note that static information can never be lifted through dynamic information. A good example of this [Consel and Danvy 1991; Bondorf 1992] is with conditional statements:

$$\begin{aligned}&\mathsf{mkIf} :: \mathsf{Lift\ a} \Rightarrow \mathsf{Code\ Bool} \to \mathsf{a} \to \mathsf{a} \to \mathsf{Code\ a}\\ &\mathsf{mkIf\ qc\ sx\ sy} = [\![\ \mathbf{if}\ \ \$\mathsf{qc}\ \mathbf{then}\ \mathsf{sx}\ \mathbf{else}\ \mathsf{sy}]\!]\\ &\mathsf{example\ qc} = [\![\$(\mathsf{mkIf\ qc}\ 5\ 6) + 3]\!]\end{aligned}$$

The mkIf function constructs a dynamic conditional statement with static outcomes. This is an example of where static information may be hiding underneath dynamic information; examine the example function, which adds three to the result of the dynamic conditional. The outcomes of the conditional are statically known, so ideally this can be optimised by constant-folding, but the dynamic information is getting in the way.

Applying the CPS transformation to the mkIf function, however, will change the binding-time properties of the function and allow the static information to escape temporarily: whatever happens next must occur under the quote, so a dynamic value is still necessarily produced – hence temporarily. The classic CPS transform must change shape slightly to accommodate this, so that it returns a Code r instead of an unconstrained r. The transformed function is as follows:

$$\begin{aligned}&\mathsf{mkIf}' :: \mathsf{Code\ Bool} \to \mathsf{a} \to \mathsf{a} \to (\mathsf{a} \to \mathsf{Code\ r}) \to \mathsf{Code\ r}\\ &\mathsf{mkIf}'\ \mathsf{qc\ sx\ sy\ k} = [\![\ \mathbf{if}\ \ \$\mathsf{qc}\ \mathbf{then}\ \ \$(\mathsf{k\ sx})\ \mathbf{else}\ \ \$(\mathsf{k\ sy})]\!]\\ &\mathsf{example}'\ \mathsf{qc} = \mathsf{mkIf}'\ \mathsf{qc}\ 5\ 6\ (\lambda \mathsf{x} \to \mathsf{lift}\ (\mathsf{x} + 3)) \quad \text{-- } \textit{equivalent to}\ [\![\ \mathbf{if}\ \ \$qc\ \mathbf{then}\ 8\ \mathbf{else}\ 9]\!]\end{aligned}$$

By introducing a continuation into mkIf′, the static information in each branch can be potentially combined with more static information from the continuation to form an optimised branch. In the improved example′, the fully static nature of the continuation is directly exposed, and the lifted result of adding three to each branch is injected directly into the branch. There are fewer quotations and splices in the new example′: a side-effect of the CPS that typically fewer quotes are needed, which is precisely demonstrating an improvement in binding-time.

"The Trick"

While it seems intuitive enough that dynamic data cannot be known at compile-time, this does not have to be the case, though it may be costly. In meta-programming folklore, the process of coercing dynamic data into static data is known as "The Trick" [Danvy, Malmkjær, and Palsberg 1996; Jones, Gomard, and Sestoft 1993].

The idea is simple: if all the possible values a dynamic value can take can be statically known, a runtime case analysis can be performed, yielding a known static outcome for each branch of the case split. As an example, here is "The Trick" applied to Bool:

```
boolTrick :: Code Bool → (Bool → Code r) → Code r
boolTrick qb k = ⟦ if $qb then $(k True) else $(k False)⟧
```

Notice the use of CPS (§2.4.1) in boolTrick: this is necessary since a Code Bool → Bool is ill-staged. While the Code Bool can be refined into a Bool value, it does so underneath a quotation, and therefore any processing done on that static information must again result in dynamic information being spliced. As such, it is not a magic way of breaking the stage restriction, but instead a way of temporarily circumventing it.

The true cost of "The Trick" comes from the generation of the case-split, along with the sheer amount of code that may be generated along each branch. For smaller types or possible sets of values "The Trick" can be an effective way of statically refining information.

The "Monadicrux"

The `parsley` library is a "*staged selective* parser combinator library"; staging is used to unlock domain-specific optimisation and improve performance, and selectives provide the backbone of the combinators that can be used to write parsers. However, most parser combinators libraries are monadic, so why selective? The problem lies in the type of the (»=) combinator (when specialised to Parser):

```
(»=) :: Parser a → (a → Parser b) → Parser b
```

Note that the static information used to optimise a parser is the structure of the parser itself, and the input and result are the dynamic information. To that end, the function (a → Parser b) must operate on dynamic values of type a, or must itself be a dynamic function:

```
Parser a → (Code a → Parser b) → Parser b
Parser a → Code (a → Parser b) → Parser b
Parser a → (Code a → Code (Parser b)) → Parser b
```

The first definition is already suspicious, as it is refining static information from dynamic information; and the second and third definitions produce dynamic parsers. Since the structure of the next parser cannot be known until runtime when the result is produced after processing input, it is not possible to compile (»=) ahead-of-time. In languages with runtime staging [Stucki, Biboudis, and Odersky 2020], like Scala, this obstacle can be overcome by performing code-generation at runtime, but the performance of such operations would be poor.

Why do selectives fare better? Selective functors (§2.2.4) still have an interaction with dynamic information: branch me mx my will only know which of mx and my to execute dynamically– when the result of me is known at runtime – but the outcomes mx and my are still statically known. The limitation is that static analysis on selectives is approximate since the precise outcome is not statically known.

Selectives and "The Trick" A connection can be made between selective functors and "The Trick" (§2.4.1) in the way that selectives can be used to create a static (»=) combinator. This combinator was identified by Mokhov et al. [2019] by the name of sbind:

```
sbind :: (Selective f, Eq a, Bounded a, Enum a) ⇒ f a → (a → f b) → f b
```

The definition is omitted for brevity, but each of the possible values within [minBound . . maxBound] is tested for equality against the value, using branch as many times as required to perform the case-splitting. Like the metaprogramming version of "The Trick," the selective sbind is refining a purely static program structure from what is otherwise a dynamically structured program.

2.4.2 Partially-Static Data

Not all data must be one of static or dynamic: it is possible for something to be occasionally static and occasionally dynamic, or a mix of both kinds of components. This is the essence of *partially-static data* [Yallop, Glehn, and Kammar 2018]. At its simplest, all that is required is to switch from just using Code to using a type like PS:

```
data PS = Sta a | Dyn (Code a)

toCode :: Lift a ⇒ PS a → Code a
toCode (Dyn q) = q
toCode (Sta x)   = lift x
```

By working with this datatype, it is already possible to build more interesting operations. For example:

```
addPS :: PS Int → PS Int → PS Int
addPS (Sta x) (Sta y) = Sta (x + y)
addPS (Sta 0) py      = py
addPS px      (Sta 0) = px
addPS px      py      = Dyn ⟦$(toCode px) + $(toCode py)⟧
```

The addPS function allows for the combining of static values to create more static values, preserving their usefulness for as long as possible. It also applies an optimisation when one value is known to be 0. However, for anything that involves dynamic information, dynamic information must be returned. This is far from perfect, however; consider the following examples:

```
example = toCode ((Dyn qx `addPS` Sta 6) `addPS` Sta 4)
          = ⟦$qx + 6 + 4⟧
example′ = toCode (Dyn qx `addPS` (Sta 6 `addPS` Sta 4))
          = ⟦$qx + 10⟧
```

Depending on the associativity of the application of addPS, the static values will either reduce, or will be irreducible because one has been combined with a dynamic value. Worse would be if the dynamic value were the middle argument to the call: in this case, either associativity would produce the optimal form. Yallop, Glehn, and Kammar [2018] suggest creating more specialised partially-static datatypes that additionally leverage domain-specific laws; in this case, addition is commutative and associative – this can be used to produce a partially-static structure that normalises:

```
data PS a = PS [Code a] a

fromDyn     :: Monoid a ⇒ Code a → PS a
fromDyn qx = PS [qx] mempty
fromSta       :: a → PS a
fromSta x     = PS [ ] x

toCode :: (Lift a, Monoid a) ⇒ PS a → Code a
toCode (PS [ ] x) = lift x
toCode (PS vs@(_ : _) x)
    | x ≡ 0         = qv
    | otherwise = ⟦$qv + x⟧
  where
    qv = foldr1 (λqx qy → ⟦$qx • $qy⟧) vs
```

This datatype represents a partially static commutative monoidal expression as a pair of a list of dynamic irreducible terms, and a single reduced static term. A bag-like structure is more appropriate than a list in this case, but for simplicity a list suffices. To turn this structure into plain Code, the static constant can be checked to avoid redundant 0s in the result, and then the dynamic parts combined with a fold. By splitting apart static and dynamic information in this way, it is possible to make a new version of addPS that can always optimise all static information in this domain perfectly:

```
addPS :: Monoid a ⇒ PS a → PS a → PS a
addPS (PS vs₁ x) (PS vs₂ y) = PS (vs₁ ++ vs₂) (x • y)

example = toCode ((fromSta 4 `addPS` fromDyn qx) `addPS` fromSta (negate 4))
          = qx
```

The definition of addPS is simpler now than before: it merely suffices to append the dynamic components together and merge the static components using (•). With both this and toCode, the expression `4 + x + -4` is optimised to `x`. Partially-static data is applied in CHAPTER 6: ANALYSIS AND OPTIMISATION to aid optimisation and allow for static data to be aggregated and exploited throughout the compilation of a parser.

2.5 Cayley Representations for Monoids

A Cayley transform on a data-structure is a form of optimisation for functional programs that is useful for avoiding pathologically left-associative operations: this is desirable if an operation is $O(n)$ in the left-hand argument. The two most common kinds of Cayley transform are on monoids and monads [Hinze 2012; Voigtländer 2008]: in this dissertation, the monoid variant is useful in a variety of places, but the monad variant is not required. Cayley representations will find use in CHAPTER 3: EMBEDDING A PARSER COMBINATOR LIBRARY.

2.5.1 Monoids

A monoid M is a type for which there is an identity element $e : M$ and a binary operation $\bullet : M \to M \to M$ subject to the following two laws:

$$e \bullet x = x = x \bullet e \qquad (2.53) \qquad\qquad (x \bullet y) \bullet z = x \bullet (y \bullet z) \qquad (2.54)$$

In Haskell, a monoid $(M, e, \bullet)$ is represented as an instance of the Monoid typeclass. Two instances relevant for this dissertation are the list monoid and the endofunction (or endomorphism) monoid:

```
class Monoid m where          instance Monoid [a] where     instance Monoid (a → a) where
  mempty :: m                   mempty = [ ]                  mempty = id
  (•)    :: m → m → m           (•)    = (++)                 (•)    = (·)
```

2.5.2 Cayley Transformation

Cayley [1854] gave a theorem that there is an isomorphism between groups and permutations of those groups. A permutation can be thought of as an endomorphism that maps elements to their new position in a permutation. This gives rise to an alternative formulation of the theorem [Rivas and Jaskelioff 2014], which says that every monoid M is isomorphic to a submonoid $M \to M$ of the endomorphism monoid. This applies to functional programming as follows:

```
newtype Endo a = Endo {unEndo :: a → a} deriving Monoid

toCayley :: Monoid m ⇒ m → Endo m
toCayley x = Endo (x •)

fromCayley :: Monoid m ⇒ Endo m → m
fromCayley (Endo cx) = cx mempty
```

The functions toCayley and fromCayley are monoid homomorphisms that map elements of a monoid onto a submonoid of the endomorphism monoid Endo and vice-versa. fromCayley · toCayley = id, since:

```
(fromCayley · toCayley) x = fromCayley (Endo (x •))
                          = x • mempty
                            -- by eq. (2.53)
                          = x
```

This establishes the isomorphism [Hinze 2012], assuming that the possible endomorphims provided to Endo are only constructed in a structured way with toCayley:

```
(toCayley · fromCayley) (Endo (x •)) = toCayley ((x •) mempty)
                                     = toCayley (x • mempty)
                                       -- by eq. (2.53)
                                     = toCayley x
                                     = Endo (x •)
```

The basic idea is to replace the use of mempty and (•) in the underlying monad by mempty and (•) in the Endo monoid. This has the effect of right-associating all the operations on the underlying monoid.

2.5.3 Difference Lists

The best-known example of a Cayley representation is the difference list [Hughes 1986]. This is useful because (++) is $O(n)$ in the left-hand list, and this causes a pathological $O(n^2)$ complexity for left-associated applications of (++): this can happen very easily when using (++) in recursive functions, where the associativity of the applications cannot be directly controlled. Instead, the Cayley transformation guarantees that the final list will be assembled in a right-associative way, in $O(n)$:

```
fromCayley $ (toCayley xs • toCayley ys) • toCayley zs
   = fromCayley $ (Endo (xs ++) • Endo (ys ++)) • Endo (zs ++)
   = fromCayley $ Endo ((xs ++) · (ys ++)) • Endo (zs ++)
   = fromCayley $ Endo ((xs ++) · (ys ++) · (zs ++))
   = ((xs ++) · (ys ++) · (zs ++)) [ ]
   = (xs ++) ((ys ++) ((zs ++) [ ]))
   = xs ++ (ys ++ (zs ++ [ ]))
```

Even though the original expression (dxs • dys) • dzs was left-associated, the natural right-associative application of (·) results in xs ++ (ys ++ zs) in the final term. To complete the list functionality, a function diffCons is required:

```
type Diff a = Endo [a]
diffCons :: a → Diff a → Diff a
diffCons x dxs = Endo (x :) • dxs
diffSnoc :: a → Diff a → Diff a
diffSnoc x dxs = dxs • Endo (x :)
```

Now all three basic list constructors: (:), (++), and [ ] can be represented in the Cayley representation Diff. The disadvantage of the Cayley representation when working with lists is that the intermediate list cannot be inspected during the construction process without using the $O(n)$ fromCayley function.

2.6 Structured Recursion Schemes

Folds have long been an important concept in functional programming. They are an abstraction for dealing with recursion over lists [Hutton 1999; Kleene 1952]. More generally, however, folds can be synthesised for almost any algebraic datatype: the recipe is to build a function where each constructor of the datatype is represented as a function passed as an argument to the fold – these form an *algebra*. This can be thought of as a function that evaluates a datatype using its Church-encoding [Church 1941], or more specifically its Böhm-Berarducci encoding [Böhm and Berarducci 1985], since it is typed. For example:

```
data List a   = Cons a (List a)     | Nil
foldList      :: (a → b → b)        → b        → (List a → b)
data Maybe a = Just a               | Nothing
foldMaybe     :: (a → b)            → b        → (Maybe a → b)
```

```
data Tree a   = Fork (Tree a) a (Tree a)   | Tip
foldTree      :: (b → a → b → b)          → b          → (Tree a → b)
data Tree′ a  = Branch (Tree a) (Tree a)  | Leaf a
foldTree′     :: (b → b → b)              → (a → b) → (Tree′ a → b)
```

Making a separate folding function for each datatype is tiresome, however. Instead, a generic fold can be produced that takes the algebra as a single entity [Hagino 1987; Hinze, Wu, and Gibbons 2013; Bird and Paterson 1999]. This algebra is usually a function that collapses a *syntactic functor* containing the results of recursive folds into a new value. To make use of this, datatypes must instead be expressed as the least fixed-point of some syntactic functor using the Fix datatype:

```
newtype Fix f = In { out :: f (Fix f) }
```

This datatype encodes general datatype recursion explicitly by wrapping nested structure in the In constructor. Given an algebra on the functor f producing a b, it is possible to generically fold a Fix f into a b:

```
cata :: Functor f ⇒ (f b → b) → Fix f → b
cata alg = alg · fmap (cata alg) · out
```

The function cata, called a catamorphism [Hagino 1987; Meertens 1988], first unwraps a layer of a recursive structure, applies itself recursively within the syntactic functor, and then applies the algebra to collapse the current layer fully. With this, it is possible to implement an entire host of other recursion schemes [Hinze, Wu, and Gibbons 2013]. As an example, here is the List datatype as a fixed-point of the ListF functor as well as the implementation of the fold function using cata:

```
data ListF a k = Cons a k | Nil deriving Functor
type List a = Fix (ListF a)
foldList :: (a → b → b) → b → List a → b
foldList cons nil = cata alg where alg (Cons x xs) = cons x xs
                                   alg Nil         = nil
```

The syntactic functor ListF represents the two constructors of lists but notably without recursion: instead, the parameter k represents a hole for a recursive list to go, or indeed the intermediate results of a fold; k will be filled in by List a by Fix to construct a value of the type. The foldList function can be implemented as a single cata, where the algebra is built by applying the provided functions under a pattern match on the syntax of the list.

2.6.1 Different Recursion Schemes

On its own, cata can handle any structurally recursive function on a data-structure. However, there are plenty of variations on this scheme [Yang and Wu 2022]: each is specialised to be more ergonomic for specific sub-patterns of recursion. These will appear in various chapters across this dissertation.

Mutumorphisms

The first specialised cata is the mutumorphism [Fokkinga 1989], which captures the pattern of mutually-recursive algebras:

```
mutu :: Functor f ⇒ (f (a, b) → a) → (f (a, b) → b) → Fix f → (a, b)
mutu algl algr = cata (algl △ algr) where (f △ g) x = (f x, g x)
```

It can be used when two algebras depend on each other's results, but each result can be produced individually.

Zipper For computations that can be done in parallel but do not otherwise depend on each other, the zipper scheme can be used to compute them together in one pass:

```
zipper :: Functor f ⇒ (f a → a) → (f b → b) → Fix f → (a, b)
zipper algl algr = mutu (algl · fmap fst) (algr · fmap snd)
```

This scheme very simply performs a mutumorphism but disassociates the correspondingly unneeded value before feeding the other to each algebra.

Histomorphisms

The second specialised cata is the histomorphism [Hinze and Wu 2013; Uustalu and Vene 1999], which captures the pattern of needing access to historic values within the fold:

```
data Cofree f a = a ◂ f (Cofree f a) deriving Functor
extract :: Cofree f a → a
extract (x ◂ _) = x
histo :: Functor f ⇒ (f (Cofree f a) → a) → Fix f → a
histo alg = extract · cata (λx → alg x ◂ x)
```

A histomorphism, called histo, stores every result alongside the value used to make it in a Cofree f structure (which is the regular co-free co-monad); this means the history of the computation is stored in the shape that produced it, which can be freely traversed if necessary. This is useful for light-weight dynamic programming where a computation may depend on sub-results, for example Fibonacci on natural numbers:

```
data NatF = Succ k | Zero
fib :: Fix NatF → Int
fib = histo alg where alg :: NatF (Cofree NatF Int) → Int
                      alg Zero                       = 1       -- fib 0 = 1
                      alg (Succ (_ ◂ Zero))          = 1       -- fib 1 = 1
                      alg (Succ (x ◂ Succ (y ◂ _))) = x + y   -- fib n = fib (n − 1) + fib (n − 2)
```

To identify the one case, it is necessary to peel back a layer of the computation history to establish whether the previous result was a zero. For the general case, two layers of the history are peeled back to reveal the fib (n − 1) and fib (n − 2) results, which can then be combined.

2.6.2 Joining Syntax

Syntax can be composed by using a coproduct functor [Kiss, Pickering, and Wu 2018; Magalhães and Löh 2014], which can be useful: this might be because some of the syntax is removed or added later, or just that a compositional semantics is desired.

```
data (f :+: g) k = L (f k) | R (g k) deriving Functor

( ▽ ) :: (f a → a) → (g a → a) → ((f :+: g) a → a)
(f ▽ _)  (L x)  = f x
(_ ▽ g) (R y) = g y
```

This functor f :+: g combines two functors f and g by allowing one of them to appear with the given continuation k depending on which of L or R is used. The operator (▽) allows for two algebras working on two independent functors to be combined to make an algebra that works on the co-product of these functors. With this in place, cata and the other schemes can be used normally, but the provided algebras will use (▽) where appropriate.

2.6.3 Indexed Recursion Schemes

When additional type information needs to be included in a syntactic functor, they need to be indexed [McBride 2011]. This also means that the folds over them must be aware of the indices and preserve them:

```
class IFunctor f where
  imap :: (∀i.a i → b i) → f a i → f b i

data IFix f i = In (f (IFix f) i)

icata :: IFunctor f ⇒ (∀i.f a i → a i) → IFix f i → a i
icata alg = let go (In x) = alg (imap go x) in go
```

Here, IFunctor's imap operation demands that the function that maps over the indexed values within the structure preserves the indices. These indexes must materialise in the IFix type, where the type i may change throughout a recursive structure. The icata function behaves much like cata, with an identical definition, but where the algebra must preserve the type indices for the layer it is working on. As with IFix, this index is allowed to vary for different calls to alg, so the type is universally quantified within the function.

To keep things simple, the indices will be ignored in the implementations of algebras, as they are always preserved anyway. The type Fix, and each of the recursion schemes will be overloaded to work on any number of type indices. The indexed Const functor will not be required to wrap non-indexed types that result from folds – this is unnecessarily noisy boilerplate. However, to avoid confusion, use of the Const functor will be marked with a commented Const in the type signature.

Chapter 3

Embedding a Parser Combinator Library

The following chapter has been adapted from both "Staged Selective Parser Combinators" [Willis, Wu, and Pickering 2020] (in collaboration with Nicolas Wu and Matthew Pickering) and "Oregano: Staging Regular Expressions with Moore Cayley Fusion" [Willis, Wu, and Schrijvers 2022] (in collaboration with Nicolas Wu and Tom Schrijvers).

As opposed to many classic parser combinator libraries, `parsley` is a domain-specific language [Fowler 2010; Hudak 1996] implemented as a *deep embedding* [Gibbons and Wu 2014], which means that parsers are pure data that is interpreted with various semantics. This allows for parsers to be inspected, which facilitates analysis and optimisation – discussed more in CHAPTER 6. To avoid the overheads of the deep embedding, `parsley` uses staging to generate Haskell code of hand-written quality. As an example, consider the following example identifier parser:

```
identifier :: Parser String
identifier = filterS ⟦∉ keywords⟧ (letter <:> many (letterOrDigit <|> char '_'))
```

This parser reads an alphabetical character, followed by zero or more other identifier characters, but ensures that the result is not a keyword in the language. There are two things of note here: firstly, a staged parser combinator library exposes Code into its high-level API, seen here with ⟦∉ keywords⟧, as user-defined predicates and values interact with dynamic input; and the filterS combinator is used, which means that `parsley` has a selective API. This is another artefact of the staging, since the parser must be compiled statically, a monadic API is not possible (§2.4.1), but selectives can fill the gap. In this case, selectives provide lightweight context-sensitivity that allows for contextual validation – like checking for a keyword. The result of feeding this parser through `parsley`'s machinery would be something like the following, with some details omitted for clarity:

```
ident input = $(parse identifier ⟦input⟧)
            = let loop (c : cs) dxs finish | isAlpha c ∨ isDigit c ∨ c ≡ '_' = loop cs (dxs · (c:)) finish
                  loop cs dxs finish = finish (dxs [ ]) cs
              in case input of
                  c : cs | isAlpha c → loop cs id $ λxs _ →
                     if (c : xs) ∉ keywords then Just (c : xs) else Nothing
                  _ → Nothing
```

The resulting code does not have any evidence of the high-level abstractions of the combinators, and compiles to a tight loop leveraging a difference list to collect the results of the many (§2.5.3).

Representing parsers Whilst the datatype – or AST – for parsers aligns closely with the high-level combinators, such a structure does not have the best shape for exploiting static information via staging: a CPS-like structure provides an improvement to binding-time and exposes control-flow properties of the parser more clearly [Kennedy 2007], but loses scoping and high-level information about the combinators (§2.4.1). An aim of this chapter is to provide a derivation of the low-level machinery that the combinator tree is translated into (§3.1).

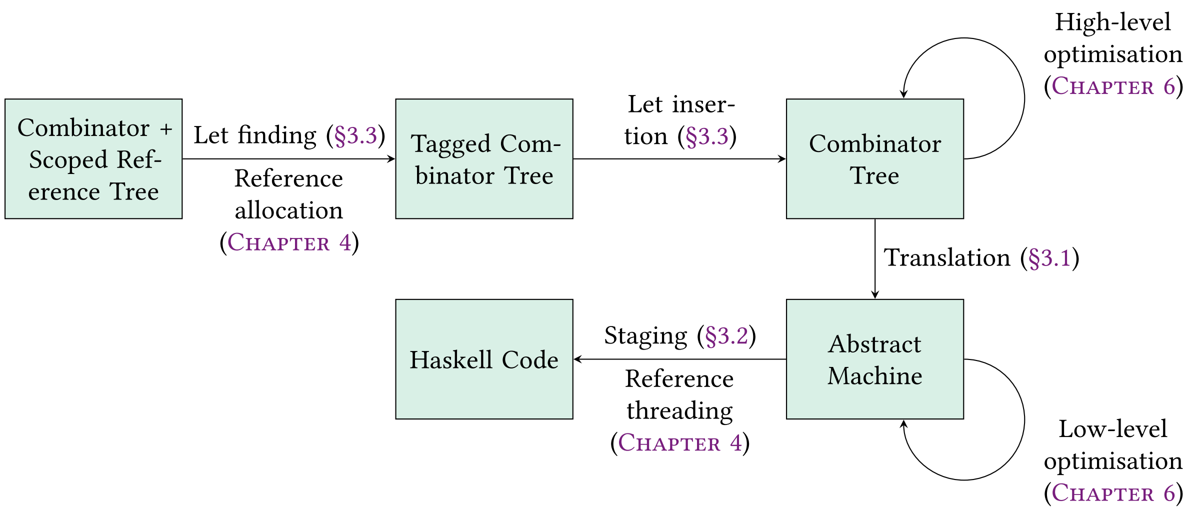

Combinator + Scoped Reference Tree
Let finding (§3.3)
Reference allocation (CHAPTER 4)
Tagged Combinator Tree
Let insertion (§3.3)
Combinator Tree
High-level optimisation (CHAPTER 6)
Translation (§3.1)
Abstract Machine
Low-level optimisation (CHAPTER 6)
Staging (§3.2)
Reference threading (CHAPTER 4)
Haskell Code

Fig. 3.1: `Parsley`'s compilation pipeline

Staging machinery As mentioned, `parsley` uses staged metaprogramming (§2.4) to help optimise its parsers and produce fast code of hand-written quality. To achieve this, it is necessary to establish what aspects of parsers can be leveraged as static information and incorporate the right binding-time distinctions into both representations of parsers. The calculated machine is evaluated, and this is then staged, eliminating the static information (§3.2).

Observing recursion and let-bindings Users of parser combinator libraries expect to be able to write parsers in a natural, direct style, where the recursion machinery of Haskell can be used to define parsers. The discussion in §3.1 will, for simplicity, assume that the API encodes recursion explicitly using special combinators. Instead, it is possible to make the implicit recursion in Haskell observable using *let-finding analysis*, which is employed by `parsley` to factor out common parsers and handle recursion without further user intervention (§3.3).

3.1 From Combinators to Automata

Traditionally, the various levels of expressiveness for languages in the Chomsky hierarchy [Chomsky 1956] are each mapped onto a specific kind of automaton [Sipser 2013]. The representation of automata in Haskell has a naturally CPS-like structure, which makes the control-flow properties of an automaton easy to analyse. Since this is desirable for staging (§2.4.1) and facilitates control-flow analysis and optimisation, it also makes sense to use a finite-state automaton for the intermediate representation of parser combinators. This section demonstrates the conversion from combinators to automata in an incremental way, starting with a small subset of the combinators, corresponding to regular expression recognisers, and working up to supporting the full set of `parsley` primitive combinators. It shows how the transformation from combinator to automata is an instance of the Cayley transformation (§2.5.2) demonstrating this as a calculation [Bahr and Hutton 2015; Pickard and Hutton 2021; Bahr and Hutton 2022], with the compile function as a by-product:

- First, the relationship between regular expressions and NFAS (§2.1.3) is recapped, representing both as Haskell datatypes and showing their relation via a Cayley transformation (§3.1.1).

- Then, the automaton is expanded to incorporate the output characteristics of a Moore machine (§2.1.3), allowing for regular expression parsers returning homogeneous results as opposed to recognisers (§3.1.2).
- To reach full parser combinators, the results of the automaton are made heterogeneous in a parametricity-preserving way – introducing extra type-indices to track the in-flight results (§3.1.3).
- Regular expressions are not powerful enough to parse many real-world languages, and commonly context-free power is expected. The language will be expanded to include explicitly annotated recursion, additionally requiring a shift to a push-down automata (§2.1.3) inspired extension of the Moore machine (§3.1.4).
- There is no backtracking for non-deterministic push-down automata, and the input can only be consumed in one go, albeit non-deterministically. This is not the semantics of PEG (§2.1.2), which only allows for taking the second branch of a choice when the first fails; nor is it the semantics of `parsec` (§2.3), which additionally requires that the first branch cannot consume input. Since `parsley` also adheres to this semantics, a notion of explicit backtracking must be introduced, which allows for the support of the PEG positive- and negative-lookahead operations as well in the form of look_+ and look_- (§3.1.5).
- Since `parsley` is a selective parser combinator library, it needs to support a limited form of context-sensitive parsing via the branch combinator. To do this, the final augmentation to the automata allows for the inspection of the top-most element on the output stack. This causes the final machine to brush against the Turing machine, which hints at its improved expressivity (§3.1.6).

3.1.1 Regular Expressions and Non-Deterministic Finite Automata

Regular expressions are a lightweight tool for parsing regular languages: they consist of a small handful of operations, namely Eps, which accepts the empty-language; Empt, which always rejects; Lit c, which recognises a character c; Cat r1 r2, which recognises the expression r1 followed by r2; Alt r1 r2, which recognises either the expression r1 or r2; and Rep r, which recognizes zero-or-more occurrences of the expression r. Importantly, with the exception of repetition, they cannot be recursive. This can be represented by the following *syntactic functor* (§2.6):

```
type Regex = Fix Reg
data Reg k where
  Eps  :: Reg k
  Empt :: Reg k
  Lit  :: Char → Reg k
  Cat  :: k → k → Reg k
  Alt  :: k → k → Reg k
  Rep  :: k → Reg k
```

This datatype is sufficient to express any regular expression *recogniser*, so long as care is taken to avoid writing recursive values. Smart constructors can be built to model regular expression operators directly:

```
eps = In Eps
cat r1 r2 = In (Cat r1 r2)
 ...
```

These functions are more convenient to work with and are used liberally in place of their data constructor counterparts throughout this section. As an example, take the regular expression $a(b|c)d$, which is expressed as:

```
ex1 = lit 'a' `cat` (lit 'b' `alt` lit 'c') `cat` lit 'd'
```

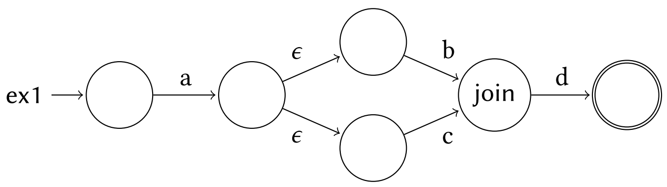
Representing Finite State Machines

There are two automata that precisely correspond to the regular class of languages: DFAs and NFAs (§2.1.3). Both are finite in the number of states that are used to encode an automaton, which means they can be represented as a finite value in Haskell. Of the two, NFAs are the more common, as they retain a much smaller state space, trading off for multiple in-flight execution paths. They can be represented by the following syntactic functor:

```
type NFA = Fix State
data State k where
  Accept :: State k
  Fail   :: State k
  Split  :: k → k → State k
  Item   :: Char → k → State k
```

The four constructors each map to an automaton "widget" (fig. 3.2): the Accept constructor is used to denote an accept state, which is the final state of an automaton denoting success – even if more input follows; Fail denotes a terminal branch of the automaton that does not accept, and cannot progress onwards to another state, killing a branch; Item c k denotes a transition from one state to another state k where the character c must be read from the input to traverse; and Split denotes a non-deterministic ϵ-transition to both of two states. Conspicuously, the shape of the state widgets is in full CPS-style, with the Item state explicitly requiring the next state as opposed to having a concatenation widget. The example from above can be encoded as an automaton as follows:

```
ex1 = let join = item 'd' accept in item 'a' (split (item 'b' join) (item 'c' join))
```



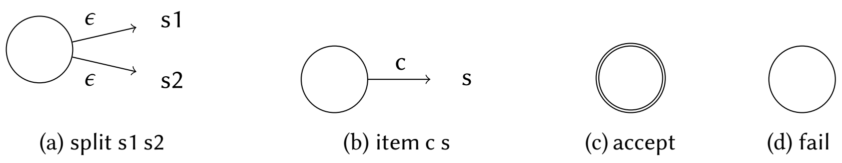
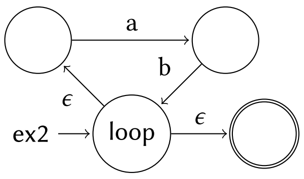


Fig. 3.2: Mapping from automata widgets to constructors

Here, the two states arising from the split are explicitly joined back into a single path using a Haskell let-binding; this stops an exponential explosion in the number of states resulting from every split. The use of explicit join-point also allows for this set of widgets to implement looping behaviours; consider the expression (ab)∗:

```
ex2 = let loop = split (item 'a' (item 'b') loop) accept in loop
```



Crucially, while the value is self-recursive, it is knot-tied: this means the value is finite in size, though infinitely traversable. Consider the difference between the two automaton combinators:

```
loopBad   :: (NFA → NFA) → NFA → NFA
loopBad r k   = split (r (loopBad r)) k
loopGood :: (NFA → NFA) → NFA → NFA
loopGood r k = let loop = split (r loop) k in loop
```

The loopGood combinator, which implements ex2 as loopGood (item 'a' · item 'b') accept, makes use of an explicitly knot-tied value loop and is finite. The loopBad combinator instead is a truly recursive combinator, and results in an infinite state repetition: operationally, this could be interpreted in the same way to achieve the same results, but it is not a finite state automaton. The end goal is to stage the evaluation of the final automaton, so it is particularly important that the finiteness is respected: it is not practical to generate an infinite amount of code[1].

Translation via Cayley Transformation

The connection between regular expressions and NFAs is well known [Sipser 2013; Hopcroft, Motwani, and Ullman 2001]. However, producing a conversion from one to the other is shown to be an instance of the Cayley transformation. Recall that a Cayley transformation maps a monoid's (•) operation onto function composition and the mempty value onto the identity function. In this case, the Regex type forms a monoid (quotiented by normalisation):

```
instance Monoid Regex where
  (•) = cat
  mempty = eps
```

This is one of two possible monoids for Regex, the other with alt and empt, however it is also the relevant one for the calculation of the compile function. First, apply the Cayley transformation to Regex:

```
cat :: Endo Regex
cat = (•)
```

[1]As it stands, it is not possible to generate code for these values either, as the cycles need to be observable – this will be explicitly introduced in §3.2.4, however implicit sharing keeps the discussion in this section clearer: regardless, these values are still finite.

```
eps :: Endo Regex
eps = mempty

empt :: Endo Regex
empt = Endo (λk → In Empt)

lit :: Char → Endo Regex
lit c = Endo (λk → In (Cat (In (Lit c)) k))

alt :: Endo Regex → Endo Regex → Endo Regex
alt (Endo r1) (Endo r2) = Endo (λk → In (Alt (r1 k) (r2 k)))
```

As per the transformation, cat and eps map to the monoid operations for the endomorphism monoid (specialised to Regex). Then, empt becomes a function that discards its argument (since it does not continue); lit concatenates the regular literal recogniser with its continuation k, and alt distributes a continuation k across two regular expressions. As usual, a Regex can be acquired from Endo Regex by feeding the In Eps value to the function.

Handling the Rep case is slightly trickier. In a parser combinator sense, repetition is usually implemented as a recursive combinator:

```
rep r = let loop = (r `cat` loop) `alt` eps in loop
```

This works as is, however for clarity, it can be massaged into the form:

```
rep r = Endo (λk → let loop = (r `cat` loop) `alt` eps in unEndo loop k)
```

This can be simplified further using the definitions above and equational reasoning to end up at a definition that does not refer to other Cayley transformed combinators:

```
rep (Endo r) = Endo (λk → let loop = In (Alt (r loop) k) in loop)
```

With these Cayley transformed combinators in hand, the next step is to eliminate Cat from the datatype entirely: this is only used for the definition of the lit combinator, and this could be removed by adding the continuation k straight onto the Lit AST node. In fact, noting that the rep node has also been eliminated from the datatype, the residual functor remaining is:

```
data RegCPS k where
  Eps'  :: RegCPS k
  Empt' :: RegCPS k
  Alt'  :: k → k → RegCPS k
  Lit'  :: Char → k → RegCPS k
```

This functor is isomorphic exactly to the State functor. In fact, the Cayley transformed combinators serve precisely as the corresponding compilation rule for each constructor. To make this concrete, a compile :: Regex → NFA function can be written as a single fold using cata:

```
compile :: Regex → NFA
compile = cata comp accept
```

```
  where comp :: Reg (NFA → NFA) → (NFA → NFA)
        comp Eps          = id
        comp (Cat r1 r2)  = r1 · r2
        comp (Lit c)      = item c
        comp Empt         = λk → fail
        comp (Alt r1 r2)  = λk → split (r1 k) (r2 k)
        comp (Rep r)      = loopGood r
```

Each of the clauses in the algebra comp correspond to the Cayley transformed combinator definition, but with the constructors replaced by their corresponding CPS equivalents.

3.1.2 Adding Results with Non-Deterministic Moore Machines

The Regex type is fundamentally a *recogniser* for regular expressions: it can report whether a string is in a given language, but not return any meaningful information about the derivation. The aim of this section is to refine this datatype so that it can return meaningful user-directed results.

Writing Results to a Tape

Moore machines (§2.1.3) are an extension of deterministic finite state automata (DFA) that write values onto a write-only tape on entry to a state. For the purposes of the overarching translation, this definition is altered slightly for convenience:

- A value does not need to be emitted on entry to every state, instead only on select states: implicitly allowing a unital element for the alphabet on the output tape that is emitted when no other value is desired.
- A transition to an outputting state must be an ϵ-transition, where no input is consumed.
- For notational convenience, values are emitted to the tape on the transition between two states as opposed to on the state itself: since the transition to an outputting state does not inspect the input, this does not mean the machine is a Mealy machine.

It is possible to define a non-deterministic variant by allowing a tape to duplicate when the machine non-deterministically makes a transition. The two tapes will be independently modified across the branches, which will mean that multiple complete tapes (of varying lengths) will be returned from a successful parse.

The addition of output adjusts the definition of the State syntactic functor defined in §3.1.1, where an Out widget is added to the existing machinery. In addition, the machine becomes parameterised by its tape alphabet:

```
type Moore a = Fix (State a)
data State a k where
   ...
  Out :: a → k → State a k
```

The Out constructor represents a transition to state where a given value of type a is emitted to the (current) tape.

Introducing Results to Regular Expressions

The introduction of result tapes to the State functor means that the Regex type can now support emitting values during parsing. For simplicity, this can be assumed to happen on the eps operation, where no input is consumed:

```
data Reg a k where
  Eps :: a → Reg a k
  ...
```

Now that Eps produces values, the comp algebra from §3.1.1 is adjusted to map Eps to an Out node instead of just the identity transition:

```
comp :: Reg a (Moore a → Moore a) → (Moore a → Moore a)
comp (Eps x) = out x
```

The rest of the algebra remains the same.

The Shape of the Results

While the new Moore machine allows for the generation of results within a parser, its utility is limited. Consider the evaluation function on values of type Moore a:

```
eval :: Moore a → String → [[a]]
```

This function (whose implementation is not important) takes a machine, and the input string to query and returns a list of tapes, which themselves are represented by lists. This is not a particularly useful result to return to the programmer, and it can be improved by introducing a deforestation:

```
eval :: (a → b → b) → b → Moore a → String → [b]
```

Now, the evaluation of a machine can produce results of a different type to the individually emitted results, so long as a function is given to combine emitted results with the remaining results, as well as a value to return on the Accept state. Still, while this allows the programmer some level of freedom to construct appropriate results, it is a far cry from the expressive power of applicative parser combinators, which would allow the results to be combined in arbitrary ways throughout the process.

3.1.3 Parametricity-Preserving Heterogeneous Results

By switching to a Moore machine, it is possible to start generating meaningful results from the regular expression parsers. However, these results are homogeneous and not as expressive as those expected from parser combinators.

The groundwork has been laid to get more expressive results, but the specifics of the machinery needs to be adjusted to support heterogeneity. Suppose that the tape of the Moore machine encodes a stack, then instead of just writing values, the alphabet of the tape can be stack operations which allow for the pushing and reducing of values. This is straightforward, and for convenience, the tape of the Moore machine is substituted for a stack. Crucially, the stack can have values of several types on it, so long as the in-flight values' types are tracked in the type of the machine.

Generating Results from Combinators

If a stack of heterogeneous values is used to encode the results of a parser, it is important that the results eventually collapse to a single value. In this case, every complete combinator should put precisely one value onto the stack on success and must therefore consume all the sub-values generated by its sub-parsers. As such, it is no longer the case that only Eps produces results: every combinator must produce results, except Empt, which always fails. This first necessitates a change to the type of the Reg signature functor, which is now allowed to vary at each level, and so is now an indexed functor (§2.6.3):

```
type Regex a = Fix Reg a                    type Parsley a = Fix Combinator a
data Reg k a where                          data Combinator k a where
  Eps  :: a → Reg k a                         Pure  :: a → Combinator k a
  Empt :: Reg k a                             Empty :: Combinator k a
  Lit  :: Char → Reg k Char                   Char  :: Char → Combinator k Char
  Cat  :: k (a → b) → k a → Reg k b           (:<*>:) :: k (a → b) → k a → Combinator k b
  Alt  :: k a → k a → Reg k a                 (:<|>:) :: k a → k a → Combinator k a
  Rep  :: k a → Reg k [a]                     Many  :: k a → Combinator k [a]
```

Each of the six constructors have been assigned a return type: Eps continues to output a pure value; Empt produces any value as it never succeeds; Lit outputs the exact character that it parses; Cat applies a function returned by the first expression to the result of the second expression, returning the result of the function; Alt returns the successful results of its sub-branches, whose types must match; and Rep will return a list of the returns parsed by the sub-parser. In fact, this aligns perfectly with the classic applicative and alternative combinators for parser combinators (sections 2.2.2 and 2.2.3); the Lit constructor corresponds to a Char constructor, which matches any specific character and returns it (§2.3.2). As such, this functor now forms the basis for `parsley`'s Combinator AST: from here on Reg is renamed to Combinator, along with its constructors and smart constructors.

Adapting to a Stack Machine

To make the existing Moore machine work for heterogeneous values, it is necessary to switch to using a heterogeneous stack instead of a tape. The types of the in-flight stack values must be preserved in the type of the machine [Benton 2005]. For simplicity, the Out widget will be replaced by two widgets: Push which pushes a value to the stack, and Red, which combines the top two values on the stack with a given function. These are still the only two machine widgets that need to interact with the stack. When a machine accepts, it is assumed there is exactly one value left on the stack, and this will match up with the advertised goal type of the parser itself. The machine widgets are modified as follows:

```
type Moore a = Fix (State a) '[ ]
data State g k xs where
  Accept :: State g k '[g]
  Fail   :: State g k xs
  Split  :: k xs → k xs → State g k xs
  Item   :: Char → k xs → State g k xs
  Push   :: x → k (x : xs) → State g k xs
  Red    :: (x → y → z) → k (z : xs) → State g k (y : x : xs)
```

The Accept state now explicitly links the goal type g, which is fixed for the entire Moore g, with the sole g that remains on the stack; Push requires a stack xs on entry, and adapts this stack by placing an x on the front before feeding it onwards to the next state; the Red widget requires a stack with two values of types x and y, and combines them into a value of type z, which is placed onto the stack before the continuation state. The overall fixed type ensures that the initial stack provided to a machine is empty.

The comp algebra now changes to accommodate the new types, which must preserve the parametricity of the original parser [Morrisett et al. 2002]: these stronger types in the conversion help provide a lightweight guarantee that the translation is correct and also ensure that any interpreter for the machine will be total with respect to the stack. The algebra is now defined as follows:

```
type CodeGen g x = ∀xs.Fix (State g) (x : xs) → Fix (State g) xs
comp :: Combinator (CodeGen g) x → CodeGen g x
comp (Pure x)       = push x
comp (pf :<*>: px) = pf · px · red ($)
comp (Char c)       = item c · push c
comp Empty          = λk → fail
comp (p :<|>: q)    = λk → split (p k) (q k)
comp (Many p)       = λk → push mempty loop
  where loop = split (p (red (flip diffCons) loop)) (push fromCayley (red (flip ($)) k))
```

Firstly, notice the type of the carrier for the algebra, CodeGen, which links a combinator that results in an x to a continuation state that accepts the x to progress towards some goal g: this explicitly encodes the property that every complete combinator pushes exactly one value to the stack on success. This is like the shape of compile noted by Pickard and Hutton [2021]. The goal type here is the type of the outermost Parsley in the compile function itself. Secondly, the (:<*>:) rule has been adjusted to reduce the two results on the stack arising from both pf and px with function application, and the Char rule also pushes the parsed character. Thirdly, the Many combinator has been adjusted to thread a difference list (§2.5.3) through the stack, which collects up the results, reducing them to a list just before the continuation state k: this is slightly more efficient.

From `Char` to `Sat`

Section 2.3.2 mentions that satisfy is the most suitable primitive, not the char combinator as shown in §3.1.3. This can be straightforwardly implemented by introducing a Sat widget:

```
Sat :: (Char → Bool) → k (Char : xs) → State g k xs
```

This replaces Item and corresponds directly to a Satisfy constructor. By using this, there is no reason to include a push after input consumption: the Sat widget pushes the parsed character itself. From this point, Sat and Satisfy are used instead of Item and Char. For convenience, the item c k smart constructor is retained:

```
item c k = satisfy (≡ c) k
```

Popping from the Stack

The Push widget has a natural dual in the form of a Pop widget:

```
Pop :: k xs → State g k (x : xs)
```

This widget adapts a stack with a redundant value of type x so that it can be fed to a continuation that does not demand an x. It can be used to compile (:<*:) and (:*>:) combinators as follows:

```
comp (p :*>: q) = p · pop · q
comp (p :<*: q) = p · q · pop
```

These are useful to have, as they are more readily inspectable than their pure combinator definitions. This is the final adjustment that is made to the Combinator functor for regular expression parsing.

3.1.4 Introducing Recursion with Push-Down Automata

Sections 3.1.1 to 3.1.3 developed a Combinator tree that supports regular expression parsing with an applicative interface. However, there is no support for recursion in the parsers – recall that the tree must be finite, and any attempt to use recursion at this point will result in an infinite tree converted to an infinite state automaton.

Context-free grammars differ from regular expressions in that they allow for non-tail recursion in the rules of the grammar (any tail-recursive non-terminals can be represented by iteration via the many combinator). Classically, the increased expressivity of context-free languages requires push-down automata to parse (§2.1.3).

Using PDAs Sipser [2013] describes how a push-down automaton can be constructed to recognise any context-free grammar. The idea is to construct an automaton that first pushes the starting rule onto the stack; while the stack is not empty, a rule is popped and the possible right-hand sides are pushed non-deterministically; or if there is a terminal on the stack, it is parsed. This repeats until the stack is empty. While this does work, it differs significantly from the scheme currently used to compile the combinator language into an automaton.

Explicit Recursion Combinators

Firstly, there needs to be some way of denoting recursion within a parser. Assume, for now that all *named* parsers (recursive or otherwise) are given a unique (and opaque) name:

```
data NT a
```

Any parser that the user wants to refer to in multiple places (or within itself) is given a value of type NT a, where the type a matches the type returned by the parser. Section 3.3 discusses how these are generated. When a parser wishes to invoke an NT a value, it may do so using the following explicit combinator:

```
Let :: NT a → Combinator k a
```

When compiling a Parsley, a *dependent map* of the NT values mapped to their corresponding parsers must be provided: this has type DMap NT Parsley, which allows for different key/value pairs to vary in type.[2].

[2]This is provided by the `dependent-map` package on Hackage.

Extending the Machine

Currently, the abstract parsing machine as seen in §3.1.3 has a stack that is used to store intermediate results during parsing. It is not possible for the machine to select a state transition based on the contents of this stack, so this is different to the stack available to a PDA. With a PDA, the top of the stack is also used to determine the transition between one state and the next. As such, a second stack will be introduced to the machine, which can be used to influence the state transitions: in effect, a hybridisation of the heterogeneous non-deterministic Moore machine and the non-deterministic PDA.

A new scheme for recursion As opposed to the method described by Sipser [2013], a slightly different scheme is proposed: when a Let combinator is executed, it transitions from a state to the start of the automaton that represents the given NT value; in the process, it pushes the "name" of the state that should follow (i.e. the current continuation) onto the PDA stack. Instead of using Accept states, each of the parsers will end with Ret widgets, which have an edge to every valid continuation state, only transitioning if that state's "name" is on the top of the PDA stack. The outermost parser will push a reference to the Accept state before executing. The new widgets are shown in fig. 3.3. For clarity, only the valid return transitions are shown graphically.

By doing this, there is no longer one unchanging goal type: instead, the goal type g represents the value produced by the Accept state, but the type a represents what value should be accepted by the state that follows the next Ret widget, the State functor changes as follows:

```
type MoorePDA a = Fix (State a) '[ ] Void
data State g k xs a where
  Accept :: State g k '[g] Void
    ...
   Ret   :: State g k '[a] a
   Call  :: k xs x → k (x : xs) a → State g k xs a
```

The Accept state is only designed to be used when the PDA stack is empty, which is denoted by a return value of Void; the Ret state is like Accept but aligns the return type with the stack, as opposed to the goal type; the Call state takes a sub-automaton that will return with a value x and registers a second sub-automaton as the return continuation – this expects the x to have been placed on the stack; and all other widgets do not interact with the return type a. As an example of how this scheme works, consider the following example:

```
nt = In · Let
nt₁ :: NT ()
```

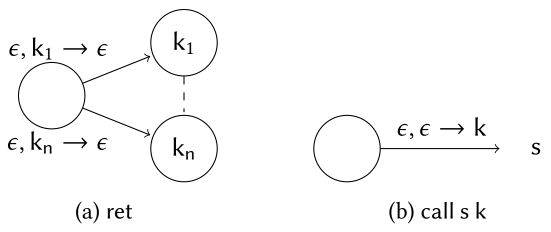

$\epsilon, k_1 \rightarrow \epsilon$
k_1
$\epsilon, k_n \rightarrow \epsilon$
k_n
$\epsilon, \epsilon \rightarrow k$
s
(a) ret
(b) call s k

Fig. 3.3: New automata widgets for recursive execution

ex3 = char 'a' *> nt nt_1 <* char 'c'

ex4 = char 'b' *> nt nt_1 <|> void (char 'a') -- *this is mapped to* $\textit{nt}_1$

This should be mapped to the automaton m, defined as follows (graphically shown in fig. 3.4):

m = call q_1 k_3

q_1 = item 'a' (pop (call q_2 k_1))

k_1 = item 'c' (pop ret)

q_2 = split (item 'b' (pop (call q_2 k_2))) (item 'a' (pop (push () ret)))

k_2 = ret

k_3 = accept

Restricting calls While the Call widget naturally suggests that the caller will thread its Moore stack directly through to the callee (as evidenced by the types), the property that every complete combinator may only push exactly one thing to the Moore stack on success indicates that the callee should not be able to rely on the contents of the caller's stack at all. As such, it is possible to make a stronger type for Call:

Call :: k '[ ] x ⟶ k (x : xs) a ⟶ State g k xs a

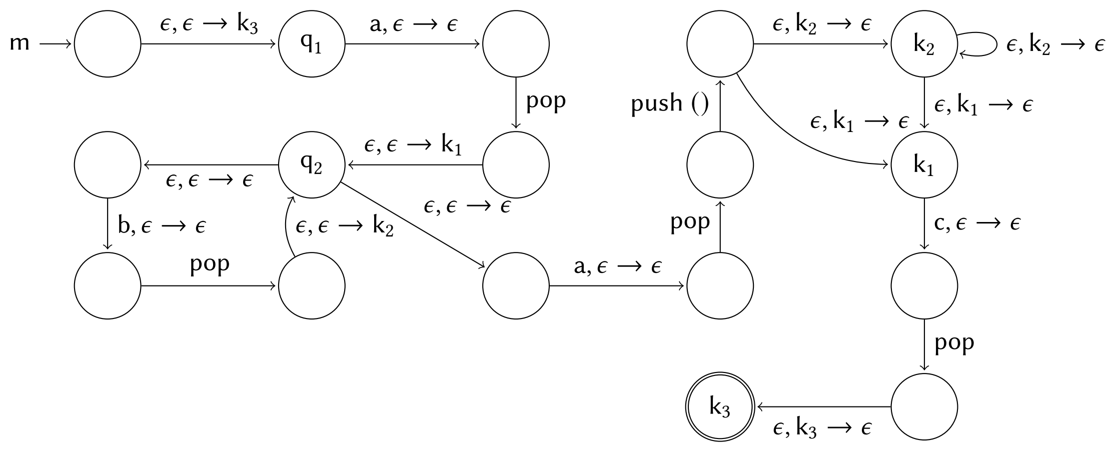
As it happens, this will be useful for the compilation, so it may be assumed that a fresh stack is initialised during a Call. This is only important when it comes to the evaluation of the machine, where the mechanism by which this is possible is clearer, so further discussion is deferred to §3.2.

Augmenting the Compiler

The addition of further bindings for compile to handle causes more significant changes to the main body of the function. Each binding must be compiled via a cata into an automaton that ends with a Ret, and then this new map of bindings can be referred to by the remaining algebra. This works due to laziness:

type CodeGen g x = ∀xs a.Fix (State g) (x : xs) a ⟶ Fix (State g) xs a

compile :: ∀g.Parsley g ⟶ DMap NT Parsley ⟶ MoorePDA g



m
$\epsilon, \epsilon \longrightarrow k_3$
q_1
$a, \epsilon \longrightarrow \epsilon$
pop
$\epsilon, \epsilon \longrightarrow k_1$
q_2
$\epsilon, \epsilon \longrightarrow \epsilon$
$b, \epsilon \longrightarrow \epsilon$
pop
$\epsilon, \epsilon \longrightarrow k_2$
$\epsilon, \epsilon \longrightarrow \epsilon$
$a, \epsilon \longrightarrow \epsilon$
pop
push ()
$\epsilon, k_2 \longrightarrow \epsilon$
k_2
$\epsilon, k_2 \longrightarrow \epsilon$
$\epsilon, k_1 \longrightarrow \epsilon$
$\epsilon, k_1 \longrightarrow \epsilon$
k_1
$c, \epsilon \longrightarrow \epsilon$
pop
k_3
$\epsilon, k_3 \longrightarrow \epsilon$

Fig. 3.4: Example automaton showing how recursion operates.

```
compile p nts = call (cata comp p ret) accept
  where ms     :: DMap NT (Fix (State g) '[ ])
        ms     = DMap.map (λp → cata comp p ret) nts

        comp :: Combinator (CodeGen g) x → CodeGen g x
        comp ...
        comp (Let nt) = call (ms DMap. !  nt)
```

The sub-parsers and main parsers are compiled in the same way. In the comp algebra nothing needs to change except for the addition of a rule for the Let combinator: this straightforwardly maps to a Call to the correspondingly compiled automaton from the dependent map ms, the continuation is used for the return continuation. The outer-most parser returns by transitioning into the Accept state. With this, fully recursive parsers and combinators can be made, allowing for the full power of context-free parsing.

3.1.5 `Parsec`-Style Backtracking

At this point, an automaton has been developed that supports the full scope of applicative non-deterministic parser combinator libraries. However, `parsley` is not a non-deterministic library, and only returns a single result on success. Instead, `parsley` more closely resembles PEG, which has a left-biased choice, with the additional constraint that backtracking can only occur if no input was consumed since a branch has been first taken.

PEG Positive Lookahead

To support lookahead, there needs to be a mechanism by which input can be rewound to an earlier position. Suppose that the entire input can be pushed onto a stack to save it, and this can be popped off the stack to restore it. This would be sufficient to implement at least positive lookahead. It does not matter which of the two stacks – the PDA stack or the Moore stack – is used for this purpose, but to improve the type-safety of the implementation, the Moore stack is chosen. First, a new combinator is added to the Combinator syntactic functor:

```
Look :: k a → Combinator k a
```

Then, three new widgets are introduced to interact with the Moore stack component of the automata:

```
Tell   :: k (String : xs) a → State g k xs a
Seek   :: k xs a → State g k (String : xs) a
Swap :: k (x : y : xs) a → State g k (y : x : xs) a
```

The Tell widget is used to push the current unconsumed input onto the stack; the Seek widget will pop some residual input and incorporate it back into the parser; and the Swap widget simply allows for the two top items on the stack to be reversed. With these, the translation of Look is straightforward:

```
comp (Look p) = tell · p · swap · seek
```

As one might expect, the ability to manipulate the input stream (even in this limited way) means the expressive power of the machine has increased: it is now possible to implement some context-sensitive languages, like the classic $a^n b^n c^n$. This is in line with PEG, however, and Ford [2004] notes that this is the case.

PEG-Style Backtracking

While the Tell and Seek widgets are useful for implementing lookahead, they can also be used to help implement the explicit backtracking in the machine – starting with the regular PEG semantics. Taking inspiration from the discussion in §3.1.4, the first branch of a choice is similar to a Call, where instead of passing a return continuation, a failure handler is provided: this can be transitioned to on encountering a Raise node (as opposed to the Fail state). This mechanism can make use of the existing PDA stack.

Dropping handlers and returns Given that Raise uses the PDA stack, it must drop all return states stored on the stack up till the next failure handler: this trivially allows for the unwinding of recursion in case of failure. Indeed, the same must now also be true for Ret, which must unwind all unused failure handlers until the next return state. Unlike Call, however, a failure handler does not last forever: when the two branches of a choice re-converge, the failure handler must no longer apply.

Ensuring failure handlers exist Some widgets (other than Empt) can fail conditionally, for instance the Sat widget. In cases where failure is possible, it must absolutely be the case that a failure handler is available to handle the error. Instead of trusting that this is the case, it is possible to add a new type index to the State, which is untouched by any widgets that do not interact with failure: the index describes how many failure handlers there are known to be on the stack. Like return states, there must always be one, and like the initial return is set to Accept, the initial handler is set to Fail. Currently, only the Sat widget interacts with the new type index:

```
Sat :: k (Char : xs) (Succ n) a → State g k xs (Succ n) a
```

This demonstrates that the Sat widget can fail, since it must be aware of at least one failure handler in scope. The following three new widgets are also added to accommodate failure logic:

```
Catch  :: k xs (Succ n) a → k (String : xs) n a → State g k xs n a
Commit :: k xs n a → State g k xs (Succ n) a
Raise  :: State g k xs (Succ n) a
```

The first widget, Catch, is like Call except it adds the second continuation state as a failure handler – signified by increasing the value of n for the first continuation – and also provides the current input onto the Moore stack for the handler; the Commit widget signifies the end of the scope of the failure handler, simply popping the top-most failure handler from the PDA stack and continuing; the Raise widget is analogous to Ret, except it transitions to the top-most failure handler on the stack.

Performing the translation With the new machinery in place, the Split widget is no longer required. However, it can be replaced by the new widgets introduced above, with the comp algebra adjusted as follows:

```
comp Empty     = λk → raise
comp (p :<|>: q) = λk → catch (p (commit k)) (seek (q k))
```

In the (:<|>:) case, a Catch is used to register a handler state seek (q k): this ensures that the input is immediately rolled back to how it was on entry to the Catch and then executes q k as before. In the p case, should p succeed, the handler is then dropped, and k is executed as normal. Similar adjustments can be made to the Many combinator, although these are not important to discuss, since many can now be implemented with naïve recursion anyway. Without Split, the machine no longer has non-deterministic behaviour, since there are no pure $\epsilon, \epsilon \longrightarrow \epsilon$ transitions remaining. The use of cata in compile is modified too, to account for the initial failure handler:

```
compile p nts = catch (cata comp p (commit ret)) fail where ...
```

PEG Negative Lookahead

With the PEG positive lookahead operation supported, and a new left-biased semantics for (<|>), it is now possible to implement negative lookahead without the introduction of any further widgets. As discussed in §2.3.2, it is not possible to implement it with pure combinators alone without at least the selective functor combinators. However, the widgets used for Look and (:<|>:) achieve the same effect:

```
NegLook :: k a → Combinator k ()
comp (NegLook p) = λk → catch (tell (p (pop (seek (commit raise))))) (seek (push () k))
```

Here, the handler for Catch says that input consumed should be rolled back, () pushed, and then continue with the regular continuation k; this is in keeping with how negative lookahead works – if the parser p fails, it should succeed and continue with no input consumed. Inside the scope of the Catch however, the current input is obtained, then p executed: if it succeeded, the result is popped, input is rolled back and then, crucially, the failure handler is dropped before raising an error. The act of dropping the handler before the Raise is not possible with the pure combinators, which is why selectives are required to perform the failure after exiting the choice.

Supporting Parsec Backtracking

Finally, with the entire suite of PEG operations implemented – and making the machine and combinator language compliant not with context-free grammars but PEG instead – it is possible to refine the comp rule for (:<|>:) to instead adopt the `parsec` backtracking semantics, where the recovery can only occur if no input has been consumed during the attempt to parse the first branch. This necessitates the introduction of the atomic combinator as well (§2.3.2), which is expressed by the following new constructor:

```
Atomic :: k a → Combinator k a
```

The implementation of comp must be adjusted to not only incorporate a translation for Atomic, but also alter the failure handler assigned in the (:<|>:) rule to one that verifies the requirement about input consumption. The Atomic rule is easy to implement using Catch:

```
comp (Atomic p) = λk → catch (p (commit k)) (seek raise)
```

When an Atomic is encountered, any failure within p is caught, the consumed input is rewound, and then the parser fails again. This allows the failure to be handled by any enclosing handler without it being aware that any input had been consumed. This straightforwardly satisfies the requirements for the atomic combinator.

To implement the adjusted semantics for (<|>), however, it is necessary to introduce another widget, Same:

```
Same :: k xs (Succ n) a → State g k (String : String : xs) (Succ n) a
```

This widget takes two inputs from the Moore stack and compares them to check if they are the same input: if so, it will invoke its provided continuation state, otherwise it will act as Raise. This can be used to implement the desired behaviour for `parsec`, adjusting the comp rule for (:<|>:) as follows:

```
comp (p :<|>: q) = λk → catch (p (commit k)) (tell (same (q k)))
```

Here, the current input is queried on entry to the handler with Tell, and then Same is used to established whether the input from the start of the Catch matches with the current input: if it does, then take the second branch, otherwise fail. This is precisely the behaviour advertised by `parsec`'s choice combinator.

3.1.6 Inspecting Results for Dynamic Branching

In §3.1.5, the Split widget was removed in favour of Catch/Commit, allowing for a left-biased choice adhering to `parsec`'s backtracking semantics. In the process, the Same widget was also introduced, which allows for a decision to be made based on the contents of the Moore stack. Given its limited utility, it does not affect the expressive power of the machine much on its own.

However, `parsley` is a selective parser combinator library and, as such, it must support the branch combinator (§2.2.4). As usual, this is expressed by adding a new combinator to the Combinator syntactic functor:

```
Branch :: k (Either x y) → k (x → a) → k (y → a) → Combinator k a
```

At its core, branch allows for the result of one parser to direct which of the other two parsers should be executed. At the machine level, this corresponds to a conditional state transition based on the result at the top of the Moore stack. The Case widget realises this:

```
Case :: k (x : xs) n a → k (y : xs) n a → State g k (Either x y : xs) n a
```

The widget works by choosing to transition to one of the two continuation states: the first if the stack has a Left x and the second if the stack has a Right y. The comp algebra needs one final rule added to accommodate Branch:

```
kase k1 k2 = In (Case k1 k2)
comp (Branch c l r) = λk → let k′ = red (flip ($)) k in c (kase (l k′) (r k′))
```

The idea is to evaluate c to put its result on the stack, then each of the two branches first put their function on the stack and then reduce the stack with function application before continuing. Using the Case widget, it is possible to remove the Same widget defined in the previous section:

```
same k = red (λnew old → if new ≡ old then Right () else Left ())
             (kase raise (pop · k))
```

The same widget first takes the two inputs on the stack and compares them for equality (there are more efficient ways to do this) and returns one of a Left () or Right (), which expresses as much information as a Bool. If a Left () is returned, representing False, then same k should fail to the next handler, otherwise the left-over () is popped from the Moore stack and k is performed.

Summary

Throughout this section, two representations of a parser combinator library have been developed: the first, the Combinator tree, is used to represent the high-level structure of combinators, where all information about scope is preserved; the second, the State functor, represents a deterministic abstract parsing machine, where the control flow information is more explicit. From here on, the constructors of the State functor are referred to as *instructions*, as opposed to automaton widgets, and they form an instruction set for the now complete abstract machine that will be executed and staged in §3.2. The machine itself is a hybridisation between a Moore machine with a heterogeneous stack, and a deterministic push-down automaton: unlike both abstractions, however, the values on the Moore stack can direct its execution – via the Case instruction.

There is one last simplification that can be made to the machine. Currently, the Accept and Fail states are still retained within the syntax, and these are registered on the PDA stack so they can be used within the internal Ret and Raise instructions. However, this is not strictly necessary: instead, assume that the implementations of these final states can be provided during machine interpretation – in other words, the machine is free in these two terminal states, and State is renamed to Instr so that:

```
type Machine a = Fix (State a) '[ ] Zero Void
```

should instead become the simplified:

```
type Machine a = Fix Instr '[ ] One a
```

Notice here that the goal type has disappeared from the Instr (since only Accept referenced the goal before), and the return type has been set to a instead; in addition, the number of failure handlers required by a machine is now set to One. The Call instruction can also be adjusted to require exactly one failure handler for a call as well, the motivation for which will become clearer in §3.2.1. This gives the final machine description for this section of:

```
data Instr k xs n a where
  Sat    :: (Char → Bool) → k (Char : xs) (Succ n) a → Instr k xs (Succ n) a
  Push   :: x → k (x : xs) n a → Instr k xs n a
  Pop    :: k xs n a → Instr k (x : xs) n a
  Swap   :: k (x : y : xs) n a → Instr k (y : x : xs) n a
  Red    :: (x → y → z) → k (z : xs) n a → Instr k (y : x : xs) n a
  Case   :: k (x : xs) n a → k (y : xs) n a → Instr k (Either x y : xs) n a
  Tell   :: k (String : xs) n a → Instr k xs n a
  Seek   :: k xs n a → Instr k (String : xs) n a
  Call   :: k '[ ] One x → k (x : xs) (Succ n) a → Instr k xs (Succ n) a
  Ret    :: Instr k '[a] n a
  Catch  :: k xs (Succ n) a → k (String : xs) n a → Instr k xs n a
  Commit :: k xs n a → Instr k xs (Succ n) a
  Raise  :: Instr k xs (Succ n) a
```

With this final description in place, the compile function can be adjusted to accommodate the new free return continuation and failure handler and simpler types:

```
type CodeGen x = ∀xs n a.Fix Instr (x : xs) (Succ n) a → Fix Instr xs (Succ n) a
compile :: Parsley a → DMap NT Parsley → Machine a
compile p nts = cata comp p ret
```

3.2 Staged Interpretation of the Machine

In §3.1 the syntactic functor of an abstract machine for `parsley` was introduced, called Instr. Parsers represented by a combinator tree are translated into machine instructions ready for interpretation. This section sheds light on the process of interpreting the machine; as well as how to optimise away the associated interpretive overheads of not only the machine, but the entire frontend for `parsley` too.

First, §3.2.1 discusses the exact evaluation semantics for the machine, and how it differs slightly from the more theoretical description in the previous section. Then, §3.2.2 highlights where the divide between the static and dynamic information for the parsers are and outlines the changes that are necessary to accommodate the staging. Following from this, §3.2.3 stages the evaluation functions, discussing the implications this has on the generated code. §3.2.3 culminates in a naïve staged interpreter, but it is broken by the presence of unobservable recursion in the machine description; §3.2.4 fixes this by modifying the Instr definition to include NT and introduces explicit *join-points* to the instruction set, allowing for further observation of shared code sequences within the machine. From this point, relevant code changes will be highlighted.

3.2.1 Evaluating the Machine

The machine developed throughout §3.1 was described as a deterministic push-down heterogeneous Moore machine, comprising of two stacks: the Moore stack, for storing intermediate parsing results; and the PDA stack, used to track returns and failure handlers. The overall type of the evaluation function is important (i.e., the semantic model for the machine as a whole):

```
eval :: Machine a → (String → Maybe a)
```

This means that, outwardly, a complete Machine a can be thought of as a function that takes the input and may produce a value of the goal type a. What this does not describe is the type of the internal state of the machine, Eval, which is the topic of §3.2.1; then §3.2.1 provides the denotational semantics for the instructions.

Concrete State

The first step is to encode the type of the state that is threaded through the evaluation of the machine. This includes the exact representation of values on the PDA stack, and how the various components relate to the strong type indices present in the type of Instr. There are (currently) three components: a heterogeneous stack of intermediate values, a stack of failure handlers and returns, and the input stream itself. The input does not require any special treatment and is represented as a String. The other two stacks, however, require some thought.

The heterogeneous Moore stack The stack used for intermediate results can contain values of many different types, as encoded by the type index xs in Instr k xs n a. For the value-level, this may be realised by the classic heterogeneous list, which can serve as a LIFO stack:

```
data HList (xs :: [∗]) where
  HNil  :: HList '[ ]
  HCons :: x → HList xs → HList (x : xs)
```

This is a standard list structure, but where each cell in the list reflects the types stored inside it at the type-level.

The PDA stack The contents of the PDA stack can contain one of two different kinds of values following the description of the machine from §3.1: a failure handler, or a return state. However, the type index n in Instr k xs n a relates to the number of handlers currently present on the stack, and the type a relates to the return required for the current return. Furthermore, §3.1.4 and §3.1.5 mention that the objects pushed onto this stack capture the current Moore stack. Taking this literally would result in a vector-like structure:

```
data PDAStack r (n :: Nat) x where
  Nil     :: PDAStack r Zero Void
  Return  :: Instr (Eval r) (x : xs) n a → HList xs → PDAStack r n a → PDAStack r n x
  Handler :: Instr (Eval r) (String : xs) n a → HList (String : xs) → PDAStack r n a → PDAStack r (Succ n) a
```

The continuation type k of the Instr values passed to the Return and Handler constructors, Eval r, is the internal type of the evaluation. However, this is quite intricate to deal with and could be improved in two practical ways: the first is to switch to a continuation-based model [Appel 2007] to make the capturing more concise and improve the binding-time of the data; and the second is to split the stack into two.

Switching to continuations The use of continuations to represent the capturing has already been hinted at when the Accept and Fail "states" were removed from the Instr syntactic functor in §3.1: without these states, it would not be possible to place Instr Eval values onto a stack to represent them. The idea is for each continuation to capture the state it requires and take as explicit arguments uncapturable state. However, the results of these continuations must be the result of the overall evaluation process, called r. This materialises as two types of continuation:

```
type Handler r = String → r
type Return r x = x → String → r
```

These two types provide a clearer definition of the states that exist on the PDA stack:

```
data PDAStack r (n :: Nat) x where
  Nil     :: PDAStack r Zero Void
  Return  :: Return r a → PDAStack r n b → PDAStack r n a
  Handler :: Handler r → PDAStack r n a → PDAStack r (Succ n) a
```

Happily, this avoids the awkwardness arising from having partially-evaluated instructions present on the stack: this would end up being a problem for binding-time later during staging.

Splitting the PDA stack Before the continuations, it was important that both returns and handlers were present on the same stack: when executing a failure, it is necessary to unwind all the way back to the next failure handler, dropping all the returns along the way. However, with continuations this is no longer an issue: a failure handler captures both the return continuations and failure handlers that exist at the point of the continuation coming into existence. This means that the stack can be freely split into two, which simplifies matters:

```
type HandlerStack r n = Vec n (Handler r)
data ReturnStack r x where
  NoReturn :: ReturnStack r Void
  Return   :: Return r x → ReturnStack r y → ReturnStack r x
```

Here, Vec n is a Nat-indexed list of specific length n with constructors Cons and Nil. To simplify things even more, notice that the type of the ReturnStack only ever cares about one return type at a time: it does not matter how many return continuations there are. In fact, while the Commit instruction exists to invalidate a Handler currently on the stack, there is no such equivalent for return continuations; since the return continuation captures the next continuation when it comes into existence: there is no need for a stack at all!

```
type ReturnStack r x = Return r x
```

The overall state At this point, the four components of the machine state can be packaged up for convenience:

```
data Γ r xs n a = Γ {input    :: String
                    ,moore    :: HList xs
                    ,handlers :: Vec n (String → r)
                    ,retCont  :: a → String → r
                    }
```

With this state Γ, the internal type of the machine evaluation, called Eval, can be defined:

```
type Eval r xs n a = Γ r xs n a → r
```

To tie this into the wider story, take the underlying type indices of a complete Machine a, extrapolate the effect on Eval and the state Γ, and a pleasing result appears:

```
type Machine a = Fix Instr '[ ] One a
cata alg :: Machine a → Eval r '[ ] One a
         :: Machine a → (Γ r '[ ] One a → r)
         :: Machine a → (String → HList '[ ] → Vec One (String → r) → (a → String → r) → r)
         :: Machine a → (String → (String → r) → (a → String → r) → r)
```

Squint a little, and this specialised Eval type matches up closely with the two-continuation definition for Parser given at the end of §2.3.3: the differences lie in the lack of error messages, and the lack of the Bool for determining whether input has been consumed. Ultimately though, these differences are superficial: the Bool was purely an optimisation to avoid an expensive String comparison and in practice is improved by carrying an offset alongside the input, so for simplicity the Bool is dropped, and expensive comparison tolerated.

Denotational Semantics for `Instr`

With the type of the state Γ and the carrier type Eval r defined, it is possible to give the definition of eval:

```
eval :: Machine a → String → Maybe a
eval m inp = cata alg m (Γ {input = inp, moore = HNil, handlers = Cons fail Nil, retCont = accept})
  where fail :: String → Maybe a
        fail _ = Nothing
        accept :: a → String → Maybe a
        accept x _ = Just x

        alg :: Instr (Eval r) xs n a → Eval r xs n a
        alg Ret      γ = evalRet γ
        alg (Push x) γ = evalPush x γ
        alg ...
```

The evaluation of a machine is a simple cata over the instructions, with a state γ threaded through. The initial failure handler and return continuation are the functions fail and accept, which mirror the eponymous states dropped from the machine in §3.1: these fix the return type r of the evaluation to be Maybe a. The algebra alg just maps instructions to its respective evaluation function, which serves as its denotation. These evaluation functions are explored below.

Moore stack instructions Firstly, semantics are given to each of the instructions that purely manipulate the Moore stack, Push, Pop, Swap, Red, and Case:

```
evalPush :: x → Eval r (x : xs) n a → Eval r xs n a
evalPush x k γ{..} = k (γ {moore = HCons x moore})

evalPop :: Eval r xs n a → Eval r (x : xs) n a
evalPop k γ{..} = let HCons x xs = moore in k (γ {moore = xs})

evalSwap :: Eval r (x : y : xs) n a → Eval r (y : x : xs) n a
evalSwap k γ{..} = let HCons x (HCons y xs) = moore in k (γ {moore = HCons y (HCons x xs)})

evalRed :: (x → y → z) → Eval r (z : xs) n a → Eval r (y : x : xs) n a
evalRed f k γ{..} = let HCons y (HCons x xs) = moore in k (γ {moore = HCons (f x y) xs})
```

Each of the above four instructions manipulates the stack, simply by manipulating the underlying heterogeneous list. There is no room for error here, as the types are very precise.

```
evalCase :: Eval r (x : xs) n a → Eval r (y : xs) n a → Eval r (Either x y : xs) n a
evalCase left right γ{..} = case moore of
  HCons (Left x) xs  → left  (γ {moore = HCons x xs})
  HCons (Right y) xs → right (γ {moore = HCons y xs})
```

The evalCase function also inspects the stack but performs a pattern match on the value to direct to one of its continuations.

Failure instructions Both Raise and Commit are also difficult to get wrong, guided by the types in Commit's case and by parametricity in Raise's:

```
evalRaise :: Eval r xs (Succ n) a
evalRaise γ{..} = let Cons h _ = handlers in h input
evalCommit :: Eval r xs n a → Eval r xs (Succ n) a
evalCommit k γ{..} = let Cons _ hs = handlers in k (γ {handlers = hs})
```

The Catch instruction, however, is more interesting, as the interactions with the state are not entirely type safe:

```
evalCatch :: Eval r xs (Succ n) a → Eval r (String : xs) n a → Eval r xs n a
evalCatch k h γ{..} = k (γ {handlers = handler : handlers})
  where handler input′ = h (γ {input = input′, moore = HCons input moore})
```

Here, the execution of the continuation k is straightforward, however, the construction of the handler to place on the stack is interesting: input needs to be placed on the Moore stack from the types, but this must be the input from the current state γ and not the updated input passed to the handler itself.

Input Interaction The Sat instruction is the only place where input is inspected explicitly:

```
evalSat :: (Char → Bool) → Eval r (Char : xs) (Succ n) a → Eval r xs (Succ n) a
evalSat f k γ{..} = case input of
  c : cs | f c → k (γ {input = cs, moore = HCons c moore})
  _            → evalRaise γ
```

If there is enough input and it matches the provided predicate, then continue onwards with k, adjusting the state γ to suit. Otherwise, it is convenient to piggy-back off the semantics for Raise.

```
evalTell :: Eval r (String : xs) n a → Eval r xs n a
evalTell k γ{..} = k (γ {moore = HCons input xs})
evalSeek :: Eval r xs n a → Eval r (String : xs) n a
evalSeek k γ{..} = let HCons input′ xs = moore in k (γ {input = input′, moore = xs})
```

Both Seek and Tell have simple interactions with the stack and input, though these are not entirely type-safe: the type of Tell, at least, could be made bullet-proof by making the Instr and Eval types parametric in the input type. In practice, this is beneficial for supporting different input backends, but this is omitted here to keep things simple. Regardless, care must be taken to set the input in the Seek instruction.

Recursion instructions The final two instructions, Call and Ret, deal with recursion, and therefore the retCont part of the state γ:

```
evalRet :: Eval r '[a] n a
evalRet γ{..} = let HCons x HNil = moore in retCont x input
```

Like Raise, Ret only has a single implementation by parametricity since a failure handler is not guaranteed to be available: call the retCont with the top of the stack!

```
evalCall :: Eval r '[ ] One x → Eval r (x : xs) (Succ n) a → Eval r xs (Succ n) a
evalCall body k γ{..} = body γ′
  where Cons h _     = handlers
        ret x input′ = k (γ {input = input′, moore = HCons x moore})
        γ′           = Γ {input = input, moore = HNil, handlers = Cons h Nil, retCont = ret}
```

The Call instruction is like Catch, and mostly writes itself due to parametricity, apart from of ensuring that the new continuation ret sets the input correctly.

3.2.2 Identifying Static Information

Now that §3.2.1 has given a concrete underlying semantics for the machine, as well as the type of the state to be threaded through the evaluation (§3.2.1), this section focuses on what parts of the state are static and which are dynamic. In the process some of the choices made previously are justified further, and some modifications to the types of the machine and combinators are required to make the jump to Code.

Separating the Data

To recap, the carrier type of the evaluator Eval, and the state of the machine, Γ, are defined as follows:

```
type Eval r xs n a = Γ r xs n a → r
data Γ r xs n a = Γ {input    :: String
                   , moore    :: HList xs
                   , handlers :: Vec n (String → r)
                   , retCont  :: a → String → r
                   }
```

The most obvious example of dynamic information here is the input String, and the most obvious example of static information is the Machine itself. A naïve first step is to render the entire Eval type as dynamic code:

```
type Eval r xs n a = Code (Γ r xs n a → r)
```

This would automatically make it so that the evaluation of the machine would become static, with the residual code being the evaluation function from Γ to the final result r. However, §2.4 states that functions themselves can be statically deconstructed, so this is best rendered as:

```
type Eval r xs n a = Code (Γ r xs n a) → Code r
```

In a similar vein, a dynamic product type can be statically split into its constituent parts when it occurs contravariantly, as is the case here. As such, the Code can, and should, be pushed into Γ, which has the effect of eliminating the record type entirely from any generated code:

```
type Eval r xs n a = Γ r xs n a → Code r

data Γ r xs n a = Γ {input    :: Code String
                    ,moore    :: Code (HList xs)
                    ,handlers :: Code (Vec n (String → r))
                    ,retCont  :: Code (a → String → r)
                    }
```

This is already a good start: the machine can be statically eliminated, as can the bundling of the state. Effectively, this will become a function with four arguments producing a result of type r. As noted above, the input is clearly dynamic, so it should remain as Code String. The exact number of values in the Moore stack is always known, evidenced by its type: as §2.4.1 notes, a strong type usually implies static data.

```
data QList xs where
  QNil  :: QList '[ ]
  QCons :: Code x → QList xs → QList (x : xs)

data Γ r xs n a = Γ {input    :: Code String
                    ,moore    :: QList xs
                    ,handlers :: Code (Vec n (String → r))
                    ,retCont  :: Code (a → String → r)
                    }
```

Given that the shape of the stack is known statically from the instructions that manipulate it, remembering of course that the instructions are statically known, the Code can be pushed deeper into it, shifting the stack to compile-time. This is realised by the refined type QList, which is now used in place of Code (HList xs) within Γ. In a similar vein, the handler stack's shape is also statically known, evidenced by the type-level natural number, and the Code can be distributed through the Vec n:

```
data Γ r xs n a = Γ {input    :: Code String
                    ,moore    :: QList xs
                    ,handlers :: Vec n (Code (String → r))
                    ,retCont  :: Code (a → String → r)
                    }
```

The continuations are dynamic as they cross a *dynamic boundary*, namely during Call. One would expect that compiling recursive parsers would result in recursively defined functions in the generated code, meaning that the single handler and return continuation required for the recursive call would need to cross through a call at runtime. In fact, it is this that makes the use of continuations so important: if the implementation passed the entire stack of handlers or return continuations through to a call, it would force these to become dynamic data too. For this chapter, this is the final type of Γ.

Adjusting the Combinators and Instructions

With the components of the evaluator separated clearly into static and dynamic parts, some of the dynamic data needs to be propagated back into the types of the instructions and even the combinators. In particular, the user-directed results have been rendered as dynamic information within the moore component of Γ, so they must be dynamic within the syntax too. Furthermore, the predicates used to check the input naturally interact with dynamic information, so must too become dynamic:

```
data Combinator k a where
   Pure    :: Code a            → Combinator k a
  Satisfy :: Code (Char → Bool) → Combinator k Char
    ...
```

As such, the Pure and Satisfy combinators now take Code directly, instead of regular values. This has the side-effect of making `parsley` no longer adhere to the classic typeclasses like Applicative and Functor. This is not so much of an issue, however, and `parsley` redefines all the combinators itself. Similarly, a few of the instructions also need to be adjusted to account for the presence of dynamic data:

```
data Instr k xs n a where
   Sat  :: Code (Char → Bool) → k (Char : xs) (Succ n) a → Instr k xs (Succ n) a
  Push :: Code x            → k (x : xs) n a → Instr k xs n a
   Red  :: Code (x → y → z)   → k (z : xs) n a → Instr k (y : x : xs) n a
   ...
```

As with the combinators, these three instructions all interact with user-defined values or dynamic information: they must all be Code. Moving forward into CHAPTER 6: ANALYSIS AND OPTIMISATION, however, *partially static* data will be applied to unlock optimisation opportunities for these values.

3.2.3 Staging the Machine

Now that the high-level changes to the Instr type have been discussed as well as the changes to the state Γ and the carrier type Eval, the impact on the binding-time analysis on the evaluation function itself will be explored. The changes to the eval function are as follows:

```
eval :: Machine a → Code String → Code (Maybe a)
eval m inp = cata alg m (Γ {input = inp, moore = QNil, handlers = Cons fail Nil, retCont = accept})
  where fail    = ⟦λ_ → Nothing⟧
        accept = ⟦λx _ → Just x⟧

        alg :: Instr (Eval r) xs n a → Eval r xs n a
        alg Ret        γ = evalRet γ
        alg (Push x) γ = evalPush x γ
        alg ...
```

Firstly, the eval function's original type, Machine a ⟶ String ⟶ Maybe a is no longer fit for purpose, as it must return Code and the String must be passed in dynamically. Other than this, not much needs to change, except that the initial failure and return continuations are instead wrapped in Code. The algebra itself remains unaltered too, and the changes are instead reflected in the individual instruction evaluators themselves.

Fully static instructions Six of the instructions are unaltered by the effect of the binding-time analysis performed in §3.2.2: Push, Pop, Swap, Commit, Seek, and Tell. In all these cases, the division of static and dynamic information has been so aggressive that they each only do compile-time work manipulating the static stacks within the state Γ; this means that they all have zero runtime cost.

One interesting observation is with Push, which is normally how user-defined values enter the system. Because this happens purely at compile-time, these user-defined values freely float around the generated code until they find themselves in the right place – either in a reduction or fed to a return continuation.

However, the most interesting consequence of the fully static nature of these instructions is that the Pop instruction has the power to ensure that entire values are not only never computed, but that the code to compute them never even exists. This phenomenon is known as *staged fusion*, where statically unused values are guaranteed not to materialise in the runtime code [Willis, Wu, and Schrijvers 2022].

Partially-static instructions The remaining instructions all contain a mix of static and dynamic interactions.

```
evalRed :: Code (x ⟶ y ⟶ z) ⟶ Eval r (z : xs) n a ⟶ Eval r (y : x : xs) n a
evalRed f k γ{..} = let QCons y (QCons x xs) = moore in k (γ {moore = QCons ⟦$f $x $y⟧ xs})
evalCase :: Eval r (x : xs) n a ⟶ Eval r (y : xs) n a ⟶ Eval r (Either x y : xs) n a
evalCase left right γ{..} = let QCons exy xs = moore in
  ⟦ case $exy of
    Left x   ⟶ $(left  (γ {moore = QCons ⟦x⟧ xs}))
    Right y ⟶ $(right (γ {moore = QCons ⟦y⟧ xs})) ⟧
```

The Red instruction is a straightforward change, where the application of the function to the top two values on the stack is done under a quote. However, while the application happens at runtime, the value is placed back onto the stack statically, so it will still float around to the right consumer. The Case instruction collects the scrutinee from the stack in a static way, but the actual pattern match happens at runtime. The two continuations are spliced in on the appropriate branch.

```
evalRaise :: Eval r xs (Succ n) a
evalRaise γ{..} = let Cons h _ = handlers in ⟦$h $input⟧
evalRet :: Eval r '[a] n a
evalRet γ{..} = let QCons x QNil = moore in ⟦$retCont $x $input⟧
```

Both Raise and Ret act similarly; the act of obtaining the right continuation to invoke is performed purely statically, and the application of the continuation to its required arguments is performed dynamically. If the handlers and continuations had been statically broken up, both instructions would have remained unchanged from their original definitions; however, since the majority of such continuations are purely dynamic, it does not make a great deal of difference and there would still be a cost associated with the application – it would just be shifted elsewhere. This will change in CHAPTER 6, however.

Instructions that bind The next instruction, Catch, is interesting because it needs to be careful about inadvertently generating massive amounts of duplicated code.

```
evalCatch :: Eval r xs (Succ n) a → Eval r (String : xs) n a → Eval r xs n a
evalCatch k h γ{..} = k (γ {handlers = handler : handlers})
  where handler = ⟦ λinput′ → $(h (γ {input = ⟦input′⟧, moore = QCons input moore})) ⟧
```

If one were to stage the Catch instruction in the minimal way to make it type-check again, the above definition would be the result: the only change is that the handler is defined within code. However, as demonstrated in the exactly example in §2.4, the quoting and splicing of code does not do any simplification. Suppose that the Raise instruction just calls the next handler on the stack, and this Catch just places a lambda onto the stack. This means that every single place where a Raise happens directly underneath this Catch, the lambda it placed onto the stack is going to be dumped exactly as written straight into the generated application. Now suppose that the handlers for Catch can sometimes include entire parsers, as is the case with (<|>), and this evalCatch can result in exponential explosion in the size of the code. to avoid this it is important to bind the new handler much more carefully:

```
evalCatch :: Eval r xs (Succ n) a → Eval r (String : xs) n a → Eval r xs n a
evalCatch k h γ{..} =
  ⟦ let handler input′ = $(h (γ {input = ⟦input′⟧, moore = QCons input moore}))
    in $(k (γ {handlers = ⟦handler⟧ : handlers})) ⟧
```

This version is much safer: the handler is explicitly let-bound at the point where the Catch occurs and a reference to the binding is fed into the handler stack. However, some handlers, like atomic's seek raise, benefit from allowing the handler to be inlined directly into the Raise, especially if it is composed of zero-cost instructions. On the other hand, the Sat instruction does not require the same care:

```
evalSat :: Code (Char → Bool) → Eval r (Char : xs) (Succ n) a → Eval r xs (Succ n) a
evalSat f k γ{..} = ⟦ case $input of
  c : cs | $f c → $(k (γ {input = ⟦cs⟧, moore = QCons ⟦c⟧ moore}))
  _             → $(evalRaise γ) ⟧
```

The Sat instruction, like Case, performs a dynamic case-analysis on the input, and statically feeds the results onwards to the appropriate continuation. Suppose, though, that GHC desugars that guard by duplicating the right-hand side, or even that the meta-programmer restructures it in such a way that two instances of evalRaise are present: is this an issue like it was in Catch? Simply put, no: all handlers are going to refer to bound functions, and a single function application is cheap. One certainly could write something like ⟦ **let** bad = $(evalRaise γ) **in**...⟧, but this just serves to introduce more laziness into the generated code, and failures are not normally expected to the be the most-likely control-flow path.

A problematic instruction The final instruction is the Call instruction for performing recursion:

```
evalCall :: Eval r '[ ] One x → Eval r (x : xs) (Succ n) a → Eval r xs (Succ n) a
evalCall body k γ{..} = body γ′
  where Cons h _ = handlers
        ret      = ⟦ λx input′ → $(k (γ {input = ⟦input′⟧, moore = QCons ⟦x⟧ moore})) ⟧
        γ′       = Γ {input = input, moore = QNil, handlers = Cons h Nil, retCont = ret}
```

The above definition of Call typechecks, with the only required change being, like Catch, enclosing the definition of the return continuation within Code. However, the expected dynamic boundary is not seen here: in fact, this evalCall has erroneously tried to stage a cyclic, non-observably recursive value, and is therefore attempting to generate an infinite amount of code. The next section will address this properly, and provide the correct definition for evalCall, and the Call instruction, as well as handling some other code-growth issues present in the machine itself, as opposed to the staging code.

3.2.4 Introducing Join-Points

The previous section applied the identified separation of static and dynamic knowledge to the evaluation of each instruction, producing a staged interpreter for the Machine language. However, there are two problems with the representation of the machine itself:

1. While the machines are technically finite, they are not finite in an observable way.
2. Continuations can be duplicated in multiple parts of the machine.

This section addresses both issues, resulting in a complete, working, and efficient staged interpreter for `parsley`'s low-level abstract machine.

Unobservable Recursion in the Machine

The first issue is one that has already been discussed before, back in both sections 3.1.1 and 3.1.4: using knot-tying to represent self-references forms a finite graph, but one that is infinitely traversable. This was a problem when compiling recursion, because a cata performs a bottom-up traversal of the structure and would have to infinitely traverse to reach this bottom-most point. To fix this, NT values were introduced to provide explicit names for parsers, the bindings for which kept in a dependent map.

However, when translating these down to automata, the NT values were discarded in favour of structural sharing again: this kept the discussion simple and allowed for a clear demonstration of how the Call instruction maps to an automaton widget, but it results in a non-observably recursive machine instead, which is now a problem. Thankfully, the fix is very straightforward, the Call instruction is adjusted so that it does not have an explicit continuation, but instead has the NT value:

```
Call :: NT x → k (x : xs) (Succ n) a → Instr k xs (Succ n) a
```

Now the compile function no longer needs access to the map of bindings; this work can be done by the caller of compile instead:

```
compile :: Parsley a → Machine a
compile p = cata comp p ret
  where comp :: ...
        comp (Let n) = call nt
        comp ...
```

Now, it suffices to call compile p and DMap.map compile nts to obtain both a Machine a and a DMap NT Machine. Both are now going to be required for the eval function, which has the new signature:

```
eval :: Machine a → DMap NT Machine → Code String → Code (Maybe a)
```

The internal changes to the Eval carrier type to make this work are covered in §3.2.4.

Preventing Exponential Code Growth

The second problem with the current machine representation is that, for simplicity, continuation instructions are allowed to occur in multiple places within the compile function's comp algebra. This occurs in two rules:

```
comp (p :<|>: q)     = λk → catch (p (commit k)) (tell (same (q k)))
comp (Branch b l r) = λk → let k′ = red ⟦flip ($)⟧ k in b (kase (l k′) (r k′))
```

In both cases, k appears twice in the right-hand side of the rule (in Branch this is masked behind k′, which itself appears twice). This may seem innocuous, since the k is referentially the same in both positions, but, like the problem with recursive parsers and NT, the staging will duplicate the logic for each individual occurrence of k. Since the evaluation produces code, this means that each (<|>) or branch combinator will double the size of the remaining parser.

One way of fixing this would be to introduce new NT values and use call to unify the branches, but this is heavy-handed: the dynamic-boundary created by call requires both a failure handler and a return continuation to be provided, but localised join-points like those found in these rules only require a return continuation, called let-insertion [Yallop 2017]. Instead, two new instructions are introduced to the machine:

```
MkJoin :: Φ x → k (x : xs) n a → k xs n a → Instr k xs n a
Join   :: Φ x → Instr k (x : xs) n a
```

These instructions represent short-lived let-bound machines that expect a single value from the stack to be passed to them when called (like a return continuation). These join-points [Maurer et al. 2017] allow for otherwise duplicated instruction trees to be factored out explicitly. The Φ type is used to give a name to a join-point, much like NT gives a name to a non-terminal.

Representing join-points The type Φ is represented as follows:

```
newtype Φ x = Φ Word64
```

That is, each Φ node is just a wrapper around a number that uniquely identifies it. Each Φ is associated with a phantom type that represents the value that needs to be provided to invoke it. The idea is to keep track of the join-points that are in scope – a join-point is in scope within its second continuation, and out-of-scope otherwise – during the evaluation of the machine, so they can be referred to as needed. To do this, the bindings will need to be kept, like with NT, in a dependent map: this is explored further in §3.2.4.

A Φ can be used in a dependent map only if it has generalised equality GEq and generalised GCompare instances (this is true of NT as well, see §3.3.2):

```
instance GEq Φ where
  geq :: Φ a → Φ b → Maybe (a :∼: b)
  geq (Φ x) (Φ y)
     | u ≡ v      = Just (unsafeCoerce Refl)
     | otherwise = Nothing
instance GCompare Φ where
  gcompare :: Φ a → Φ b → GOrdering a b
  gcompare φ₁@(Φ x) φ₂@(Φ y) = case compare x y of
    LT  → GLT
    EQ → case geq φ₁ φ₂ of Just Refl → GEQ
    GT → GGT
```

Without making every single parser and combinator Typeable, it is not possible to prove that two Φ values with the same underlying word have the same type: since these words are designed to be unique per-machine and only created during the compile function, it is permissible to use unsafeCoerce to provide the "proof" of type equality.

Generating join-points Join-points are generated on-demand during the compile function. One way of doing this is to thread a Word64 through the comp algebra via the State monad. The freshΦ operation can be used to generate a new Φ:

```
freshΦ :: State Word64 (Φ x)
freshΦ = do φ ← get
            put (φ + 1)
            return (Φ φ)
```

The compile function is then modified so that the result of the cata (after providing the final ret continuation) is inside the State Word64 monad:

```
type CodeGen x = ∀xs n a.Fix Instr (x : xs) (Succ n) a → State Word64 (Fix Instr xs (Succ n) a)
compile :: Parsley a → Machine a
compile p = evalState (cata comp p ret) 0
  where comp :: ∀x Combinator CodeGen x → CodeGen x
        comp (Pure x)        = return . push x
        comp (pf :<*>: px)   = pf <<< px · red ⟦$⟧
        comp ...
        comp (p :<|>: q)     = λk → do φ ← freshΦ @x
```

```
                              liftM2 (mkJoin φ k · catch) (p (commit (join φ)))
                                                   (fmap (tell · same) (q (join φ)))
comp (Branch b l r) = λk → do φ ← freshΦ @x
                              let k′ = red ⟦flip ($)⟧ (join φ)
                              liftM (mkJoin φ k) (b ≪ liftM2 kase (l k′) (r k′))
```

Mostly, the changes are ensuring that return and (≪) are used in appropriate places to make the types work. However, for the two rules that use join-points, a fresh φ is generated, and then the continuation k is bound to φ using a mkJoin. Then, join φ is used in place of k elsewhere in the rule.

Threading the Bindings

Now that both NT and Φ have been employed in the instructions they need to be accounted for during evaluation. Compared with the state Γ, which contains state relevant to the semantics of the language, the machinery for bindings is more related in nature to the specifics of the staging. As such, it is kept in a different structure, called Ctx, to keep the distinction clear:

```
type NTBound r a = Code (String → (String → r) → (a → String → r) → r)
type ΦBound r a   = Code (String → (a → String → r) → r)
data Ctx r = Ctx {nts :: DMap NT (NTBound r)
                 , φs  :: DMap Φ (ΦBound r)
                 }
```

The context Ctx contains two dependent maps: one for the bound non-terminals, and one for the bound join-points. The types of these bindings are code of functions: notice that NTBound is the same as the simplified type of Eval when given a full machine (at the end of §3.2.1); and join-points are the same but without a provided failure continuation – the relevant handler will already be in scope when the join-point is bound. To accommodate this new information, Eval has a Ctx added:

```
type Eval r xs n a = Ctx r → Γ r xs n a → Code r
```

For most instructions, the Ctx parameter is just passed through to continuations without any interaction at all. To be clear, here is the evalPush function altered to accommodate this change:

```
evalPush :: Code x → Eval r (x : xs) n a → Eval r xs n a
evalPush x k ctx γ{..} = k ctx (γ {moore = QCons x moore})
```

The only change is the additional argument ctx which is just passed on untouched to the continuation k. This is not a particularly insightful change, so the other trivially adjusted instructions are just omitted. More interesting is the effect on Call, which does interact with the context:

```
evalCall :: NT x → Eval r (x : xs) (Succ n) a → Eval r xs (Succ n) a
evalCall nt k ctx{..} γ{..} = ⟦$(nts DMap. ! nt) $input $h $ $ret⟧
```

```
  where Cons h _ = handlers
        ret      = ⟦λx input′ → $(k ctx (γ {input = ⟦input′⟧, moore = QCons ⟦x⟧ moore}))⟧
```

While the definition of the return continuation ret remains the same, the actual code for the call has changed. This time, the relevant bound non-terminal is pulled from the context, and then a call to this function is made, passing the next handler and the new return continuation.

```
evalJoin :: Φ x → Eval r (x : xs) n a
evalJoin φ ctx{..} γ{..} = let QCons x _ = moore in ⟦$(φs DMap. ! φ) $input $x⟧
evalMkJoin :: Φ x → Eval r (x : xs) n a → Eval r xs n a → Eval r xs n a
evalMkJoin φ body k ctx{..} γ{..} =
  ⟦ let join x input′ = $(body ctx (γ {input = ⟦input′⟧, moore = QCons ⟦x⟧ moore}))
     in $(k (ctx {φs = DMap.insert φ ⟦join⟧ φs}) γ)⟧
```

Both Join and MkJoin share similarities with Call. The Join instruction evaluates to the code of a function call to the bound join point collected from the φs map. The MkJoin instruction creates the code for a let-bound function join, the Code of which is inserted into the φs map for the continuation k. The body of the join function is the same as a return continuation, except it uses a let-expression instead of a lambda.

Setting up Non-Terminal Bindings

While the evalMkJoin function offers clues to how the bindings corresponding to non-terminal NTs might be generated, matters are complicated by the presence of mutual recursion in the bindings. Effectively, all the non-terminals should be generated at the same time, in the same let-rec block. The difficulty is that a let-rec block cannot be generated using typed Template Haskell unless the exact number of bindings and their names are known: this is not the case here. Yallop and Kiselyov [2019] explores this issue in the context of MetaOCaml, but an implementation of the primitive combinator they describe has not been brought to Haskell.

To construct a well-typed combinator that allows for mutually-recursive bindings to be generated, it is necessary to resort to *untyped* Template Haskell. Any untyped code will be converted back to "typed" code with unsafeCodeCoerce.

```
type CodeT f x = Code (f x)
letRec :: ∀k f1 f2 r.GCompare k ⇒ DMap k f1 → (∀x.k x → String)
       → (∀x.k x → f1 x → DMap k (CodeT f2) → CodeT f2 x)
       → (DMap k (CodeT f2) → Code r) → Code r
letRec bindings nameOf genBinding expr = unsafeCodeCoerce $
  do names ← DMap.traverseWithKey (λkey _ → makeName key) bindings
     let typedNames = DMap.map makeTypedName names
     let makeDecl key body =
       do let Const name = names DMap. ! key
          binding ← unTypeCode (genBinding key body typedNames)
          return (FunD name [Clause [ ] (NormalB binding) [ ]])
```

```
      decls   ← forallKeyVals makeDecl bindings
      exp     ← unTypeCode (expr typedNames)
      return (LetE decls exp)
  where makeName :: k x → Q (Const Name x)
        makeName key = fmap Const (newName (nameOf key))

        makeTypedName :: Const Name x → CodeT f₂ x
        makeTypedName (Const name) = unsafeCodeCoerce (return (VarE name))

        forallKeyVals :: Applicative m ⇒ (∀x.k x → f x → m b) → DMap k f → m [b]
        forallKeyVals f k = DMap.foldrWithKey (λkey val → liftA2 (:) (f key val)) (pure [ ])
```

The letRec combinator exposes a typed API for creating a let-rec block. There are four arguments: a map of (possibly mutually recursive) bindings; a function for turning a key into a string, which will serve as its name; a function to generate an individual binding given the final map of bindings; and a function to generate the body of the let-rec, again given the final map of bindings. The CodeT type reflects that the values produced must be Code, though the internal value may vary by the type f[3]. The combinator works by first using the makeName helper function to construct a new TH Name value for each of the bindings in the map before mapping over these names to create typed TH values instead by building a variable node and unsafe coercing it (this is internally how typed TH works, unfortunately). With both the names and typedNames available, the bindings map is traversed, and a binding is generated for each, first by using genBinding, which takes the typedNames map, then constructing a function-declaration node, FunD, with the raw name and the untyped binding – these are collated into a list. Once all the untyped decls have been produced, the typedNames are again passed to the expr function to build the body of the let, and then the decls and the new exp are stitched together into a LetE node. Taking CodeT f_2 x to be the type NTBound r x from earlier, the eval function can be modified to handle the recursive bindings:

```
 eval :: Machine a → DMap NT Machine → Code String → Code (Maybe a)
 eval m ms inp =
   letRec ms show (λnt m nts → ⟦λinp h ret → $(evalMachine m nts ⟦inp⟧ ⟦h⟧ ⟦ret⟧)⟧)
                  (λnts → evalMachine m nts inp fail accept)
   where fail   = ⟦λ_   → Nothing⟧
         accept = ⟦λx _ → Just x⟧

         evalMachine  ::  Machine x → DMap NT (NTBound r) → Code String → Code (String → r)
                      → Code (x → String → r) → Code r
         evalMachine m nts inp h ret =
           cata alg m (Ctx {nts = nts, φs = DMap.empty})
                      (Γ {input = inp, moore = QNil, handlers = Cons h Nil, retCont = ret})

         alg :: Instr (Eval r) xs n a → Eval r xs n a
         alg Ret      ctx γ = evalRet ctx γ
         alg (Push x) ctx γ = evalPush x ctx γ
```

[3]This is cheating a little with Haskell's type system and will not compile without a newtype and a little work on the caller side as well to formulate f, but it helps illustrate the idea without having to monomorphise. A generic version can be found in appendix B.

```
        alg ...
```

The eval function generates a let-rec binding for all the given non-terminals (where the show for NT is used to make the name), which makes a function taking the binding's input, failure handler, and return continuation and gives those to evalMachine, which performs the cata and feeds it a fresh context Ctx and state Γ. The body of the let is then similar, except the fail and accept continuations are provided as before.

Summary

Over the course of this section, a concrete denotational semantics has been given to `parsley`'s abstract machine; a careful untangling of the static and dynamic information was performed; and the resulting binding-time separations allowed for the staging of a completed machine, and its associated non-terminals, into a single Haskell function from String to Maybe a. This provides a sound basis for further development in CHAPTER 6: ANALYSIS AND OPTIMISATION, which will introduce additional analysis into the front-end and perform addition optimisations and binding-time improvements to improve the interpreter further. Discussion about the performance of this approach is deferred until after these improvements are introduced.

While there is still some discussion left to be had in this chapter, concerning the automatic detection of non-terminals (§3.3), it is useful to take stock of what the current end-to-end pipeline looks like, and what generated code for a simple example looks like post-sections 3.1 and 3.2.

The pipeline There are currently two components to the overarching system currently in place: a compile function, which converts Parsley values to Machine values; and an eval function, which converts a Machine into code that represents a Haskell function modelling that machine. Currently, it is expected that the user provides a mapping of NT values to their respective parsers. With all this in mind, the parse function is as follows:

```
parse :: Parsley a → DMap NT Parsley → Code (String → Maybe a)
parse p nts = dynFunc (eval m ms)
  where m  = compile p
        ms = DMap.map compile nts
```

First, both the main parser and all its non-terminals are compiled into Machines with the compile function. Then, these are passed into the eval function to produce a Code (Maybe a) when provided with a reference to the input (dynFunc was defined in §2.4). This is spliced in to produce the final function that is returned.

A compiled example Recall the example given to show the compilation of recursion to an automaton in §3.1.4:

```
nt₁ :: NT ()
p = char 'a' *> nt nt₁ <* char 'c'
q = char 'b' *> nt nt₁ <|> void (char 'a')   -- this is mapped to nt₁
```

For clarity, here is the direct-style version of the same parser:

```
p = char 'a' *> q <* char 'c'
q = char 'b' *> q <|> void (char 'a')
```

Now, applying the parse function will allow a peek behind the curtain to see the current state of the staging:

```
f = $(parse p (DMap.singleton nt₁ q))
  = λinput → let nt₁ inp h ret =
                    let join x input′ = ret x input′
                    in let handler input′ =
                      case (λnew old → if new ≡ old then Right () else Left ()) input′ inp of
                        Left () → h input′
                        Right () → case input′ of
                          c : cs | (≡ 'a') c → join () cs
                          _                  → h input′
                      in case inp of
                        c : cs | (≡ 'b') c → nt₁ cs handler (λx input′ → join x input′)
                        _                  → handler inp
              in case input of
                c : cs | (≡ 'a') c → nt₁ cs (λ_ → Nothing) $ λx input′ →
                    case input′ of
                      c : cs | (≡ 'c') c → (λx _ → Just x) x
                      _                  → (λ_ → Nothing) input
                _                  → (λ_ → Nothing) input
```

This is the precise code that would be generated for this parser, with no simplification performed at all. It is reasonable, with some obvious areas for improvement: firstly, the continuations in the "main body" of the parser could be β-reduced, which, as discussed, can easy be done by staging, but only applies in this one place; secondly, the join defined for nt_1 could be η-reduced and then inlined for simplicity, including in the return continuation of nt_1, which should also be η-reduced; and the logic to check the input is unchanged in the defined handler could be simplified and optimised, as it is quite expensive as it stands. Some of this will be done by GHC – though a meta-programmer should not really rely on that being the case.

```
f = $(parse p (DMap.singleton nt₁ q))
  = λinput → let nt₁ inp h ret =
                    let handler input′ = if input′ ≡ inp then
                                           case input′ of
                                             c : cs | c ≡ 'a' → ret () cs
                                             _                → h input′
                                         else h input′
                    in case inp of
                      c : cs | c ≡ 'b' → nt₁ cs handler ret
                      _                → handler inp
              in case input of
                c : cs | c ≡ 'a' → nt₁ cs (λ_ → Nothing) $ λx input′ →
```

```
        case input′ of
          c : cs | c ≡ 'c' → Just x
          _                → Nothing
      _                → Nothing
```

This code is normally what the generated code would look like after GHC has optimised it. Regardless, this is very low-level without any high-level abstraction present, which is likely to make it fast.

3.3 Automatic Let-Finding for Direct-Style Recursion

In §3.1.4, the type NT was introduced to denote the opaque names of recursive non-terminal parsers. This was assumed to happen manually, with the user of the library manually annotating parsers and packaging them up into a DMap NT Parsley for the compile function. In fact, identifying parsers to annotate with NT is the role of the *let-finding* (§3.3.1) and *let-insertion* (§3.3.2) phases of the `parsley` compiler. This is important for two reasons:

- As discussed in §3.1.4, parsers must be represented by a finite tree, and so any recursion in the parser must be made explicit to prevent infinite traversals.
- Referring the same parser in multiple places within a larger parser may create intolerable amounts of code growth and duplication.

The second point is more nuanced than the first; consider the definition of the some combinator:

```
some p = p <:> many p
```

In this combinator, p appears in two positions. Suppose that the p passed is a large parser: if the tree is simply incorporated by value into both positions it will be traversed twice during compile – this could result in a much larger amount of code. This is a problem identified by Willis and Wu [2018], where duplication in the combinators used to build expression parsers resulted in a 330× increase in the size of C expression parser.

3.3.1 Finding Let-Bound Parsers

The first task to factoring out let-bindings is to identify which parsers have been bound by let-expressions. In a language like Scala, where every object can be compared for referential equality, this is straightforward: simply traverse the tree and keep note of what references have been seen during the traversal as well as how many times. If the same reference is seen twice that must mean that the parser in question was bound to a variable. However, Haskell is not a language with referential equality.

Instead, Gill [2009] first suggested using GHC's StableName functionality to achieve this. A StableName is an opaque name assigned by the runtime on demand to a value: no matter how many times it is queried, the StableName is guaranteed to be the same and unique, even if the value moves in memory. The caveat is that one must be careful to force the evaluation of a value before asking for its StableName, because a thunk will have a separate name from the forced value. For the purposes of *let-finding*, the StableNames are wrapped up:

```
data ParserName = ∀a.ParserName (StableName (Parsley a))
```

The reason for this is to make the result type of the parser existential: it is necessarily the case that two equal StableNames would have the same underlying type, and it is a hindrance to have to rely on data structures that use the GEq typeclass, which also requires proof of type-equality. A ParserName can be sourced for a Parsley with the following function:

```
makeParserName :: Parsley a → ParserName
makeParserName !p = unsafePerformIO (fmap ParserName (makeStableName p))
```

Though this uses unsafePerformIO, this is a safe and referentially transparent function: it will always return the same result every time it is called, and p has been appropriately forced to ensure this.

Tagging Parsers with their `StableName`

The first step of the *let-finding* phase is to assign each node in the combinator tree its StableName. This is done by using the Tag indexed functor:

```
data Tag t f k a = Tag {tag :: t, tagged :: f k a}
instance IFunctor f ⇒ IFunctor (Tag t f) where ...
```

The idea is to perform a single traversal over the combinator tree, assigning a ParserName as a tag at each combinator. However, to generate the ParserName, access to the original combinator is required. This is like a paramorphism [Yang and Wu 2022], however traditionally, this reconstructs the original argument during the recursion. This is not fit for purpose here, because each occurrence of the Parsley will be reconstructed to a different value, making the StableNames useless. Instead, a small hack is required:

```
para′ :: Functor f ⇒ (Fix f → f a → a) → Fix f → a
para′ alg = let go i@(In x) = alg i (fmap go x) in go
```

This (non-indexed) recursion scheme realises the paramorphism in a slightly different way that provides the true original structure into the algebra and not just a copy. Like the other recursion schemes, it easily generalises to an indexed variant, so it is overloaded to also work for Combinator.

```
tagParser :: Parsley a → Fix (Tag ParserName Combinator) a
tagParser = para′ (In · Tag · makeParserName)
```

The tagParser function uses the new recursion scheme to tag each combinator node with its ParserName, leaving the content otherwise untouched – note that the identity cata is cata In. Now that the parsers have names they can be freely inspected. Crucially, it is possible to define instances for ParserName for both Eq and Hashable (but not Ord), so ParserName can be used with HashSet and HashMap[4].

Traversing the Tagged Combinator Tree

With a way to tag each combinator with its unique name, it is now possible to walk this tree and identify the sets of let-bound parsers, recursive or otherwise. In practice, it is beneficial to distinguish between the sets of

[4]Both HashSet and HashMap come from the `unordered-containers` package on Hackage.

recursive and non-recursive bindings so that simple inlining can be performed, but this is uninteresting and is omitted from the discussion.

The traversal makes use of the State monad to keep track of the number of occurrences each name has within the structure. It is implemented as a simple cata:

```
type Occurrences = HashMap ParserName Int
type LetFinder   = State Occurrences ()

findLets :: Fix (Tag ParserName Combinator) a → HashSet ParserName
findLets p = collectLets (execState (cata alg p) HashMap.empty)
  where collectLets =
          HashMap.foldrWithKey (λk n ls → if n > 1 then HashSet.insert k ls else ls) HashSet.empty
        alg :: Tag ParserName Combinator ( {- Const -} LetFinder) a → ( {- Const -} LetFinder)
        alg (Tag name p) = do
          seen ← gets (HashMap.member name)        -- has this parser ever been seen before?
          modify (HashMap.insertWith (+) name 1)   -- increment occurrences by 1, or set to 1
          when (not seen) $ do                     -- if this parser has not been processed before
            isequence_ p                           -- traverse the sub-tree
```

The algebra for findLets first checks whether the current name has ever occurred at all within any parents or left siblings of the currently inspected combinator – if it has it will exist in the Occurrences map. Then, it will either increment the occurrences count for the name if it has already been seen or set it to one otherwise with HashMap.insertWith. Finally, if the parser has never been processed before, then the sub-combinator will be explored in a similar way, otherwise it should be ignored: either it is recursive, and traversing will cause an infinite recursion, or any sub-combinators may also erroneously be given an artificially high occurrence count. The isequence_ function is an indexed variant of the sequence_ function, which sequences all the applicative actions found within a Traversable structure: Combinator can be made an instance of the equivalent ITraversable class. Once the full traversal over the tagged parser is complete, any keys in the resulting map with more than one occurrence are clearly let-bound parsers, the collectLets function bundles all of these into a HashSet.

3.3.2 Replacing the Parsers

With a method of finding a set of all the bindings within a parser, a *let-insertion* phase can be performed that replaces these bindings with a Let node. For this, values of type NT need to be made, so it is necessary to lift the veil from this type and understand exactly what it is:

```
newtype NT a = NT Word64
instance GEq NT where ...
instance GCompare NT where ...
```

In fact, it is represented just as Φ was in §3.2.4, including the same kind of instance for GEq and GCompare, still using unsafeCoerce Refl, but in a way that is known to be safe.

The insLets function is responsible for taking a tagged combinator tree and the set of let-bindings and factoring out the shared parsers into a dependent map and inserting Let nodes in the relevant places:

```
type LetInserter a = State (DMap NT Parsley) (Parsley a)

insLets :: HashSet ParserName → Fix (Tag ParserName Combinator) a → (Parsley a, DMap NT Parsley)
insLets lets p = runState (cata alg p) DMap.empty
  where words :: HashMap ParserName Word64
        words = HashMap.fromList (zip (HashSet.toList lets) [0 . .])

        alg :: ∀a.Tag ParserName Combinator LetInserter a → LetInserter a
        alg (Tag name p) = case HashMap.lookup name words of
          Just w    → do let nt = NT @a w
                         whenS (gets (not · DMap.member nt)) $ mdo
                           modify (DMap.insert nt p′)
                           p′ ← isequence p
                           return ()
                         return (Let nt)
          Nothing → isequence p
```

Like findLets, this function runs in the State monad, which here tracks the map from NT values to their corresponding factored parsers. First, the set of ParserNames known to correspond to let-bound parsers is converted into a map, words, from names to their unique words. The algebra for the cata looks up each tag in the words map: if it is not there, then the let-insertion is sequenced through the combinator and it is returned. If a name is found in the words map, then an NT value of the right type, nt, is constructed for it; if nt is already in the state a Let node is returned without further work, otherwise the parser must be processed with isequence and then inserted into the map. Crucially, however, this happens in reverse: first the fully processed parser is placed into the state and then it is processed – this is possible via GHC's "recursive do" [Erkök and Launchbury 2002] extension. This is necessary because a parser should only be traversed the very first time it is encountered, otherwise recursive parsers would infinitely try and insert themselves into the map; it works because the processed parser p′ is lazy and only the existence of the key is queried during isequence p, leaving p′ unevaluated during self-processing.

Putting it all together With the tagParser, findLets, and insLets functions defined it is possible to refine the parse function defined at the end of §3.2:

```
parse :: Parsley a → Code (String → Maybe a)
parse p = dynFunc (eval m ms)
  where tp        = tagParser p
        (p′, nts) = insLets (findLets tp) tp
        m         = compile p′
        ms        = DMap.map compile nts
```

Compared with the original parse function, this version no longer requires the user to pass a mapping of DMap NT Parsley explicitly, which simplifies the act of writing parsers. Other than the fact that direct-style recursion is now supported, there is no change to how the code would be generated. An alternative approach would be to have a dedicated fixpoint combinator [Lickman 1995], but this comes at an ergonomic cost.

Summary

This chapter has provided an end-to-end compiler and staged interpreter for a selective parser combinator library called `parsley`. Working from a representation of plain regular expressions and non-deterministic finite automata §3.1 described how to translate from one to the other via a Cayley transformation. It then showed how to extend this machinery incrementally through to fully recursive selective parser combinators with PEG/`parsec`-style backtracking semantics. The result is a deterministic heterogeneous push-down moore machine, which consists of two stacks – one for results and the other for control.

Section 3.2 provided an evaluation semantics for this machine, adjusting the representation slightly to make the evaluation more straightforward, showing how this corresponds cleanly to the classic CPS-style parser combinator representation. The evaluator is then staged to allow for the elimination of the overheads of both the compiler and interpreter itself, which can now all run at compile-time.

From here, Chapter 4: Context-Sensitive Parsing with References expands the expressive power of the library by adding mutable parsing state, discussing its implementation and how it may be used to improve the definitions of some combinators and perform context-sensitive tasks. Then, Chapter 6: Analysis and Optimisation implements various improvements over the base implementation to refine the generated code. The performance of this library is evaluated in §6.5.

Chapter 4

Context-Sensitive Parsing with References

While `parsley`, as described so far, can parse some context-sensitive languages either via selective combinators (§2.2.4) or by the PEG lookahead operations (§2.1.2 and §2.3.2), it is not capable of full context-sensitive parsing, which is normally unlocked via monadic combinators.

Instead, a register machine abstraction is introduced on top of the existing stack-based architecture. The idea is that any number of well-scoped "references" can be created and used to store arbitrary state during parsing. These references will be threaded through the generated parser and allow for straightforward access to Turing-powerful parsing. Notably, Mokhov et al. [2019] suggest that one limitation of selective functors is that they cannot implement join, even with access to "The Trick" (§2.4.1). However, with enough perseverance, it is possible to implement an *interpreter* for join with the references machinery in conjunction with selective operations; note, however, this does not make `parsley` a monad. The structure of this chapter is as follows:

- First, §4.1 describes the high-level API for references, along with examples of how it can be used to realise context-sensitive grammars cleanly.
- Then, §4.2 describes how the so-called *iterative combinators* can be reformulated to use references and one tail-recursive combinator: this will be optimised in CHAPTER 6: ANALYSIS AND OPTIMISATION.
- The additional changes required to the existing frontend to accommodate references is detailed in §4.3, including the translation to abstract machine instructions.
- The required infrastructure required in the backend staging to integrate references with the existing machinery is described in §4.4, along with a frontend analysis pass required to do so.
- Finally, an informal demonstration of the Turing-completeness of selectives and references is given in §4.5 by providing the implementation of a full interpreter for a Turing-complete language in `parsley`.

4.1 Reference Combinators

There are two primary inspirations behind the design of the reference combinators: the first is the classic API for the State monad – the operators get and put – and the second is the ST monad [Launchbury and Peyton Jones 1994]. The ST monad allows for fully mutable references, STRefs, that are safely encapsulated within the monad: it is not possible for such references to leave the ST monad, and by running out the ST monad one is forced to give up their references permanently. This differs from the IORef, which, while similar[1], is not constrained to exist only within the scope of IO and may escape if unsafePerformIO is used: runST, on the other hand, is safe.

How ST works: rank-2 types ST requires all its operations and data to be associated with a type s:

[1]In fact, IORef and STRef are the same, as **type** IO = ST RealWorld, which subverts the safety of ST by monomorphising its state token.

```
newSTRef   :: a → ST s (STRef s a)
writeSTRef :: a → STRef s a → ST s ()
readSTRef  :: STRef s a → ST s a
```

In each of these types, the type s is present, but never interacts with anything: there are no values of type s, but the s must match up across different components. Instead, the type a is what the programmer gets to specifically pick at any given time, for instance: STRef s Bool or STRef s Int. The magic comes from the runST function:

```
runST :: (∀s.ST s a) → a
```

The parenthesisation here is crucial: this creates a scope for s which cannot be chosen by the caller of runST, instead only by the callee. This is known as a rank-2 type [McCracken 1984]. This means that a value of type ST s (STRef s a) cannot be provided to runST, as the return type would be STRef s a, and the caller has "chosen" the type of s. In fact, trying to do this results in the following error:

```
ghci> x = newSTRef True
ghci> :t x
x :: ST s (STRef s Bool)
ghci> runST x

<interactive>:9:7: error:
    * Couldn't match type ‘a’ with ‘STRef s Bool’
      Expected: ST s a
        Actual: ST s (STRef s Bool)
        because type variable ‘s’ would escape its scope
      This (rigid, skolem) type variable is bound by
        a type expected by the context:
          forall s. ST s a
```

How `parsley` differs `Parsley`'s reference mechanism is inspired by a similar idea, but more fine-grained: instead of a reference being bound to an entire parser (which would have to be parameterised by some type r), each reference is bound to the specific combinator that brought it into existence. Concretely, given the opaque type Ref r a:

```
newRef :: Parser a → (∀r.Ref r a → Parser b) → Parser b
```

This is notably a CPS-transformed variant of newSTRef, which takes a continuation within which the created reference can be used but, due to the rank-2 type, cannot escape. The use of CPS here means that the Parser type does not need to be parameterised by the r, and the reference does not need to be returned in Parser r (Ref r a), which would hinder its use. As a result, `parsley`'s references do not share the same type r, in fact they are all uniquely scoped. The first argument of newRef being a Parser as opposed to a pure value is a common theme within the API: without a (»=) combinator is not possible to extract a pure value from a parser to feed into newRef, which would make it heavily restricted. Instead, pure versions are also given:

$\text{newRef\_} :: \text{Code a} \to (\forall r.\text{Ref r a} \to \text{Parser b}) \to \text{Parser b}$
$\text{newRef\_} = \text{newRef} \cdot \text{pure}$

4.1.1 Interacting with State

So far, a mechanism for creating references has been given, but no combinators for interacting with a reference. The API for references is inspired by the classic operations of the State monad, get and put:

$\text{get} :: \text{State s s}$ $\text{put} :: \text{s} \to \text{State s ()}$

These are subject to the State laws [Gibbons and Hinze 2011], which offer sensible interactions between the operations:

$$\text{get} \gg\!= \text{put} = \text{pure ()} \qquad (4.1)$$
$$\text{put x} *\!> \text{get} = \text{put x} *\!> \text{pure x} \qquad (4.2)$$
$$\text{put x} *\!> \text{put y} = \text{put y} \qquad (4.3)$$
$$\text{f} <\$> \text{get} <*> \text{get} = (\lambda \text{x} \to \text{f x x}) <\$> \text{get} \qquad (4.4)$$

However, the type of put is restrictive: observe that of the four laws, eq. (4.1) requires monadic combinators to express. This is because put as presented requires a pure value, and as mentioned previously, this necessitates the use of ($\gg\!=$) unless a constant value is already providable. Instead of monadic-style state, `parsley` adopts an applicative-style state, where the write operation must take its payload from a parser directly:

$\text{read} :: \text{Ref r a} \to \text{Parser a}$ $\text{write} :: \text{Ref r a} \to \text{Parser a} \to \text{Parser ()}$

Obviously, the addition of multiple threaded references means that read and write are also parameterised by the reference they act on. As with newRef, a pure version of write, called write_ is provided for convenience. With these two combinators, the state laws can be reformulated:

$$\text{write r (read r)} = \text{pure ()} \qquad (4.5)$$
$$\text{write\_ r x} *\!> \text{read r} = \text{write\_ x} *\!> \text{pure x} \qquad (4.6)$$
$$\text{write r p} *\!> \text{write\_ r x} = \text{p} *\!> \text{write\_ r x} \qquad (4.7)$$
$$\text{f} <\$> \text{write r} <*> \text{write r} = (\lambda \text{x} \to \text{f x x}) <\$> \text{write r} \qquad (4.8)$$

Compared to the simple elegance of the original laws, these are slightly more restrictive: both eqs. (4.6) and (4.7) assume the purity of the argument to the write combinator. For a specific r, the new reference laws can be derived in terms of the original state laws by assuming that write p = p $\gg\!=$ put.

4.1.2 Composite Combinators

With the three primitive combinators newRef, read, and write, a whole host of composite combinators can be defined. These are either for convenience or allow for the elegant description of context-sensitive workloads.

Basic state combinators The following two combinators are both analogous combinators to the composite equivalents usually provided by the State monad:

$\text{reads} :: \text{Ref r a} \to \text{Parser (a} \to \text{b)} \to \text{Parser b}$
$\text{reads r p} = \text{p} <*> \text{read r}$

```
modify :: Ref r a → Parser (a → a) → Parser ()
modify r = write r · reads r
```

A replacement for (»=) One of the main things lost by restricting a library to no longer have a (»=) combinator is the ability to easily persist results or use them multiple times. However, this is something that references can achieve with ease:

```
bind :: Parser a → (Parser a → Parser b) → Parser b
bind p f = newRef p (f · read)   -- equivalent to p »= (f · pure)
```

Compared with the sbind combinator for exhausting the domain of a type to refine dynamic information into static structure, the reference version of bind allows for the distribution of a statically known location where a value can be found to multiple parts of a wider static structure. This ends up being a useful combinator.

For-loops as a combinator One interesting combinator with a direct application to context sensitive parsing is the forP combinator:

```
forP :: Parser a → Parser (a → Bool) → Parser (a → a) → (Parser a → Parser b) → Parser [b]
forP init cond step body = n
  newRef_ ⟦mempty⟧ $ λdxs →
       newRef init (λi →
         let cont = reads i cond
         in  whenS cont (whileS (modify dxs (⟦diffSnoc⟧ <$> body (read i)) *> modify i step *> cont)))
     *> reads_ dxs ⟦fromCayley⟧
```

This combinator embodies the structure of a classic C-style `for`-loop, with a single variable of type a initialised with init, a termination condition cond, and an iteration step step; the body of the loop can make use of the variable but may not alter it, and the results of the body are collected up in a difference list (called dxs above).

One use of this combinator is helping to parse the classic context-sensitive language $a^n b^n c^n$ in a single pass:

```
as_bs_cs :: Parser ([Char], [Char], [Char])
as_bs_cs = newRef_ ⟦0⟧ $ λn →
  ⟦(,,)⟧ <$> many (char 'a' <* modify_ n ⟦(+ 1)⟧)
        <*> downTo0 n (char 'b')
        <*> downTo0 n (char 'c')
         <* eof
    where downTo0 n p = forP (read n) (pure ⟦(> 0)⟧) (pure ⟦subtract 1⟧) (λi → p)
```

This parser starts by first parsing zero or more of the character `'a'`, incrementing the counter n each time; then it will read n copies of `'b'` followed by another n copies of `'c'` – this is done by writing the forP combinator equivalent of `for (int i = n; i > 0; i--)`. Although it was already possible to parse this grammar using backtracking, this implementation is more efficient.

4.2 References for Iterative Combinators

The forP combinator defined in the previous section hints at an alternative approach to implementing so-called *iterative combinators*. These combinators repeatedly parse something and often collect up the results into a list. They are characterised, however, by their tail-recursive shape, modulo the collecting of results.

4.2.1 What do Iterative Combinators Look Like?

As an example of the shape, consider the skipMany combinator, which does not collect results:

```
skipMany :: Parser a → Parser ()
skipMany p = let go = (p *> go) <|> unit in go
```

The invocation of the go parser is in a tail-position in the left-hand side of (<|>): to be precise, the generated machine for this parser would contain call $\mathrm{nt}_{\mathrm{go}}$ ret – this is exactly what is conventionally known as a tail-call[2]. Not only can this tail-call be optimised in the same way it might be in a GPL like Haskell or C, the failure characteristics of this combinator are idempotent: unwinding the (<|>) combinators on failure will have the same result regardless of how many iterations have been performed. The fact that this idempotency can be exploited is discussed in CHAPTER 6: ANALYSIS AND OPTIMISATION, but regardless, this general shape is interesting.

However, while the many combinator enjoys the same idempotency of the failure handlers, it does not have the same tail-call because of the need to combine the result list. If would be nice if that were not the case, since this would allow for many to benefit from the same kind of optimisations that skipMany enjoys.

Generalising `skipMany` The first step towards the goal of unifying all the iterative combinators is to generalise the shape of skipMany slightly, for convenience:

```
loop :: Parser a → Parser b → Parser b
loop p exit = let go = (p *> go) <|> exit in go

skipMany p = loop p unit
```

The loop combinator is the archetypal iterative combinator: it does p as many times as possible, then finishes by doing exit: this returns the meaningful result for the combinator. By combining this combinator with references, it is possible to give definitions for the other iterative combinators, like many.

4.2.2 Using References to Collect Results

In §4.1, the forP demonstrated how references can be used to collect up results of a complex parser without having to carefully reformulate it to feed results applicatively. It is exactly this idea that can be used to help reformulate iterative combinators using loop. To give a sense of what this involves, here is the many combinator:

```
many p = newRef_ ⟦mempty⟧ (λdxs → loop (modify dxs (⟦diffSnoc⟧ <$> p)) (reads_ dxs ⟦fromCayley⟧))
```

[2]Strictly it would be call $\mathrm{nt}_{\mathrm{go}}$ (commit ret) but commit ret ≅ ret, which is true due to stack capture in return continuations.

This definition makes a difference list and snocs a new element onto this list on every iteration. At the exit point, this list can be forced into a regular list. As demonstrated in CHAPTER 7: THE DESIGN PATTERNS OF PARSER COMBINATORS, it is possible to implement many as a "parser fold," which in turn makes use of the prefix or postfix combinators: these are two combinators that capture this pattern very nicely.

```
postfix :: Parser a → Parser (a → a) → Parser a
postfix p op = newRef p $ λacc → loop (modify acc op) (read acc)

prefix :: Parser (a → a) → Parser a → Parser a
prefix op p = newRef_ ⟦id⟧ $ λacc → loop (modify acc (⟦flip (·)⟧ <$> op)) (read acc <*> p)
```

The postfix combinator is the archetypal example of a tail-recursive iteration, the kind that would be realised by a foldl, where the accumulator is plainly modified during the iteration. The prefix combinator, on the other hand, represents a regular recursive reduction, rendered in tail-recursive form by introducing a function as the accumulator. In fact, prefix can be implemented as a postfix in the same way foldr is implementable as foldl:

```
prefix op p = postfix (pure ⟦id⟧) (⟦flip (·)⟧ <$> op) <*> p
```

As it happens, most iterative combinators can be implemented in terms of postfix (via prefix or otherwise), including many: its simple definition in terms of loop means that it is very amenable to optimisation, which is a strength. It should be possible to optimise away the references here in the backend and reduce it to a tight loop with an accumulation parameter: this is left to future work, however.

4.3 Compiling References

The high-level API for references was discussed in §4.1, however, it provides no insights in how references should be represented within the combinator tree, or indeed in the abstract machine. While the underlying implementation of references is the topic of §4.4, this section discusses what changes and additions are made to the `parsley` frontend to support the new machinery. Until now, Ref r a has been treated as an opaque type. It is defined as follows:

```
newtype Ref r a = Ref (ΣVar a)
newtype ΣVar a = ΣVar Word64

instance GEq ΣVar where ...
instance GCompare ΣVar where ...
```

This is like how NT and Φ are defined: a Ref just wraps a ΣVar with a phantom type parameter r used for the scoping, and a ΣVar associates a type with a unique Word64. The implementations of GEq and GCompare are the same as those for NT and Φ, including the unsafeCoerce Refl.

4.3.1 Syntactic Representation of References

Unlike other combinators, representing references syntactically is made more complex by the scoping in the newRef combinator. The presence of a function that results in static structure blocks information from flowing up through the combinator. As such, the newRef constructor is kept as a separate syntactic functor:

```
data ScopeRef k a where ScopeRef :: k a → (∀r.Ref r a → k b) → ScopeRef k b
```

In addition, the Combinator syntactic functor is augmented with three additional constructors:

```
MakeRef :: ΣVar a → k a → k b → Combinator k b
ReadRef  :: ΣVar a → Combinator k a
WriteRef :: ΣVar a → k a → Combinator k ()
```

The MakeRef constructor replaces the ScopeRef constructor when the reference has been created and fed through the parser. While it loses the scoping information, the reference use must have been well-scoped to begin with and there can be no occurrences of its reference outside of the k b continuation.

The type exposed to the user API for `parsley` is now the co-product of these two syntactic functors:

```
type Parser a = Fix (Combinator :+: ScopeRef) a
```

This is distinct from the Parsley type seen so far in this dissertation: it is important to eliminate the problematic ScopeRef syntax as soon as possible and get back to pure Parsley. This is done by changing the tagParser function from §3.3.1 so that it creates concrete Ref r a values and removes the ScopeRef:

```
tagParser :: Parser a → Fix (Tag ParserName Combinator) a
tagParser p = para′ (In · Tag (makeParserName p) ▽ descope) p
  where descope  ::  ScopeRef (Fix (Tag ParserName Combinator)) a
                 → Fix (Tag ParserName Combinator) a
        descope (ScopeRef p f) = freshRef refSource (λref@(Ref σ) → MakeRef σ p (f ref))

        refSource :: IORef Word64
        refSource = newRefSource p
```

The change to tagParser is the addition of a separate algebra for handling ScopeRef: this makes use of the mysterious freshRef function, which given an IORef and a continuation expecting a Ref r a, generates a reference and provides it to the continuation[3]. The definitions of freshRef and newRefSource are as follows:

```
 {-# NOINLINE newRefSource #-}
newRefSource :: a → IORef Word64
newRefSource x = x `seq` unsafePerformIO (newIORef 0)

 {-# NOINLINE freshRef #-}
freshRef :: IORef Word64 → (∀r.Ref r a → x) → x
freshRef source scope = scope $ unsafePerformIO $
  do x ← readIORef source
     writeIORef source (x + 1)
     return (Ref (ΣVar x))
```

These functions are exceedingly delicate: both are marked as `NOINLINE` to ensure GHC does not make any attempt to optimise them within the call-site. This is important as both make use of unsafePerformIO. The newRefSource

[3]The reason a continuation is used is to keep GHC happy with the scope of the type r.

function just makes an IORef, but needs to take a seemingly useless argument x, which is sequenced beforehand: this is used to ensure that GHC is not able to hoist the IORef out of the tagParser function by forming a hard data-dependency: this ensures a unique IORef for each independent parser that is compiled in the same file. The freshRef function defines an atomic operation, where the reading and writing to the IORef must happen together: it does not matter in which order references are allocated, so long as the counter updates atomically.

This is necessary because the use of the State monad, or the "full" IO monad will cause the traversal to diverge across any recursive calls as it tries to establish the next state or RealWorld token. This is avoided with Reader, but this only produces locally unique names: as the soundness of the unsafeCoerce in GEq relies on global uniqueness, the unsafePerformIO is the safest option.

4.3.2 Translating References to the Abstract Machine

With the high-level syntax settled, the next step is to consider the mapping down to the abstract machine. This is a simple translation, following the structure of other combinators like Satisfy, which are just simply CPS-transformed into their instruction equivalents:

```
Make :: ΣVar x → k xs n a → Instr k (x : xs) n a
Read  :: ΣVar x → k (x : xs) n a → Instr k xs n a
Write :: ΣVar x → k xs n a → Instr k (x : xs) n a
```

With the appropriate rules added to the comp algebra within the unchanged compile function:

```
comp (NewRef σ p body) = p · make σ ⋘ body
comp (ReadRef σ)       = return · read σ
comp (WriteRef σ p)    = p · write σ · push ⟦()⟧
```

Each added rule is self-explanatory. By far, the more complex part of the reference implementation is the staging.

4.4 Staged Evaluation of References

With the high-level combinators and low-level instructions of references described, it is time to give a concrete runtime model for references as well as give semantics to the reference instructions. Code changes during this section will continue to be highlighted.

Modelling References The most obvious candidate to represent SigmaVar at runtime would, unsurprisingly, be the STRef. Both represent mutable state, with similar underlying design principles. To do this, the Eval type will need to change to accommodate the integration of the ST monad into the parsing system:

```
type Eval r s xs n a = Ctx r s → Γ r s xs n a → Code (ST s r)
```

Notice that Ctx and Γ have also inherited the type variable s: this is because the types of the underlying components now need to refer to Code (ST s r) instead of Code r in all cases. This is a straightforward substitution, so is omitted for brevity. In fact, the natural CPS structure of the instructions means that all the current evaluation functions are unaffected by this change. It is only the type signatures that need to change, however, the eval function will be altered slightly in due course (§4.4.4).

Concrete Representation With the model established, how do ΣVars – or rather STRefs – get to the parts of the generated code they need to be in? Their appearance in the generated code is more of a static artefact rather than a dynamic one (the ST s itself is the dynamic state), so STRefs will be kept in the static context Ctx:

```
type NTBound r s a = Code (String → (String → ST s r) → (a → String → ST s r) → ST s r)
type ΦBound r s a  = Code (String → (a → String → ST s r) → ST s r)
type QSTRef s a    = Code (STRef s a)

data Ctx r s = Ctx {nts  :: DMap NT (NTBound r s)
                   , φs   :: DMap Φ (ΦBound r s)
                   , refs :: DMap ΣVar (QSTRef s)
                   }
```

The new component, refs, tracks the in-scope references and provides a mapping from their abstracted names to the reference of the STRef that concretely backs it. This is enough to now give a semantics for the Get, Put, and Make instructions (§4.4.1).

Closure Issues One thing that is important to consider is the fact that any given non-terminal can be defined such that it closes over a defined reference. This is not uncommon, and has appeared in practice already. Consider the postfix combinator from earlier, with the definition of loop inlined:

```
postfix :: Parser a → Parser (a → a) → Parser a
postfix p op = newRef p $ λacc → let go = modify acc op *> go <|> read acc in go
```

The non-terminal go, which will be picked up by *let-finding* (§3.3), refers to the reference acc. This is indeed well-scoped, as it has been defined within newRef's continuation. However, post-flattening, acc is given a name, and the go non-terminal is hoisted up away from the newRef call itself: this means is that go is *free* in the acc reference. This is not a problem, and §4.4.4 will be able to provide free-references to bindings before being called, but it will also be necessary to analyse all non-terminals to find their transitive set of free-references (§4.4.3). This will have an impact on how the eval function is defined.

4.4.1 The Semantics of References

The evaluation functions for each of the three reference instructions are the only ones that interact with the ST monad, which is now threaded throughout the parser.

```
evalRead :: ΣVar x → Eval r s (x : xs) n a → Eval r s xs n a
evalRead σ k ctx{..} γ{..} =
  ⟦ do x ← readSTRef $(refs DMap.! σ)
       $(k ctx (γ {moore = QCons ⟦x⟧ moore}))⟧
```

```
evalWrite :: ΣVar x → Eval r s xs n a → Eval r s (x : xs) n a
evalWrite σ k ctx{..} γ{..} =
  let QCons x xs = moore
  in ⟦ do writeSTRef $(refs DMap.! σ) $x
          $(k ctx (γ {moore = xs}))⟧
```

Both Read and Write collect the relevant STRef from the refs map within the context and perform readSTRef and writeSTRef respectively using it. This operation is sequenced monadically with the continuation k.

```
evalMake :: ΣVar x → Eval r s xs n a → Eval r s (x : xs) n a
evalMake σ k ctx{..} γ{..} =
  let QCons x xs = moore
  in ⟦ do ref ← newSTRef $x
          $(k (ctx {refs = DMap.insert σ ⟦ref⟧ refs}) (γ {moore = xs}))⟧
```

The Make instruction is like Write, except that it also updates the refs map, so it contains a reference to the newly created STRef.

4.4.2 Lambda Lifting

References can be defined at any point in a parser, including externally to sub-parsers. This means that some sub-parsers refer to a reference by closure: *let-insertion* analysis lifts these bindings up to the top level, where the reference is unacceptably no longer in scope.

This is like the problem of *lambda lifting* [Johnsson 1985; Danvy and Schultz 2002; Morazán and Schultz 2008], where functions in a functional program can be lifted to the top-level but must account for any free variables within the functions body. The idea is that before a function can be lifted to the top-level, it must be augmented with additional arguments – one for each free variable – and all its call-sites adjusted to provide bindings that were previously captured but would no longer be in scope post-lifting. As an example:

```
let f x =
  let g y = x + y
      h y = g y ∗ y
  in h (x + 2)
in ...
```

The functions g and h may be lifted to the top-level only if they are modified so that all the free variables are provided explicitly. In this case, the obvious free variable is g's reference to x: post-transformation, g y should become g x y, and all calls to g must now account for x. However, in doing this, the body of h now becomes g x y ∗ y, which means that h now has a free variable x: the process of removing free-variables from functions can cascade to introduce free-variables to others, and this is solved by iteration. The final transformed program should be:

```
let f x   = h x (x + 2)
    g x y = x + y
    h x y = g x y ∗ y
  in ...
```

The original solution to this problem by Johnsson [1985] is an iterative process, where the free-variables for all functions are eliminated, and each call-site is augmented, before this process repeats until no functions have any free-variables remaining; this is a $O(n^3)$ algorithm in the number of bindings n. Danvy and Schultz [2002] improved on the complexity, giving an $O(n^2)$ algorithm that approximates the optimal binding requirements by coalescing strongly connected groups of functions in a call graph. Morazán and Schultz [2008] further refine these approaches into a $O(n^2)$ algorithm that computes the optimal distribution of arguments; their algorithm relies on dominance trees, which can be computed in $O(n)$ [Alstrup et al. 1999].

Free-reference analysis and lambda lifting are equivalent problems, assuming that each binding is augmented with extra arguments for each of its free references (see §4.4.4), so it is possible to perform this factoring in $O(n^2)$.

4.4.3 Free-Reference Analysis

The process of finding the complete transitive set of free references for all non-terminals proceeds in five steps:

1. Identify which of the non-terminals are directly reached from the top-level, called the *roots* (§4.4.3).
2. Analyse each non-terminal to determine the references it defines and uses, as well as which other non-terminals it calls directly (§4.4.3).
3. Construct a call(ee) graph as well as a reversed callers graph (appendix C.1).
4. Establish a topological ordering for this graph by assigning each non-terminal a unique *df-num* representing their visit order in a depth-first traversal starting from each of the roots (appendix C.2).
5. Propagate the free references of each non-terminal transitively by iteration until a fixed-point (appendix C.3).

These will be used to construct a function called freeReferences:

```
freeReferences :: Parsley a → DMap NT Parsley → Map Word64 (Set (Some ΣVar))
```

This function will be defined later, though it is useful to discuss up-front what Some ΣVar is. The type Some f for f :: ∗ → ∗ is type that captures existential knowledge about f: Some f can be thought of as "there exists an x such that f x." In this case, Some ΣVar says that there is a reference, but the type is not exposed. It is defined simply as:

```
data Some f = ∀x.Some (f x)
```

Instances for Eq (Some ΣVar) and Ord (Some ΣVar) are assumed.

Finding *Roots*

Identifying the *root* non-terminals is a cata over the top-level parser to collect up all the referenced NT values. This will be required later, so the algebra is parameterised by a Maybe Word64 representing the name of the parser being analysed: Nothing for the top-level, and Just nt for other non-terminals.

```
type Calls = State (Set Word64) ()

roots :: Parsley a → Set Word64
roots p = execState (cata (callAlg Nothing) p) Set.empty

callAlg :: Maybe Word64 → Combinator ( {-Const -} Calls) a → {-Const -} Calls
callAlg (Just v) (Let nt@(NT u)) = unless (u ≡ v) (calls nt)
callAlg Nothing (Let nt)         = calls nt
callAlg _       p                = isequence_ p

calls :: NT a → Calls
calls (NT w) = modify (Set.insert v)
```

The roots function passes Nothing to the callAlg algebra to denote that this is from the perspective of the top-level parser. The callAlg traverses the Combinator structure by looking for Let nodes that are not the same as the enclosing parser: this is to avoid self-calls being added to the call graph later, since these are redundant. For non-Let combinators, the monad is sequenced through in a similar way to the *let-finding* code.

Analysing Non-Terminals

Free-reference analysis needs to establish three things about each non-terminal: the set of references it uses, the set of references it defines, and the set of non-terminals it calls directly. This is also a cata, and the existing callAlg can be reused for this purpose:

```
analyseNTs :: DMap NT Parsley → (Map Word64 (Set (Some ΣVar))
                                , Map Word64 (Set (Some ΣVar))
                                , Map Word64 (Set Word64))
analyseNTs = DMap.foldrWithKey f (Map.empty, Map.empty, Map.empty)
  where
    f (NT w) p (used, defs, calls) =
      let (us, ds, cs) = analyseNT w p
      in (Map.insert w us used, Map.insert w ds defs, Map.insert w cs calls)

analyseNT :: Word64 → Parsley a → (Set (Some ΣVar), Set (Some ΣVar), Set Word64)
analyseNT w p = (uses, defs, calls)
  where (usesDefsM, callsM) = zipper useDefsAlg (callAlg (Just w)) p
        (uses, defs)        = execState usesDefsM (Set.empty, Set.empty)
        calls               = execState callsM Set.empty
```

The analyseNTs function applies the analysisNT function across all the non-terminals, accumulating the results up into three maps, which will be used in the coming steps. The analyseNT function applies a cata using the zipper scheme, which allows for the simultaneous computation of two results within one traversal. Both algebras return monadic programs, which are executed to get the relevant results. The algebra callAll is provided with the name of the non-terminal this time to prevent self-calls. The useDefsAlg is as follows:

```
type UseDefs = State (Set (Some ΣVar), Set (Some ΣVar)) ()
```

```
useDefsAlg :: Combinator ( {-Const -} UseDefs) → {-Const -} UseDefs
useDefsAlg (ReadRef σ)       = uses σ
useDefsAlg (WriteRef σ p)    = uses σ ≫ p
useDefsAlg (MakeRef σ p q)   = defines σ ≫ p ≫ q
useDefsAlg p                 = isequence_ p

uses :: ΣVar a → UseDefs
uses σ = modify (first (Set.insert (Some σ)))

defines :: ΣVar → UseDefs
defines σ = modify (second (Set.insert (Some σ)))
```

This is a simple traversal over Combinator that inspects the nodes that interact with references. The uses and defines helper functions insert encountered references into the right set.

Putting it Together

With all above steps defined (including those in appendix C), the definition of freeReferences can be given:

```
freeReferences :: Parsley a → DMap NT Parsley → Map Word64 (Set (Some ΣVar))
freeReferences p nts = propagate dfnums uses defs callees callers
  where rs                  = roots p
        (uses, defs, calls) = analyseNTs nts
        callees             = buildCallGraph calls
        callers             = invert callees
        dfnums              = topoOrdering roots callees
```

This function can be used to provide the transitive-free references for any non-terminal. However, Set (Some ΣVar) is not the best representation of this information. Instead, the results in the map can be transformed by combining the existential references under one existential as opposed to many:

```
data Refs rs where
  NoRefs  :: Refs '[ ]
  FreeRef :: ΣVar r → Refs rs → Refs (r : rs)
```

This type is a heterogeneous list of references, where the collective set is described by the type-level list rs. The function makeRefs can be used to construct an existentially bound version of this type:

```
makeRefs :: Set (Some ΣVar) → Some Refs
makeRefs = Set.foldr (λ(Some σ) (Some σs) → Some (FreeRef σ σs)) (Some NoRefs)
```

By unpacking each existential in turn, the information can be combined into a single existential Refs rs value, which will be useful moving forward. The freeReferences function is adjusted as follows:

```
freeReferences :: Parsley a → DMap NT Parsley → Map Word64 (Some Refs)
freeReferences p nts = Map.map makeRefs (propagate dfnums uses defs callees callers) where ...
```

This information will be required for the eval function, so the overall pipeline is adjusted to feed it through:

```
parse :: Parser a → Code (String → Maybe a)
parse p = ⟦λinput → $(eval (compile p′) (DMap.map compile nts) refs ⟦input⟧)⟧
  where tp       = tagParser p
        (p′, nts) = insLets (findLets tp) tp
        refs     = freeReferences p′ nts
```

Note that the input type of parse is now Parser a, reflecting the syntactic presence of references, though this is refined into Parsley by the time any compilation or other analysis happens.

4.4.4 Threading References into Bindings

With access to a map of the free references for each non-terminal, the references can be threaded correctly into bindings during staging. The idea is that free references will be passed into bindings as arguments, and these will be provided from the context when a call is made. To do this, the type of bindings in the context will change, since each binding has a different number of required references and therefore different types. Following a similar approach as §2.4.1 with the Arity type family [Schrijvers et al. 2008], bindings will be represented with a type-family that tracks the types of the free references:

```
type BaseFunc r s a = String → (String → ST s r) → (a → String → ST s r) → ST s r
type family Func xs r s a where
  Func '[ ]      r s a = BaseFunc r s a
  Func (x : xs) r s a = STRef s x → Func xs r s a
```

The BaseFunc type is the original type of a binding, and Func extends BaseFunc with zero or more references of given types. Since the Func type has an arbitrary number of provided references, there needs to be functions for getting from and to the common BaseFunc type:

```
takeRefs  :: DMap ΣVar (QSTRef s) → (DMap ΣVar (QSTRef s) → Code (BaseFunc r s a))
          → Refs xs → Code (Func xs r s a)
takeRefs refs f NoRefs          = f refs
takeRefs refs f (FreeRef σ σs) = ⟦λref → $(takeRefs σs (DMap.insert σ ⟦ref⟧ refs) f)⟧
giveRefs  :: Code (Func xs r s a) → DMap ΣVar (QSTRef s) → Refs xs → Code (BaseFunc r s a)
giveRefs qf refs NoRefs          = qf
giveRefs qf refs (FreeRef σ σs) = giveRefs ⟦$qf $(refs DMap. ! σ)⟧ refs σs
```

The function takeRefs takes a binding that requires some free references and introduces a new lambda for each, placing the code of the argument into a map of ΣVars to STRefs. The filled map is then passed to a function that then creates a BaseFunc: the result of takeRefs overall is to create a lambda with as many arguments as free references plus the normal three. The opposite function is giveRefs, which takes the code for a function requiring free references of type xs and iteratively provides each one, by pulling them from the map: this results in the code for a plain base function without any free references. To be able to use the Func type in the rest of the code, the definition of NTBound needs to change to keep track of the free references and keep the types of these references existential, so that it does not interfere with the rest of the system:

```
data NTBound r s a = ∀rs.NTBound (Refs rs) (Code (Func rs r s a))

makeNTBound :: Some Refs → (DMap ΣVar (QSTRef s) → Code (BaseFunc r s a)) → NTBound r s a
makeNTBound (Some refs) f = NTBound refs (takeRefs DMap.empty f refs)

applyNTBound :: NTBound r s a → DMap ΣVar (QSTRef s) → Code (BaseFunc r s a)
applyNTBound (NTBound σ qf) refs = giveRefs qf refs σs
```

The NTBound type now existentially defines rs, which describes the types of references within the binding, and this information is preserved by keeping the Refs rs inside the structure. The makeNTBound and applyNTBound functions wrap around the takeRefs and giveRefs functions to hide the types of the references from the rest of the system. Finally, the eval and evalCall functions need to change to accommodate the new NTBound type.

```
eval :: Machine a → DMap NT Machine → Map Word64 (Some Refs) → Code String → Code (Maybe a)
eval m ms freerefs inp =
  letRec ms show (λ NT w m nts → makeNTBound (freerefs ! w) $ λrefs →
                   ⟦λinp h ret → $(evalMachine m nts refs ⟦inp⟧ ⟦h⟧ ⟦ret⟧)⟧)
                 (λnts → ⟦ runST $(evalMachine m nts DMap.empty inp fail accept)⟧)
  where fail   = ⟦λ_   → return Nothing⟧
        accept = ⟦λx _ → return (Just x)⟧
        evalMachine :: Machine x → Map NT (NTBound r s ) → DMap ΣVar (QSTRef s)
                    → Code String → Code (String → ST s r) → Code (x → String → ST s r)
                    → Code ( ST s r)
        evalMachine m nts refs inp h ret =
          cata alg m (Ctx {nts = nts, φs = DMap.empty, refs = refs })
                     (Γ {input = inp, moore = QNil, handlers = Cons h Nil, retCont = ret})
        ...
```

The eval function has a few notable changes here: firstly, runST is inserted around the top-level parser to run out the ST monad, and both terminal continuations use return to wrap up the result in ST. The evalMachine now takes in the reference map as an argument to be provided to the empty context, and the function for generating bindings provided to letRec now collects the required free references for a specific non-terminal and uses makeNTBound to package them up. Note that this does require a change to the type of letRec, which now needs to be aware of the NTBound type: this is unfortunate, but it is not otherwise possible to avoid unless untyped code is exposed in the type of letRec. The change to letRec itself is very minor, so is omitted.

```
evalCall :: NT x → Eval r s (x : xs) (Succ n) a → Eval r s xs (Succ n) a
evalCall nt k ctx{..} γ{..} = ⟦ $(applyNTBound (nts DMap. ! nt) refs) $input $h $ $ret⟧
  where Cons h _ = handlers
        ret      = ⟦λx input′ → $(k ctx (γ {input = ⟦input′⟧, moore = QCons ⟦x⟧ moore}))⟧
```

The evalCall function is adjusted to make use of applyNTBound, which will immediately provide references to the call sourced from the current in-scope references – these are either free or have been defined along the way. With these changes made, references are fully supported by the system. One improvement that could be made is to allow recursive bindings to close over their free references as to not need to provide them every self-call (adding additional function arguments results in a non-negligible performance slow-down): this is done in practice.

4.5 Turing-Completeness of References

With the API and implementation of references covered, this section focuses on demonstrating the expressive power that references have added to the library. In fact, the addition of references makes `parsley` able to generate Turing-powerful machinery, demonstrated here by giving the implementation of a parser and evaluator for a known Turing-complete language called *Branflakes*[4].

However, there is some nuance to this. `Parsley` is not monadic, and yet it can be shown to be able to interpret a Turing-complete language: it seems contradictory. The key is that the static component of `parsley` cannot implement (»=), or join, because these combinators require parametric dynamic information to produce static information. However, the static component of `parsley` can be used to generate code that can perform monadic parsing: in effect, `parsley` can build interpreters for monadic combinators that can execute during parse-time, but itself cannot express monadic combinators at compile-time. When something is not implementable with `parsley` directly, it is instead possible to push it down to runtime, where it can be implemented.

In a purely dynamic but still deep-embedded variant of `parsley`, the bind combinator introduced in §4.1.2 allows for join to be defined by interpreter, and therefore unlocks monadic power: such an interpreter would be ill-staged in the staged version. Instead, a static join_s would have type:

$$\text{join}_s :: \text{Parser}_s\ (\text{Parser}_d\ a) \rightarrow \text{Parser}_s\ a$$

Where Parser_s is the purely static syntax of `parsley`, and Parser_d is a purely dynamic syntax of `parsley` where all Code has been stripped away. In Parser_d's syntax, however, join_d can be encoded:

$$\text{join}_d :: \text{Parser}_d\ (\text{Parser}_d\ a) \rightarrow \text{Parser}_d\ a$$

While join_s cheats to provide all the power that join_d would otherwise provide, it is not a monadic join because the type is incorrect. Nevertheless, its existence allows parser to engage in any context-sensitive workload.

4.5.1 Static Destructing of Results

With access to the bind combinator and selectives, it is possible to implement a powerful form of pattern matching that is less restrictive that the regular branch combinator. As an example, a list can be de-structured using selectives with the following signature (the implementation is unimportant):

$$\text{listS} :: \text{Selective}\ f \Rightarrow f\ [a] \rightarrow f\ b \rightarrow f\ (a \rightarrow [a] \rightarrow b) \rightarrow f\ b$$

This has a similar shape to branch, which can be called eitherS to make the correspondence clearer. However, the processing of the individual components of the constructors is dynamic, and the structure of the branches is

[4] https://esolangs.org/wiki/Brainfuck

static and cannot refer to the components in multiple places. Consider a monadic version that can break up the components and generate structure entirely dynamically:

```
eitherM :: Monad m ⇒ m (Either a b) → (a → m c) → (b → m c) → m c
eitherM me left right = me »= either left right
```

This has more expressive power, as the left and right branches can depend arbitrarily on the extracted values from the match. However, by restricting this signature slightly, like the restrictions on the signature of bind, a version can be made using references that allows for purely static branch synthesis that can still refer to the components in any position and any number of times:

```
eitherR :: Parser (Either a b) → (Parser a → Parser c) → (Parser b → Parser c) → Parser c
eitherR pe left right = bind pe $ λe → ifS (⟦isLeft⟧ <$> e) (left (⟦fromLeft⟧ <$> e)) (right (⟦fromRight⟧ <$> e))
listR :: Parser [a] → Parser b → (Parser a → Parser [a] → Parser b) → Parser b
listR pxs nil cons = bind pxs $ λxs → ifS (⟦null⟧ <$> xs) nil (cons (⟦head⟧ <$> xs) (⟦tail⟧ <$> xs))
```

Both combinators start by performing a bind on the scrutinee parser to allow for its result to be persisted without re-parsing it. Then they make use of a standard selective conditional branch, where the condition asks if the value uses a specific constructor: this is, in principle, like how bindS works, except that uses plain equality. If each case, the corresponding continuation can be used, using irrefutable deconstructors to pull the data out of the original value[5]. Another example, which will be useful later, is the maybeR combinator – the definition is omitted, as it is equally as formulaic as listR and eitherR.

```
maybeR :: Parser (Maybe a) → (Parser a → Parser b) → Parser b → Parser b
```

This combinator can be used to check for a single constructor individually, most effective when the constructor has a single argument (though a pair can be de-structured irrefutably with a pairR combinator anyway).

4.5.2 A Parser for *Branflakes*

Before an interpreter is given for the language *Branflakes*, it is useful to see what the AST for the language is, and how it relates to the syntax. The operators of the language are represented by the following type:

```
data BranflakesOp = RightPointer | LeftPointer | Increment | Decrement
                  | Output | Input | Loop [BranflakesOp] deriving (Eq, Lift)
```

Here, each of the six non-recursive operators in the language are represented as constructors, with the control flow operation Loop, which corresponds to [·], containing a recursive list of further operations. This type is given Eq and Lift instances to allow it to be used with the matchS combinator later. The parser is as follows:

```
branflakes :: Parser [BranflakesOp]
branflakes = whitespace *> bf
```

[5]If one wished, these could also be persisted using bind to prevent re-evaluation of the destructors for efficiency: deconstructing is pure however, so no harm comes from re-evaluation.

```
  where whitespace = skipMany (noneOf "<>+-[],.$")
        lexeme p = p <* whitespace
        bf = many (lexeme  ( (char '>' $> ⟦RightPointer⟧) <|> (char '<' $> ⟦LeftPointer⟧)
                           <|> (char '+' $> ⟦Increment⟧)    <|> (char '-' $> ⟦Decrement⟧)
                           <|> (char '.' $> ⟦Output⟧)       <|> (char ',' $> ⟦Input⟧)
                           <|> lexeme (char '[') *> (⟦Loop⟧ <$> bf) <* char ']'))
```

The parser is very straightforward, collecting zero or more operations, which are just a straightforward mapping from character to constructor. After each token is consumed "whitespace" is dropped from the input, which is any other character. As a slight deviation from the language, the $ character is significant and denotes the end-of-input: the reason is that the input for the execution of the parsed program will follow the delimiting $.

4.5.3 An Interpreter for *Branflakes*

With the infrastructure in place, it is now possible to build an in-`parsley` interpreter for *Branflakes*. The language is centred around an infinite tape of integers, which are initially all set to zero. This structure is easy to define, and a few operations for working with a tape are provided, though the implementations are not important here:

```
data Tape = Tape [Int] Int [Int]        emptyTape = Tape (repeat 0) 0 (repeat 0)
left   :: Tape → Tape                   readTape  :: Tape → Int
right  :: Tape → Tape                   writeTape :: Int → Tape → Tape
```

During the valuation of the list of operations, each operator in turn will need to be inspected. For this, the listR combinator is needed to deconstruct the list, and then a loopR combinator is needed to check for Loop:

```
loopR :: Parser BranflakesOp → (Parser [BranflakesOp] → Parser a) → Parser a → Parser a
loopR pop = maybeR (getLoop <$> pop)
  where getLoop :: BranflakesOp → Maybe [BranflakesOp]
        getLoop (Loop p) = Just p
        getLoop _        = Nothing
```

This is simply implemented in terms of the maybeR combinator from §4.5.1. To deal with the remaining operators, the matchS combinator can be used. The formulation of matchS used here differs slightly from the definition given by Mokhov et al. [2019]:

```
matchS :: (Eq a, Lift a) ⇒ [a] → Parser a → (a → Parser b) → Parser b → Parser b
```

The first argument is the list of values that can be handled, followed by the parser that produces an element of that list. Then a function is provided which handles the cases for each of the known inspectable values along with a default case used when the value produced by the first parser is not part of the list of elements. The domain of a matchS that does not handle Loop is:

```
nonLoopOps = [RightPointer, LeftPointer, Increment, Decrement, Output, Input]
```

The outer-most part of the interpreter is responsible for fetching the *Branflakes* operations and setting up the initial state:

```
evalBf :: Parser [Char]
evalBf = newRef (branflakes <* char '$') $ λinstrs →
                 newRef_ ⟦emptyTape⟧ $ λtape →
                   newRef_ ⟦[ ]⟧ $ λout →
                     evalBf′ instrs tape out *> reads_ out ⟦reverse⟧
```

The interpreter first parses branflakes and persists the result in the reference instrs, then parses the delimiter $. Then two references are created, one containing the tape, tape, and the other containing the output stream for the program, out. With the state set-up, the interpreter invokes the evalBf′ "combinator" with each of the three references. Finally, the result of the out reference is reversed and returned as the result of the interpreter.

```
evalBf′ :: Ref r₁ [BranflakesOp] → Ref r₂ Tape → Ref r₃ [Char] → Parser ()
evalBf′ instrs tape out = go
  where go = listR (read instrs) unit $ λop remaining →
                 write instrs remaining
             *> loopR op (λops → whileS (readsT ⟦≢ 0⟧) (local instrs ops go))
                         (matchS nonLoopOps op evalOp empty)
             *> go
        evalOp :: BranflakesOp → Parser ()
        evalOp ...
        readsT :: Code (Int → a) → Parser a
        readsT qf = reads_ tape ⟦$qf · readTape⟧
```

The evalBf′ forms the bulk of the interpreter itself: it closes over the three references and operates as a mostly tail-recursive loop (except for recursion in the handling of the Loop construct). Each step, the listR combinator is used to fetch the next op from the instructions; the remaining operations are placed back into instrs for the next iteration. Then, the parser uses loopR to check whether the current operation is a Loop or not. If it is, then while the current tape cell is zero (this is the termination condition for loops in *Branflakes*) it temporarily sets the instructions to be the loop body with local, executes the body, then iterates – the iteration itself is handled by whileS. If the current operation is not a Loop matchS is used to handle the other cases:

```
evalOp RightPointer = modify_ tape ⟦right⟧
evalOp LeftPointer  = modify_ tape ⟦left⟧
evalOp Increment    = writeT (reads ⟦succ⟧)
evalOp Decrement    = writeT (reads ⟦pred⟧)
evalOp Output       = modify out (reads ⟦(:) · chr⟧)
evalOp Input        = writeT (⟦ord⟧ <$> item)
evalOp Loop { }     = error "this is impossible"
```

```
writeT :: Parser Int → Parser ()
writeT px = modify tape ([[writeTape]] <$> px)
```

Each case maps to the corresponding operation on the tape or output stream. In this case, the Loop constructor is not possible since it has been ruled out by loopR. Amusingly, this is the only place where parsing is performed in this parser: the Input case reads any character from the input using the item parser. In all, the interpreter is surprisingly concise: it is similar in structure to an imperative implementation, relying on recursion, `while`-loops, `if`-statements, and mutable variables. This gives a sense of how join_s might be implemented by writing an interpreter for the dynamic `parsley` AST using similar techniques. A working interpreter for a Turing-complete language demonstrates that `parsley` itself can generate Turing-complete parsers, despite not being monadic.

Summary

Restricting `parsley` to only support selective combinators is necessary to allow for ahead-of-time compilation of the DSL into pure Haskell code. While this limits the expressive power of the library to not be capable of parsing all context-sensitive grammars, the reference machinery introduced in this chapter helps recover that.

Implementing pieces of mutable state in a non-monadic way provides alternative representations for the classic State laws, and combinators like forR provide an intuitive API for some of the more classic context-sensitive workloads. They are sufficient to implement combinators for destructuring data while allowing their sub-components to be re-used multiple times, allowing for more complex parser descriptions.

To make references work in the context of the global binding hoisting performed by requires implementing a lambda-lifting algorithm to add the closed references into the bindings. This changes the type of the bindings so they can have an existentially bound selection of references: in future, this can be taken much future to allow for fine-grained configuration of all sorts of bindings within the backend.

Chapter 5

Optimising for Semi-Contiguous Data

The following chapter has been adapted from an unpublished draft in collaboration with Nicolas Wu.

The implementation of a staged parser combinator library as seen in CHAPTER 3: EMBEDDING A PARSER COMBINATOR LIBRARY gives an opportunity for domain-specific optimisations and analysis that are not feasible to perform during the runtime of a parser – which would be necessary in an unstaged library – due to performance implications. This is the focus of CHAPTER 6: ANALYSIS AND OPTIMISATION, which describes the various optimisations and analyses employed by `parsley` to help improve the generated code. However, this, in many cases, involves the static inspection of character predicates; as it is not possible to arbitrarily inspect static functions in the metaprogramming framework, these Char ⟶ Bool functions need to be converted – effectively via a form of "The Trick" (§2.4.1) – to a data structure that is inspectable.

Motivation A potential candidate for such a data structure would be the humble Set Char, which can be used to check a candidate character is a member of the set. However, this can be very large; as an example, using the item combinator, which uses the predicate const True, would correspond to a massive $\approx 2^{20}$-element set requiring around 88MiB of memory to accommodate: this would also require worst case 20 comparisons to query, as opposed to the zero comparisons of const True.

Predicate Shapes The core set of combinators for character consumption are as follows:

```
satisfy :: (Char ⟶ Bool) ⟶ Parser Char

item :: Parser Char
item = satisfy (const True)

char :: Char ⟶ Parser Char
char c = satisfy (≡ c)

oneOf :: Set Char ⟶ Parser Char
oneOf cs = satisfy (∈ cs)

noneOf :: Set Char ⟶ Parser Char
noneOf cs = satisfy (∉ cs)
```

This puts the most "regular" shapes of predicates as matching all characters, a single character, or a specific set of characters. In the last case, the shape of those sets is usually a collection of characters that are often sequential in nature – `A-Z`, or `0-9`, say – or a mix of such sequential sets. Of course, matching all characters is the same as matching within the sequence to `\0-\x10FFFF`, and a negative inclusion on *cs* is the same as checking inclusion in the set `[\0-\x10FFFF]` \ *cs*. It is irregular for the predicates to be pathologically sparse, with a random distribution of characters in the set: usually they are highly structured.

Range Sets This chapter concerns the implementation of a generic data structure that leverages the semi-contiguity of the average parsing predicate to vastly reduce its space requirements and optimise the query time relative to Set. This is known as a *range set*, which stores contiguous ranges in the nodes as opposed to single values. With this structure, the const True predicate only requires around 33 bytes of memory with only two comparisons to check membership: a substantial improvement over Set.

A high-level comparison of RangeSet and Set is provided in §5.1, with the implementation of RangeSet given in §5.2. Optimisations that can be employed to further improve the structure are discussed in §5.3 and benchmarks evidencing its effectiveness are provided in §5.4.

5.1 Comparing Different Sets

A set is a data structure that supports membership queries as well as efficient insertion, union, intersection, and difference, whilst ensuring no duplicates in the structure. The performance characteristics of these operations may vary between different implementations of sets, but they are assumed to perform well for at least some of their operations. They are used in a variety of applications across computer science, both practical and theoretical.

A simple implementation of a set might be a function of type Eq a ⇒ a → Bool, which supports many operations in constant time, but membership queries degrade to linear time in the number of additive and subtractive operations used to build it. Lists can also act as sets, assuming duplicates are removed, but this suffers from poor performance characteristics across all the operations of either $O(n)$ or $O(n^2)$. Commonly, one might expect that sets support their operations in $O(\log n)$ or $O(n \log n)$ time. A common implementation of a set that does provide these guarantees – or better – are self-balancing binary trees: this requires a tighter bound on the elements of the set, namely Ord a.

`Data.Set` Haskell supports a sized-balanced binary tree implementation of a Set in the `containers`[1] package [Nievergelt and Reingold 1972; Adams 1993; Adams 1992]. It supports membership, insertion, and deletion in $O(\log n)$, as well as union, intersection, and difference in $O(m \log(\frac{n}{m} + 1))$ with m and n as the sizes of the two sets such that $m \leqslant n$ [Blelloch, Ferizovic, and Sun 2016]. It is described as follows:

```
data Set a = Bin Int a (Set a) (Set a) | Tip
```

Given some node Bin sz x lt rt, it is guaranteed that all the elements in the left sub-tree lt will be strictly smaller than x and all the elements in the right sub-tree rt will be strictly larger than x. The size sz of the node will be size lt + size rt + 1. The idea is that the size is used to balance the structure, ensuring that the tree does not adopt a pathological shape resulting in $O(n)$ operations – this contrasts with AVL trees, which use height [Adelson-Velskii and Landis 1962].

The size of such trees is limited to $2^{63} - 1$ elements and sets that are any larger are undefined behaviour: this is not a problem in practice, as the space requirements of such a tree would be just over 700 Exbibytes! Indeed, this means the set-theoretic *complement* operation is impractical to implement for sets containing elements of even modestly sized types.

[1] https://hackage.haskell.org/package/containers

Operation	Complexity `Data.Set`	Complexity `Data.RangeSet`	Complexity `Data.IntSet`
`member`	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(u)$
`insert`	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(u)$
`delete`	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(u)$
`union`	$\Theta(m \log(\frac{n}{m}+1)), m \leqslant n$	$\Theta(m \log(\frac{n}{m}+1)), m \leqslant n$	$\Theta(u+v)$
`intersect`	$\Theta(m \log(\frac{n}{m}+1)), m \leqslant n$	$\Theta(m \log(\frac{n}{m}+1)), m \leqslant n$	$\Theta(u+v)$
`difference`	$\Theta(m \log(\frac{n}{m}+1)), m \leqslant n$	$\Theta(m \log(\frac{n}{m}+1)), m \leqslant n$	$\Theta(u+v)$
`size`	$\Theta(1)$	$\Theta(n)$	$\Theta(n)$
`complement`	not supported	$\Theta(n)$	not supported
`fromList`	$\Omega(n), O(n \log n)$	$\Omega(n), O(n \log n)$	$\Theta(nu)$

Table 5.1: Complexities of various Set and RangeSet operations.

In all, `Data.Set` is a suitable data structure for a wide variety of general tasks involving set operations. It works with a common requirement of Ord on its elements and performs well in practice.

`Data.IntSet` In addition to `Data.Set`, `containers` also includes a specialised set for integers, based on Patricia trees [Morrison 1968; Okasaki and Gill 2000], which is related to tries: a data-structure for storing a set or map of list-like keys, such as strings. Instead of performing multiple (potentially expensive) comparisons against the key, trie-like structures branch at each individual element of the key, allowing a lookup to be done in $O(n)$ time in the length of the key, not the size of the trie. This makes them especially useful for modelling sets of Strings, especially with a smaller fixed alphabet of characters. A Patricia tree supports non-list-like keys by branching on the binary representation of a key instead. This means that Patricia trees are also binary trees, as each branch only needs to support an alphabet of 0 and 1. A good Patricia tree (or trie) implementation will also compress shared prefixes within the structure, to avoid long chains of redundant branches, improving the memory requirements of the structure.

`Data.IntSet` supports linear time for most of its operations in the number of bits within the query. It also has a much more compact representation than `Data.Set`, with the set of all characters occupying just over 1MiB: this is a significant improvement over `Data.Set`, but several orders of magnitude more than is offered by range sets.

`Data.RangeSet` This chapter offers the definition of an alternative set implementation called RangeSet, which not only supports a cheap complement operation, but also refines its structure to be more optimised for semi-contiguous data. To achieve this, it imposes a tighter Enum bound on its elements and uses this to improve the density of the set, reducing both its memory consumption and time for deeper traversals of the structure as the data fed into it becomes more contiguous.

The RangeSet's Enum constraint allows the set's elements to be checked for whether they are *adjacent*[2] values: `x` and `y` are adjacent if succ `x` ≡ `y` ∨ `x` ≡ succ `y`. The core idea is to leverage this to bundle together sequences of adjacent members of the set into a single, contiguous, range within a node. There is an invariant that for any two elements x and y of a range-set X, such that $x \leqslant y$, if x and y are adjacent they must form part of the same range $l \ldots u$, such that $l \leqslant x$ and $y \leqslant u$, with all other elements between l and u also being members of the set X.

Since this structure maximises the number of elements that can be stored in the same node without requiring

[2]In practice, working with finite types means that the definition of *adjacent* needs to be more carefully defined (§5.2.1).

additional memory, it can also naturally implement a complement operation for Bounded types. For a range-set with n nodes, its complement will have between $n - 1$ and $n + 1$ nodes, and this is cheap to compute.

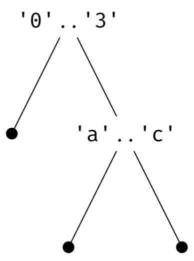
The number of elements in a RangeSet can grow to be significantly larger than those found within Set without impractical memory constraints, however, due to the use of Int in the definition of Haskell's Enum class, these sets only work on elements x such that toEnum x is less than $2^{63} - 1$ (specifically for GHC). It is easy to formulate a new version of Enum, which works with Integer, which would permit arbitrarily sized RangeSets, but Int is more than sufficient to handle the use-case with Char and will be more efficient.

5.2 Implementation

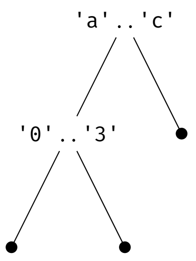
A RangeSet is a set-like structure that imposes Ord and Enum constraints on its elements based on AVL trees [Adelson-Velskii and Landis 1962], which are self-balancing binary trees. The type of the structure is described by the following datatype:

```
data RangeSet a = Fork Word8 a a (RangeSet a) (RangeSet a) | Tip
```

A Tip represents the empty tree, and a Fork h l u lt rt is a tree with height h, left- and right-children lt and rt, as well as a range of elements in the node l through u inclusive. A singleton node is one where l ≡ u. For example, the set fromList ['0','1','2','3','a','b','c'] may be represented by either of the trees in fig. 5.1.



(a) Fork 2 '0' '3' Tip (Fork 1 'a' 'c' Tip Tip)



(b) Fork 2 'a' 'c' (Fork 1 '0' '3' Tip Tip) Tip

Fig. 5.1: Possible trees for fromList ['0','1','2','3','a','b','c'].

Here, the outer Fork of fig. 5.1a has height 2, because it has one child of height 0 and one of height 1; the inner Fork has height 1 as both children are Tips. The structure is similar for the tree in fig. 5.1b; the specific value generated depends on how the fromList function has been defined. There are no other *valid* values for this set.

The height has been chosen to be of type Word8, which is an unsigned 8-bit integer: this is ok, since, as discussed, there can be at most $2^{63} - 1$ values within the set. In a balanced binary tree, the height h of a tree is proportion to $\log n$, with n the size of set: this means that the height is upper bounded by 64, fitting comfortably in the bounds of the 8-bit unsigned integer. A smaller type here will result in lower memory usage for the structure, assuming GHC represents Word8 as a single byte.

5.2.1 Valid RangeSets

Like other kinds of sets, RangeSet imposes some restrictions on the shapes of the values that inhabit it. A RangeSet is *valid* only when the following invariances are observed:

- **Balanced**: For any fork in the tree Fork h _ _ lt rt, its height h must be equal to max (height lt) (height rt) + 1 and the heights of lt and rt can differ by at most 1.
- **Ordered**: For any fork in the tree Fork _ l u lt rt, all elements in lt must be strictly less than l and all elements in rt must be strictly greater than u. Furthermore, l ⩽ u.
- **Disjoint**: For any fork in the tree Fork _ l u lt rt, the greatest element in lt, should one exist, must not be *adjacent* to l; and the least element in rt, should one exist, must not be *adjacent* to u.

As mentioned earlier, two values x and y are *adjacent* when any of the following four equations are satisfied:

$$x \equiv \text{succ } y \quad (5.1) \qquad \text{succ } x \equiv y \quad (5.2) \qquad \text{pred } x \equiv y \quad (5.3) \qquad x \equiv \text{pred } y \quad (5.4)$$

There is redundancy in this set of equations, as several pairs of these equations suffice to cover all cases of adjacency, but it is important to note all of them because RangeSet supports finite types. For finite (Bounded) types, it is not safe to use succ maxBound or pred minBound: care must be taken to ensure only the safe adjacency equations are tested in any specific context.

The first half of the **Balanced** invariance is preserved using the smart-constructor fork:

```
height :: RangeSet a → Word8
height Tip              = 0
height (Fork h _ _ _ _) = h

fork :: a → a → RangeSet a → RangeSet a → RangeSet a
fork l u lt rt = Fork (max (height lt) (height rt) + 1) l u lt rt
```

The fork constructor can be used to construct values where the height is guaranteed to be correct. There are places, however, where shortcuts can be made with contextual knowledge and Fork can be used with a known height. This will be done (and justified) when relevant.

To ensure that trees are correctly balanced, a more powerful smart constructor, balance is used in cases where it is possible that the trees have become imbalanced:

```
balance l u lt rt | height lt > height rt + 1 = uncheckedBalanceL l u lt rt
                  | height rt > height lt + 1 = uncheckedBalanceR l u lt rt
                  | otherwise                 = fork l u lt rt
```

This smart constructor checks to see if the heights of the two sub-trees are sufficiently imbalanced. If they are, it will use the biased balancing smart constructors uncheckedBalanceL or uncheckedBalanceR:

```
uncheckedBalanceL l1 u1 lt@(Fork _ l2 u2 llt lrt) rt
  | height llt ⩾ height lrt = rotr l1 u1 lt rt
  | otherwise               = rotr l1 u1 (rotl l2 u2 llt lrt) rt
```

The uncheckedBalancedL smart constructor assumes that the left tree has become larger than the right tree. The imbalanced trees are rotated in such a way that the balance is restored. This is done with rotr and rotl:

```
rotl l1 u1 p (Fork _ l2 u2 q r) = fork l2 u2 (fork l1 u1 p q) r
```

```
rotr l1 u1 (Fork _ l2 u2 p q) r = fork l2 u2 p (fork l1 u1 q r)
```

The smart constructor uncheckedBalancedR is symmetrical to uncheckedBalancedL and is omitted for brevity. It is important that these unchecked smart constructors are not used outside of the balance function: for cases where the left- or right-bias is known, balanceL and balanceR functions can also be defined that perform the necessary checks to ensure that the tree is already unbalanced.

The other two invariants, **Ordered** and **Disjoint**, need to be considered more carefully when implementing each of the set operations. However, it is safe to assume they hold true for any provided RangeSet value.

5.2.2 Testing for Membership

The simplest operation to implement on a range-set is the membership function. This is like the classic member function on binary search trees, however inclusion within the range must be tested:

```
member :: Ord a ⇒ a → RangeSet a → Bool
member x = go
  where go Tip = False
        go (Fork _ l u lt rt) | l ⩽ x = x ⩽ u ∨ go rt
                              | l > x  = go lt
```

The member function relies on the **Ordered** property of range-sets: if the element x is greater than a fork's lower-bound l, then it will not be found in the left sub-tree, and if it is not less than the upper-bound u, it will be found in the right sub-tree. In this definition, the redundant condition l > x is left in for clarity: a good implementation will minimise the number of comparisons that need to be performed. Because the set is guaranteed to be **Balanced**, membership is worst-case $O(\log m)$, where $m \leqslant n$ is the number of *ranges* in the set and n is the number of elements of the set.

5.2.3 Adding and Removing Elements

Defining insertion and deletion on a range-set is more involved than the equivalent functions on AVL trees. Unlike AVL insertion and deletion, inserting into, or removing from, a range-set requires careful consideration of more cases, to preserve the **Balanced**, **Ordered**, and **Disjoint** properties.

Insertion To implement insertion on a range-set, six cases must be considered: along with the usual empty case, already included case, and inserting into the right- or left-half otherwise cases; range-set implementations additionally need to consider whether an element to insert *fuses* with an existing node. This is made trickier by the requirement that all ranges within the set must be **Disjoint**: recall that two ranges cannot exist whose extremes are *adjacent*. The inclusion of an element that increases the size of an existing range may result in that range becoming adjacent to a range deeper in the tree: this cannot be allowed to happen and both nodes must be fused together. A quirk of this is that insertion into a range-set may not only expand the tree: it might shrink it instead! With these considerations in mind, the insert function is defined as follows:

```
insert :: (Enum a, Ord a) ⇒ a → RangeSet a → RangeSet a
insert x Tip                                  = Fork 1 x x Tip Tip
insert x t@(Fork h l u lt rt) | l ⩽ x, x ⩽ u     = t
                              | x < l,  x ≡ pred l  = fuseLeft h x u lt rt
                              | x < l,  x ≢ pred l  = balance l u (insert x lt) rt
                              | x > u, x ≡ succ u = fuseRight h l x lt rt
                              | x > u, x ≢ succ u = balance l u lt (insert x rt)
```

In the empty case, insert forms a singleton node; otherwise, if x is already included within a fork's lower- and upper-bounds, then the tree is left unchanged. Otherwise, the value should either be inserted on the left or the right. When the to-be-inserted value is adjacent to one of the bounds, the corresponding fuse function is called. Otherwise, it is inserted into the correct half of the tree: notice that the balance function used cannot assume anything about which part of the tree grew, since insertion may either grow or shrink the sub-tree. The definition of fuseLeft is as follows:

```
fuseLeft h x u Tip rt = Fork h x u Tip rt
fuseLeft h x u lt@(Fork _ ll lu llt lrt) rt
    | x ≡ succ x′ = balanceR l u lt′ rt
    | otherwise  = Fork h x u lt rt
  where (l, x′, lt′) = maxDelete ll lu llt lrt
```

If the left tree is empty, then no ranges need to be combined, and the range the new range of the node is x to u, with an unchanged height. Otherwise, delete the largest range from the left-tree: if the upper-bound x′ of this range is adjacent to x then fuse the nodes and balance with a right-bias (fusion necessarily shrank the left-tree). Otherwise, no fusion occurs, and the range is expanded with x as the new lower-bound, again with an unchanged height. The function maxDelete will delete the right-most node in a non-empty tree: to guarantee this node exists, the function unpacks its argument as opposed to taking in a RangeSet a as an argument: this is a trick that is used to cut down on unnecessary pattern matches, since fuseLeft already guaranteed the left-tree is non-empty. The definition of maxDelete is as follows:

```
maxDelete l u lt Tip = (l, u, lt)
maxDelete l u lt (Fork _ rl ru rlt rrt) =
  let (ml, mu, rt′) = maxDelete rl ru rlt rrt in (ml, mu, balanceL l u lt rt′)
```

The unpacking of the argument ensures that there is always a valid range and a new tree to return. As maxDelete always removes a range from the right, the tree is rebalanced with a left bias. The fuseRight function is symmetric to fuseLeft and makes use of a symmetric minDelete function. The complexity of insert, fuseLeft, and maxDelete are $O(\log m)$, with m the number of ranges in the tree.

Deletion The delete function needs to consider seven cases. This contrasts with the four cases required for deletion in an AVL tree. The extra three cases, like insert, arise from how elements are removed from within, or at the extremes of, a range. Removing an element from inside an existing range causes the node to undergo *fission*: by removing an element, the two remaining halves of the range are now **Disjoint**, and so will be split up and incorporated back into the tree. Like insert this means that deletion can result in either the tree shrinking, growing, or not changing in shape. The delete function is defined as follows:

```
delete :: (Enum a, Ord a) ⇒ a → RangeSet a → RangeSet a
delete _ Tip                          = Tip
delete x (Fork h l u lt rt) | l ≡ x, x ≡ u = glue lt rt
                            | l ≡ x, x < u = Fork h (succ l) u lt rt
                            | l < x, x ≡ u = Fork h l (pred u) lt rt
                            | l < x, x < u = fission l x u lt rt
                            | x > u        = balance l u lt (delete x rt)
                            | l > x        = balance l u (delete x lt) rt
```

When an entire range has been deleted (when l ≡ x ∧ x ≡ u), the two subtrees are joined back together by the glue function: this function is important for implementing the merging of two trees to implement intersection and difference. In the case where the removed element exists at the extremes of a range, the tree can be simply reconstructed with an unchanged height and a smaller range: the range in question is disjoint from its children anyway, so no further consideration is needed. However, when the removed element is within a range, that range must be broken using fission. The other three cases are standard, but the balancing cannot be biased, as fission can grow the tree and removing a singleton range shrinks it.

```
fission l1 x u2 lt rt
   | height lt > height rt = fork l1 u1 lt (unsafeInsertL l2 u2 rt)
   | otherwise             = fork l1 u1 (unsafeInsertR l2 u2 lt) rt
  where u1 = pred x
        l2 = succ x
```

The fission function is responsible for breaking a range apart into two fork nodes. To reduce the amount of rebalancing of the tree required, the new range is always inserted into the smaller of the two child trees when possible. The functions unsafeInsertL and the symmetric unsafeInsertR are used to insert this newly formed range into the tree at its extremities:

```
unsafeInsertL :: Enum t ⇒ t → t → RangeSet t → RangeSet t
unsafeInsertL l u Tip                  = Fork 1 l u Tip Tip
unsafeInsertL l u (Fork _ l′ u′ lt rt) = balanceL l′ u′ (unsafeInsertL l u lt) rt
```

When the original range is broken apart, all its elements will be less than any element found in the right-tree (or greater than any elements found in the left-tree): this is due to the **Sorted** invariant. As such, the new range to insert must be placed into the left-most position of the right-tree: this can be achieved by recursing down to the left-most Tip and replacing it with the new node. This function is referred to as unsafe because it is only legal to use when the range to be inserted is guaranteed not to overlap (or be adjacent to) any existing range in the tree. In this case, the range that is inserted has originated from the parent node, so due to the **Disjoint** invariant, it is safe to use unsafeInsertL as this range could not have occurred in the child.

```
glue Tip rt = rt
glue lt Tip = lt
glue lt@(Fork lh ll lu llt lrt) rt@(Fork rh rl ru rlt rrt)
   | lh < rh      = let (l, u, rt′) = minDelete rl ru rlt rrt in fork l u lt rt′
   | otherwise = let (l, u, lt′) = maxDelete ll lu llt lrt   in fork l u lt′ rt
```

As mentioned above, the glue function takes two disjoint trees, balanced with respect to each other, and connects them together by using an extremity range from the larger of the two trees. At worst, the two child trees were of equal height, so their height will differ by at most one post-deletion of a range: no balancing is required.

The functions unsafeInsertL and unsafeInsertR do work proportional to the height of the tree, so fission has complexity $O(\log m)$. Similarly, glue is $O(\log m)$ as it is built only upon minDelete and maxDelete. As such, delete itself only does work proportional in the height of the balanced tree, meaning it to is $O(\log m)$.

5.2.4 Composite Operations

In Adams [1992], the composite set operations of union, intersection, and difference are described in terms of divide and conquer algorithms, the complexity bounds on which proven by Blelloch, Ferizovic, and Sun [2016]. These algorithms can also be adapted to work with the range-set but with some added complexity because of the interaction with ranges themselves to ensure the relevant invariants are preserved. Though not formally proved, the adapted algorithms retain the same complexities as proven by Blelloch, Ferizovic, and Sun [2016], but where the complexity is related to the number of ranges in the set instead of its actual size (see table 5.1). A benchmark-driven quantified justification of these bounds is offered in §5.4.2.

Union The idea behind union is to take the top-most element (or in this case range) from one of the sets and use it to partition the second set. Then, the two halves of each set are recursively unioned together and the resulting disjoint trees are merged back together rooted at the original pivot.

```
union t Tip = t
union Tip t = t
union t$_1$@(Fork h$_1$ l$_1$ u$_1$ lt$_1$ rt$_1$) t$_2$@(Fork h$_2$ l$_2$ u$_2$ lt$_2$ rt$_2$)
   | h$_1$ < h$_2$      = let (lt$'_1$, rt$'_1$) = split l$_2$ u$_2$ t$_1$ in link l$_2$ u$_2$ (union lt$'_1$ lt$_2$) (union rt$'_1$ rt$_2$)
   | otherwise = let (lt$'_2$, rt$'_2$) = split l$_1$ u$_1$ t$_2$ in link l$_1$ u$_1$ (union lt$_1$ lt$'_2$) (union rt$_1$ rt$'_2$)
```

The union function is defined in terms of two auxiliary helpers, which are both used for other composite functions as well: these are split and link. Since split does more work than plain de-structuring, it is beneficial to always split on the shorter of the two argument trees: this is fine, since union is commutative.

```
link l u Tip                    Tip                    = Fork 1 l u Tip Tip
link l u Tip                    (Fork _ rl ru rlt rrt) = insertLAdj l u rl ru rlt rrt
link l u (Fork _ ll lu llt lrt) Tip                    = insertRAdj l u ll lu llt lrt
link l u lt@(Fork _ ll lu llt lrt) rt@(Fork _ rl ru rlt rrt) = disjointLink l′ u′ lt′′ rt′′
  where
```

```
    (lmaxl, lmaxu, lt′) = maxDelete ll lu llt lrt
    (rminl, rminu, rt′) = minDelete rl ru rlt rrt
    (l′, lt″)  | lmaxu ≡ pred l = (lmaxl, lt′)
               | otherwise      = (l, lt)
    (u′, rt″)  | rminl ≡ succ u = (rminu, rt′)
               | otherwise      = (u, rt)
```

The link l u lt rt function is responsible for combining two non-overlapping trees lt and rt with a new range l to u between them. This does not assume that the new range to insert is not disjoint from either of the two sub-trees, so it must check to see if the ranges could fuse. When any fusion has been performed with the extremity ranges of the two trees, then disjointLink can be used, which links two trees assuming they are not adjacent to the intermediate range. When either of the two trees is a Tip then, since the range cannot appear inside the other tree, it can be inserted directly into an extreme position (this may also be subject to fusion).

```
disjointLink l u Tip rt   = unsafeInsertL l u rt
disjointLink l u lt   Tip = unsafeInsertR l u lt
disjointLink l u lt@(Fork lh ll lu llt lrt) rt@(Fork rh rl ru rlt rrt)
   | lh < rh + 1 = balanceL rl ru (disjointLink l u lt rlt) rrt
   | rh < lh + 1 = balanceR ll lu llt (disjointLink l u lrt rt)
   | otherwise  = fork l u lt rt
```

When the two trees to link are known to be disjoint from the range that will link them, disjointLink will either insert the range directly into one of the trees when the other is empty or will insert it into the taller of the two trees (pushing it towards an extremity). In practice, this means that linking two trees does not mean the intermediate range will be at the root of the tree: the root of one of the subtrees may be root the new tree instead. If the two trees are balanced with respect to each other, however, then they can be joined with a Fork.

```
insertLAdj l u tl tu tlt trt = **case** minRange tl tu tlt **of**
   (l′, _) | l′ ≡ succ u → fuseL l tl tu tlt trt
   (l′, _) | otherwise  → balanceL tl tu (unsafeInsertL l u tlt) trt
   where
      fuseL l′ l u Tip                   rt = fork l′ u Tip rt
      fuseL l′ l u (Fork _ ll lu llt lrt) rt = fork l u (fuseL l′ ll lu llt lrt) rt

minRange :: a → a → RangeSet a → (a, a)
minRange l  u Tip                = (l, u)
minRange _ _ (Fork _ l u lt _) = minRange l u lt
```

While disjointLink can use unsafeInsertL and unsafeInsertR to inject the new range into a tree, link itself must account for fusion even when one tree is empty. This is the role of insertLAdj (and the symmetric insertRAdj): first extract the minimum range of the non-empty tree (this is found in the left-subtree), if it is the case that this range is adjacent to the range to insert, then fuseL re-traverses the tree to expand the minimum range. Otherwise, a regular unsafeInsertL can be performed to inject the disjoint range into the tree.

The above functions have all been responsible for combining the unioned subtrees back together, the split function, however, is used to break the trees apart for the divide-and-conquer algorithm. This function should break a tree t into two trees ltt and gtt with respect to some range l to u: the tree ltt should consist of all the elements of t less than l, and the tree gtt should consist of all the elements greater than u.

```
split _ _ Tip = (Tip, Tip)
split l u (Fork _ l′ u′ lt rt)
    | u < l′      = let (llt, lgt) = split l u lt  in (llt, link l′ u′ lgt rt)
    | u′ < l      = let (rlt, rgt) = split l u rt in (link l′ u′ lt rlt, rgt)
    | otherwise = (ltt, gtt)
  where ltt  = if | l′ ≡ l   → lt
                  | l′ < l   → unsafeInsertR l′ (pred l) lt
                  | l′ > l   → allLess l lt
        gtt = if | u ≡ u′ → rt
                  | u < u′ → unsafeInsertL (succ u) u′ rt
                  | u > u′ → allMore u rt
```

An empty tree is split by returning two empty trees. Otherwise, given a Fork, it is necessary to compare the ranges to establish if there is any overlap between the current tree's range l′ to u′ and the splitting range l to u. If the ranges are completely disjoint (either u < l′ or u′ < l) then a recursive split is performed, and the range l′ to u′ is combined with the recursive subset and the other subtree using link. If the ranges do overlap, however, there are three cases to consider (symmetric for each side): if the left of the splitting range is equal to the left of the tree's range, then the entire left-subtree is returned; if the splitting range is wider than the tree's range, all the elements less than the splitting range's lower bound in the left-subtree are returned; if the splitting range does not contain all of the elements of the tree's range, then those elements are inserted into the left-subtree.

```
allLess _ Tip = Tip
allLess x (Fork _ l u lt rt)
    | x ≡ l          = lt
    | x < l          = allLess x lt
    | x > l, x ⩽ u = unsafeInsertR l (pred x) (allLess x lt)
    | x > u          = link l u lt (allLess x rt)
```

When collecting all the elements in a tree less than the lower-bound of the splitting range, a cheaper function allLess can be used, which is a trimmed down version of split itself, only considering the relevant cases. In fact, split l u t ≡ (allLess l t, allMore u t), though this is less efficient.

Difference The idea behind difference t dt is like union: it takes the top-most range from the tree to remove, dt, and splits the other tree t by this range, finding all elements in t that are not in this range. Divide-and-conquer, is used to take the difference of the left-tree of the split ltt and the left-subtree of dt and the difference of the right-tree of the split gtt and the right-subtree of dt. Finally, these new trees are combined to form a new tree.

```
difference Tip _                 = Tip
difference t Tip                 = t
difference t (Fork _ dl du dlt drt) =
  let (ltt, gtt) = split dl du t in disjointMerge (difference ltt dlt) (difference gtt drt)
```

Compared to union, difference requires much less new machinery to implement, reusing functionality from union, as well as from delete. For the base cases: removing anything from the empty tree always results in the empty tree, and removing the empty tree from a tree does nothing. In the recursive case, the recursively differenced subtrees are combined with a new function, disjointMerge: this merges two trees, which may not be balanced with respect to each other, but are known to be disjoint. Here this is safe to use since ltt and gtt are guaranteed to be disjoint, as they are separated by the non-empty range dl to du.

```
disjointMerge Tip                    rt  = rt
disjointMerge lt                     Tip = lt
disjointMerge lt@(Fork hl ll lu llt lrt) rt@(Fork hr rl ru rlt rrt)
   | hl < hr + 1 = balanceL rl ru (disjointMerge lt rlt) rrt
   | hr < hl + 1 = balanceR ll lu llt (disjointMerge lrt rt)
   | otherwise  = glue lt rt
```

The function disjointMerge is an extension of the glue function from §5.2.3: this time, the subtrees to be merged may be unbalanced with respect to each other. In these cases, the taller tree is broken apart and half of it is merged with the shorter tree and rooted by its root range.

Intersection In a similar vein to both union and difference, intersection is also defined as a divide-and-conquer algorithm. The high-level idea is, again, to split one of the trees using the top-most range of the other, and recursively apply intersection. In this case, combining the two recursive results needs to also involve any overlap between the chosen range and the elements of the other tree.

```
intersection = pickShortest
  where
    pickShortest Tip _ = Tip
    pickShortest _ Tip = Tip
    pickShortest t1@(Fork h1 _ _ _ _) t2@(Fork h2 _ _ _ _)
      | h1 < h2 = intersect t2 t1
      | otherwise = intersect t1 t2
    intersect (Fork _ l1 u1 lt1 rt1) t2 =
      case overlap l1 u1 t2 of
        Tip              → disjointMerge lt′ rt′
        Fork _ x y olt ort → disjointLink x y (disjointMerge lt′ olt) (disjointMerge ort rt′)
      where (lt2, rt2) = split l1 u1 t2
            lt′ = pickShortest lt1 lt2
            rt′ = pickShortest rt1 rt2
```

If there is no overlap between the chosen splitting range and the other tree (i.e., the overlap is a Tip), then the two recursive subtrees are merged: this can be done with disjointMerge as they are at least separated by the splitting range. If there is an overlap between the splitting range and the other tree, then the top-most range of this overlap x to y can be used to root a new tree, using disjointLink: this is safe because all the elements in the left-recursive case are known to be less than x (and similarly for the right case and y), since x and y are within the bounds of l1 and u1, and, by the disjointness of the original trees, they must not touch any child ranges. By a similar argument, all the elements of the remaining overlap tree children lt′ and rt′ are within the range l1 to u1 and are similarly disjoint from the recursive subtrees: this means disjointMerge can be used to combine them. Like union, this implementation of intersection carefully picks the shortest tree to do the more expensive split and overlap operations on, making the operation symmetric: the intersect helper can, and should, be made total by unrolling the fork parameters from the pickShortest call, but this is left out for clarity.

```
overlap _ _ Tip = Tip
overlap x y (Fork _ l u lt rt)
    | x < l, u < y          = disjointLink l u lt′ rt′
    | l ⩽ x, y ⩽ u          = Fork 1 x y Tip Tip
    | x < l, l ⩽ y, y ⩽ u   = unsafeInsertR l y lt′
    | l ⩽ x, x ⩽ u, u < y   = unsafeInsertL x u rt′
    | y < l                 = lt′
    | u < x                 = rt′
  where lt′ = overlap x y lt
        rt′ = overlap x y rt
```

The overlap x y function is responsible for filtering a tree such that all its elements are within the range x to y. To combine the recursive cases back together, there are six cases:

- if the range x to y completely covers the top-most range of the tree l to u, the recursive cases are combined using disjointLink l u: this is safe, since the recursive subtrees are no bigger than the original subtrees, and they must have been disjoint.
- if the range x to y is completely within the top-most range of the tree, then there is no overlap on either side, and a singleton tree is returned from x to y.
- if one of x or y is within l to u but the other is outside it, the corresponding recursive child is returned with the overlapping segment of range inserted into it: again, this is safe as the range could not have appeared in the child and therefore not in the recursive subtree.
- if the entire range x to y is outside of the top-most range of the tree, then only the corresponding recursive subtree is returned.

It is worth noting that this function could be optimised by recognising that, unless the overlap range x to y is entirely outside of the top-most range of the tree, it is only necessary to find all elements above or below x or y, depending on which side of the tree is being searched. This is because all the elements in the left-subtree are all guaranteed to be less than y and all elements in the right-subtree are all guaranteed to be greater than x.

5.2.5 Complement

Set complement is an operation that can be naturally supported by RangeSet when it is storing elements of a Bounded type. This operation can be done in $O(m)$ time with m the number of ranges in the set and changes the size of the set by at most one node (either smaller or bigger).

The complement of a set t contains all the values that are not elements of t, and none of the elements found inside t. This can be done with minimal modifications to the shape of the tree by shifting all the ranges in the set along by one, where the successor of the upper bound of one range becomes the lower bound of the next range.

```
complement :: (Bounded a, Enum a, Eq a) ⇒ RangeSet a → RangeSet a
complement Tip                                                  = Fork 1 minBound maxBound Tip Tip
complement (Fork _ l u Tip Tip) | l ≡ minBound, u ≡ maxBound = Tip
complement t@(Fork _ l u lt rt) = case maxl of
  Nothing → t′
  Just maxl → unsafeInsertR maxl maxBound t′
  where
    (minl, minu, rest) = minDelete l u lt rt
    (t′, maxl) = if minl ≡ minBound then push (succ minu) rest else push minBound t
```

The first two cases deal with empty and full sets, respectively: in these cases, the complement is easy to construct, as it should be the opposite. By handling the full case up-front, the logic for the remaining partially full case is simplified as it can be safely assumed that the largest element of the minimum range is not maxBound.

In the third case, the minimum range is extracted from the set: if the smallest element is minBound then this range is removed, and the upper bound for that range is *pushed* into the rest of the tree. Otherwise, the tree is left unchanged and the minBound is *pushed* into the original tree. Pushing threads a value into a tree, nudging along all values in the process: it returns the element one larger than the largest found in the tree, maxl, should it exist. If maxl does not exist, that means that the largest element in the tree was maxBound, and nothing is done, otherwise, a range is inserted into the right-most position in the tree from maxl to maxBound. Pushing a value through the tree is done by the push function:

```
push maxl Tip = (Tip, Just maxl)
push min (Fork _ u max lt Tip) = let (lt′, Just l) = push min lt in (fork l (pred u) lt′ Tip, safeSucc max)
push min (Fork _ u l′ lt rt@Fork { }) =
  let (lt′, Just l) = push min lt
      (rt′, max) = push (succ l′) rt
  in  (fork l (pred u) lt′ rt′, max)

safeSucc :: (Bounded a, Enum a, Eq a) ⇒ a → Maybe a
safeSucc x | x ≡ maxBound = Nothing
           | otherwise    = Just (succ x)
```

The push function considers three cases: if the tree is a Tip, then no value can be pushed, and the element that was provided is the maximum element required by complement. The Fork cases first recursively push the given minimum element into the left-child, returning a new lower-bound for this range: this is guaranteed to exist, since the tree is **Ordered** and values larger than this value are known to exist. Then the remainder of the case depends on whether the right-child of the tree is a Tip or not:

- When the right-child is a Tip, the largest element of the fork's range is the maximal element in this tree. The tree is reconstructed with the largest element from the left-child up to the predecessor of the former lower-bound of the tree. The maximal element is returned after taking its successor (accounting for if it is maxBound).
- Otherwise, if the right-child is a Fork, the largest element of this tree is the largest element of the right-child. The former upper-bound of this range is pushed down into the right-child recursively: this succ is safe, since the right-child has elements larger than this range.

5.2.6 Discussion

In all, the implementation of the RangeSet still retains the balancing scheme of AVL trees and can make use of similar optimised algorithms to Set. The presence of ranges, however, do add additional complexity, particularly where overlap must be considered for composite operations. Preserving the invariances of the tree is important to ensure that no pathological behaviours occur with misbalanced trees, as well as ensuring that the size of the tree stays minimal with respect to the elements inside it. If adjacent ranges were allowed to not fuse, the tree may quickly degenerate into a more expensive version of the original Set.

It is worth noting that it is also possible for the complement operation to be supported for infinite types too, so long as the tree is extended with the idea that a range may extend on infinitely. This could be accomplished by adding ForkInfR and ForkInfL nodes to the tree, but this will have an impact on the size of the code for the set's operations (and in turn will likely impact the performance of the code); alternatively, a Maybe a could be used to store the elements, but this introduces additional indirection and memory-overhead, and would also reduce the performance of the tree itself.

5.3 Optimising `RangeSet`

The implementation of RangeSet in §5.2 is naïve, only considering optimisations that remove impossible cases, like unrolling the Fork constructor to remove non-sensical Tip cases and make total functions.

5.3.1 Generic Optimisations

There are a variety of generic optimisations that have been implemented to improve the performance:

- Applying strictness annotations to help GHC unbox data like the height, along with elements as they are passed through functions; as well as eliminating redundant, and expensive, thunks.
- Applying `UNPACK` pragmas to the non-polymorphic, strict, fields of the Fork constructor: this allows GHC to avoid any unnecessary indirection in these fields and improves the cache locality and memory requirements of the tree.

- Applying `INLINE` and `INLINABLE` pragmas to relevant functions and set operations: this allows GHC to optimise cross-module, which is important for specialising the operations and eliminating dictionaries passed around.
- Substituting any pairs used for unboxed tuples: these function mostly as regular tuples, but must be deconstructed immediately, as they are not allocated on the heap but instead kept in registers or on the stack. This reduces the allocations needed for functions like push from §5.2.5 and allows for data to be unboxed in these positions too.

5.3.2 Leveraging Enumerations

There is another, complementary, optimisation that can specifically applied to RangeSet because of the Enum constraint on its elements. In particular, the Enum typeclass demands that there is a reversible mapping from an enumerable type a onto Int: the existing functions described in this paper mostly use succ and pred, however this property is used to compute the enumDiff of two values.

The problem with the current implementation is that it is reliant on GHC to specialise the set operations to eliminate typeclass dictionaries for both Enum a and Ord a and the tree itself must store boxed elements as they are polymorphic. However, since RangeSet is known to only work on the subset of its elements that can be safely mapped to Ints, this can be leveraged instead in the internal representation:

```
type E = Int
data RangeSet a = Fork  {-# UNPACK #-} !Word8
                        {-# UNPACK #-} !E  {-# UNPACK #-} !E !(RangeSet a) !(RangeSet a)
                | Tip
```

Performing this transformation makes the type parameter a of the set phantom, and elements of the set are replaced by Int. This of course assumes that the programmer only uses types for which there is an injection onto Int, so RangeSet Double would be illegal, even though it is Enum – this burden is on the programmer to respect. The benefit of using Int is that these elements can be unpacked into the structure, further reducing memory requirements, and improving cache locality. Furthermore, the elements can be more readily be unboxed after strictness analysis in the functions themselves, without relying on specialisation first. The elements of the structure will also no longer require unboxing and re-boxing as they are inspected during traversals. As an additional benefit, Int already has an Ord constraint, and so the Ord constraint on each of the operations can be dropped, only requiring an Enum constraint. Care must be taken, however, when making use of minBound and maxBound at any point: these must be obtained from the Bounded a instance and not Bounded Int!

5.3.3 Unsafe Operations

Another optimisation that is applicable to RangeSet is to selectively abandon the use of the smart-constructor fork, or the other safe functions like balance or even balanceL. In certain situations, it is clear to see that, for instance, the height of a tree does not change: this means that redundant computation performed by fork can be avoided, and Fork h used instead. This has been done in the implementation discussed in this chapter, for instance using disjointLink and disjointMerge instead of safer link and merge functions: in these cases, it was shown that the operations are indeed safe in context.

Aggressive Unrolling

One final optimisation that can be applied to the RangeSet implementation is aggressively unrolling the Fork constructor when it is the only possible case. This helps create total functions (and was employed in the insertLAdj function) and reduces redundant pattern matching. However, it has not been applied as aggressively as it could have been for clarity. In particular, split, overlap, intersect, rotr, rotl, and the balanceL and balanceR functions can all be unrolled in this way, making them total and slightly more optimised.

5.4 Benchmarks

5.4.1 Comparison to `Data.Set` and `Data.IntSet`

To test the implementation of the optimised RangeSet against Set and Patricia trees, benchmarks have been formulated that measure the improvement offered as the contiguity of the data increases.

The contiguity of a RangeSet is measured by the formula: $1 - \frac{m}{n}$ where, m is the number of ranges in the set and n is the number of elements. It is a heuristic that estimates how well data is bunched together: the RangeSet should perform better as this value tends to 1.

The data for the benchmarks is obtained as follows. First, 20 bins are created, for each contiguity from 0.0 to 0.95; for each contiguity, a set of size 1000 is synthesised by sequentially taking elements from the list [0..], skipping values when required to break contiguous ranges to achieve the desired contiguity. Each generated element list is then shuffled, and all its elements shifted linearly by a random value between 0 and 1000 a total of 99 times. This forms 100 sets of equal contiguity, with a variety of element orderings and various degrees of overlap with each other. The shuffling affects the performance of operations that build up, or tear down, sets; and the shifting ensures that there is a variety of different matchups for union, intersection, and difference – other operations are shift-invariant.

Each generated element list is then used to build a RangeSet, a Set, a IntSet Patricia tree, sorted, and left as is. Operations are then performed on each bundle of 100 sets uniformly: since this involves traversing all the sets (and potentially performing multiple operations on each) each iteration of the benchmark, the overheads of the benchmark iteration are also measured. This overhead is subtracted from the results of the corresponding benchmarks, and any uncertainty in the measurements is additively combined with the uncertainty from the true measurement, which is standard practice. For clarity, the results are normalised to be relative to rangeset; when combining the data to form a relative measure, the uncertainties in the measurements ΔT_s and ΔT_r are combined into the relative uncertainty ΔR by the following formula:

$$\Delta R = \frac{T_s}{T_r}(\frac{\Delta T_s}{T_s} + \frac{\Delta T_r}{T_r})$$

In §5.3, the RangeSet was transformed to make use of unpacked Ints to store the elements, as opposed to boxed values. This gives it an edge over Set, which still has this indirection. To test the effect this has on Set, it has been optimised to also require an Enum constraint so that it can make use of the same optimisation without the additional range machinery. This has been included in the plots for each benchmark as “Set (optimised)”: it should not change the shape of the results but will shift the graph down. The data used to benchmark the sets uses Word as the set element; with a more complex type, like Peano naturals, the cheaper internal Int operations further create distance between the unoptimised and optimised forms.

Membership Queries

The first benchmark times how long it takes to query all elements within the set against itself: this means that all the queries will return True. The result of this is shown in fig. 5.2. As expected, as the contiguity of the data increases, the average query time for the elements decreases. In fact, unoptimised Set was slower across all bins despite the slight increase in logic for testing both parts of the range; this is due to the different balancing properties of Set, which may mean it is less well-balanced than RangeSet. At high contiguities, RangeSet may perform up to 2 times faster than Set. Patricia trees, however, outperform RangeSet by up to a factor of 2, but this gap closes the more contiguous the data gets. Testing elements that are not found in the set also shares a similar shape to the results for included elements, so this omitted for brevity.

Insertion and Deletion

The second type of benchmark tests how quickly a set can be constructed, or destructed, from both unsorted and sorted data. For these benchmarks, the entire set is either constructed from the empty set, or destructed until it becomes the empty set.

Unsorted Insertion and Deletion The results of this benchmark are shown in figs. 5.3 and 5.4: above 0.25 contiguity, RangeSet’s insertion outperformed the unoptimised form of Set for insertion, however the optimised Set is faster until about 0.85 contiguity. At best, RangeSet offers a 1.2× improvement over base’s version of Set. Again, Patricia trees are much more performant, and do not seem to lose their edge as contiguity increases.

For deletion, the unoptimised Set is worse from 0.15 contiguity onwards, with the optimised form only becoming slower at 0.9 contiguity. The performance of Patricia trees degrades around 0.9 contiguity, but the

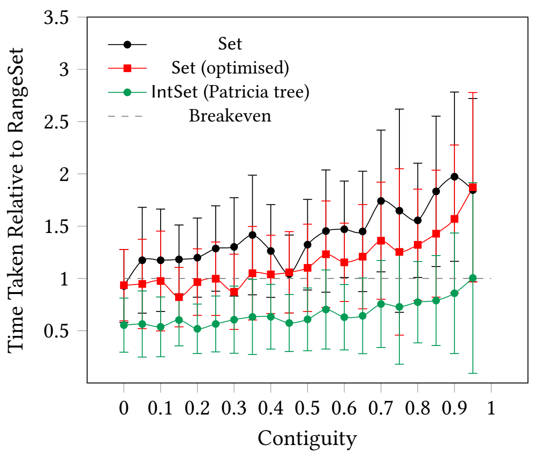


Fig. 5.2: Testing membership of only members of the set

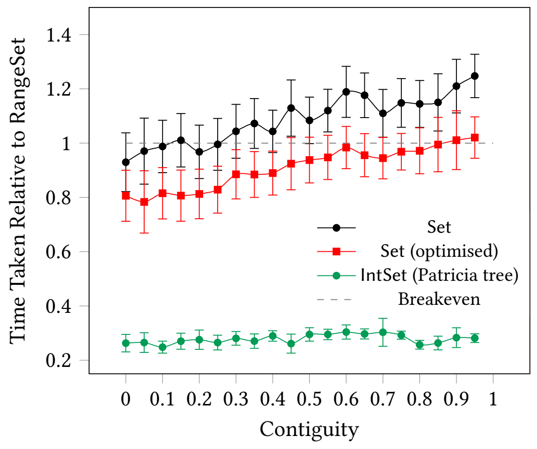


Fig. 5.3: Testing unsorted insertion from empty set

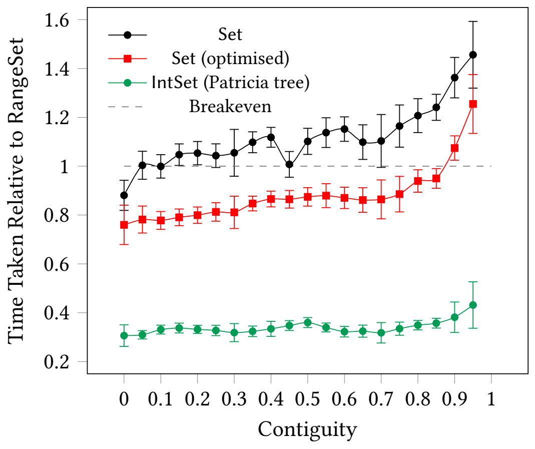


Fig. 5.4: Testing unsorted deletion until empty set

effect is likely only evident in a much larger tree.

Sorted Construction For sorted data, however, the results are more dramatic: at around 0.35 contiguity, sorted insertion (fig. 5.5) is firmly better for RangeSet than the unoptimised Set, with the optimised one degrading at around 0.7 contiguity. Compared with the unsorted variant, however, there is a maximal speedup of 3× for the unoptimised set and 2.5× for the optimised Set. The shape of this graph is similar for deletion (fig. 5.6), but with less extreme improvements. The reason for this is likely that sorted data will produce a set with the right contiguity faster, reducing the nodes in the tree and therefore the work that needs to be performed. However, Patricia trees continue to dominate here, being around 5× faster for most contiguities.

Union, Intersection, and Difference

Each of union (fig. 5.7), intersection (fig. 5.8), and difference (fig. 5.9) have similar shaped graphs, with a sharp performance improvement at high contiguities compared with both optimised and unoptimised Set. Against the unoptimised form the performance improvement ranges from 4× to 5×. Once again, however, a Patricia tree is much faster than RangeSet, owing to its linear time complexity for these operations.

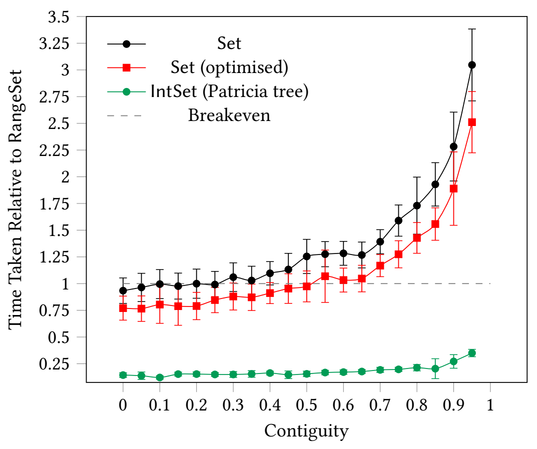


Fig. 5.5: Testing sorted insertion from empty set

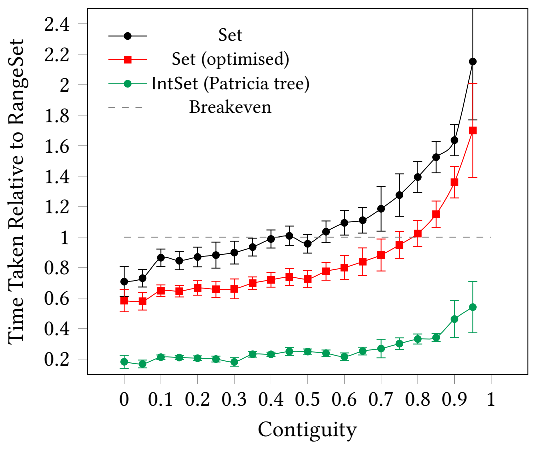


Fig. 5.6: Testing sorted deletion until empty set

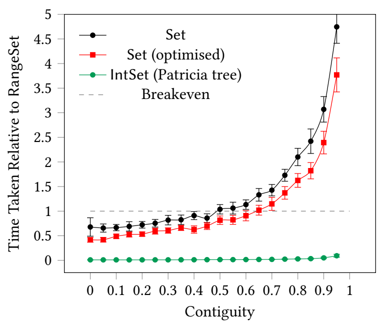


Fig. 5.7: Union of two sets

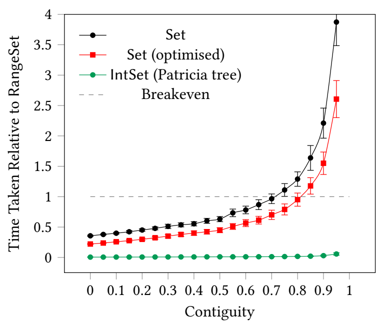


Fig. 5.8: Intersection of two sets

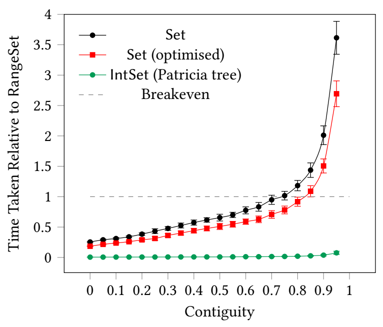


Fig. 5.9: Difference of two sets

Discussion

In all, however, RangeSet performs competitively with Set, and has an advantage when the data is more contiguous. However, it does not perform as well when compared to Patricia trees.

5.4.2 Quantified Time Complexity

The complexity proofs for `Data.Set` presented in Blelloch, Ferizovic, and Sun [2016] are difficult to apply directly to the implementation of `Data.RangeSet` developed within this chapter; nevertheless, it is useful to at least informally prove the bounds, as I expect that the implementation does not differ substantially enough in the work it is required to do to warrant a worse (or better) complexity bound.

One substitute for complexity proofs are benchmarks demonstrating that the average performance of a data-structure operation f, T_f, is within the bounds of the complexity function g scaled with some constant δ:

$$T_f(n) \in O(g(n)) \Leftrightarrow \exists \delta > 0. \exists n_0 > 0. \forall n > n_0. T_f(n) \leqslant \delta g(n)$$

However, in the case of the three Set × Set operations, the complexity is in terms of two variables: the sizes of the two argument sets m and n, such that $m \leqslant n$. As such, the desired property for compound operation f on sets X and Y with the smaller having size m and the larger n would be:

$$T_f(m,n) \in O(m \log_2(\frac{n}{m}+1)) \Leftrightarrow \exists \delta > 0. \exists m_0 > 0. \forall m > m_0. \forall n \geqslant m. T_f(m,n) \leqslant \delta m \log_2(\frac{n}{m}+1)$$

Plotting the surface $\delta m \log_2(\frac{n}{m}+1)$ is, however, difficult to do in a way that demonstrates clearly that the benchmarking samples of f fall below the surface. Instead, benchmarks are taken and then normalised with respect to the asymptotic bound so that the function g_δ is plotted instead, which has co-domain $\{x \mid x \geqslant -1.0\}$.

$$g_\delta(m,n) = \frac{T_f(m,n) - \delta m \log_2(\frac{n}{m}+1)}{\delta m \log_2(\frac{n}{m}+1)}$$

The complexity guarantee can be reframed using g_δ as:

$$T_f(m,n) \in O(m \log_2(\frac{n}{m}+1)) \Leftrightarrow \exists \delta > 0. \exists m_0 > 0. \forall m > m_0. \forall n \geqslant m. g_\delta(m,n) \leqslant 0$$

For all the benchmarks, $\delta = \frac{|T_f(N,N)|}{N}$ for the largest N tested in the symmetric case, such that $g_\delta(N, N) = 0$. Each of the operations union, intersect and difference are benchmarked against sets from size 500 to size 5000 in two configurations: both sets are entirely disjoint, or they are entirely overlapping. The candidate sets are shuffled before construction so that two overlapping sets will not have exactly the same shape – this helps eliminate outliers arising from optimisations that do not rebuild sets that have not changed. The candidate sets all have pathological discontiguity, so that disjoint sets create a pathologically sized resulting range-set; this is the worst-case for the performance of the sets, and more contiguous sets result in faster runs (as seen in §5.4.1).

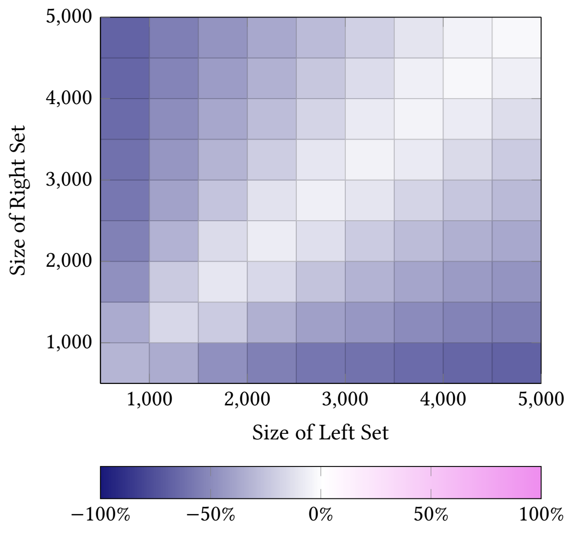


Fig. 5.10: Relative deviation from expected average performance for union x y (disjoint)

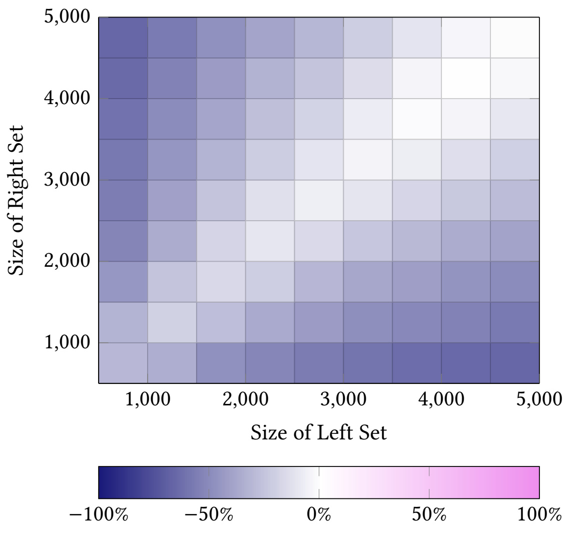


Fig. 5.11: Relative deviation from expected average performance for union x y (overlapping)

Complexity of union Figs. 5.10 and 5.11 show the plotted function g_δ for union with both disjoint and overlapping sets. The point $(5000, 5000)$ is the slowest benchmarked point, and is therefore set to 0 by δ: this means the point is rendered in white. All other points on the graphs are between white and bluish-purple, which represents a value that is faster than expected. The majority of the mid-line where both sets have equal sizes is white, and the underlying data follows an almost perfectly linear increase: $O(n \log_2(\frac{n}{n} + 1)) = O(n)$. As such, it is highly likely that union does have the expected worst-case complexity bound.

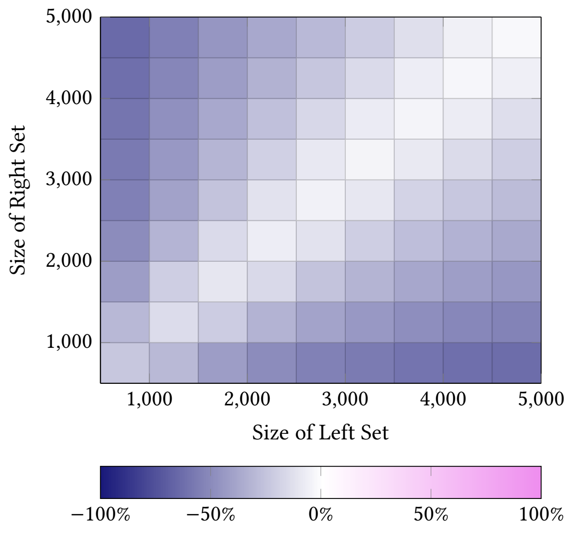


Fig. 5.12: Relative deviation from expected average performance for intersect x y (disjoint)

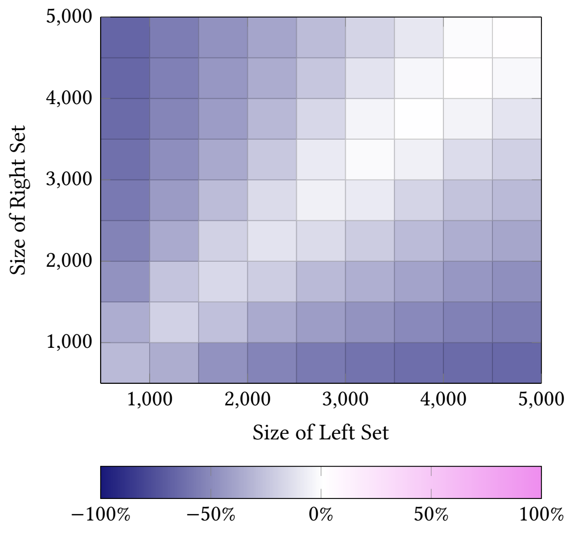


Fig. 5.13: Relative deviation from expected average performance for intersection x y (overlapping)

Complexity of intersection Figs. 5.12 and 5.13 show the plotted function g_δ for intersect with the same set pairings. Again, the slowest point is $(5000, 5000)$, which is set to 0, and all other points fall comfortably below the expected values across the graphs. As such, intersect also most likely has the advertised worst-case complexity bounds.

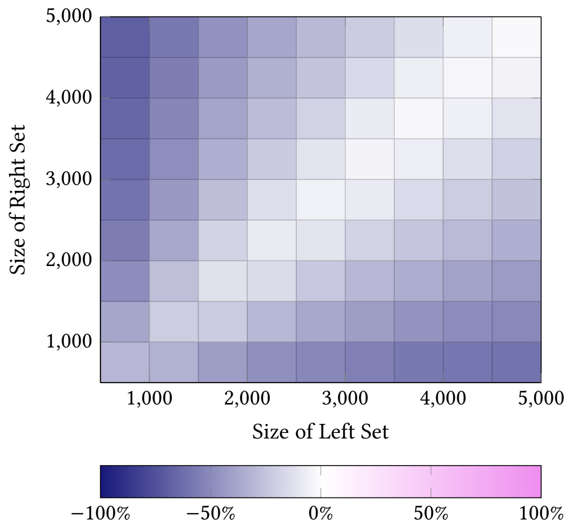


Fig. 5.14: Relative deviation from expected average performance for difference x y (disjoint)

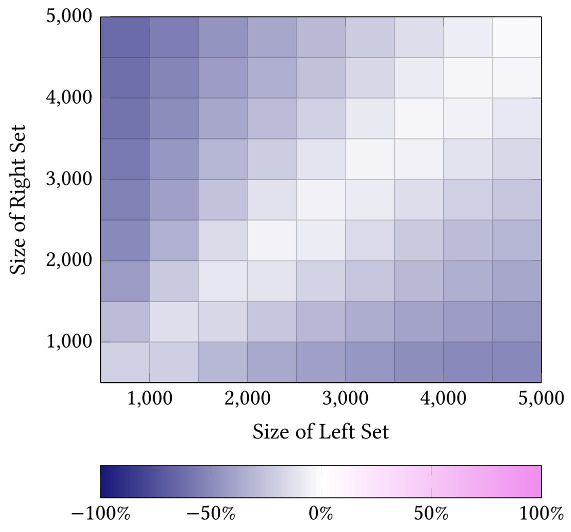


Fig. 5.15: Relative deviation from expected average performance for difference x y (overlapping)

Complexity of difference Figs. 5.14 and 5.15 show the plotted function g_δ for difference with the same set pairings. Once again, the slowest point is $(5000, 5000)$, which is set to 0, and all other points fall comfortably below the expected values across the graphs. This indicates that difference also likely has the advertised complexity. Notably, difference does have a small bias in both scenarios, where a smaller right set results in a lighter colour purple compared to the symmetric part of the grid where the left set is smaller. This means that having a smaller right set makes difference perform slightly worse: this is to be expected, since it will result in more items being present in the resulting set, which requires more work to rebuild. This bias does not occur for `union` and `intersect`, because they are commutative so the taller set can be put into the cheapest position.

Discussion

The idea behind RangeSets is to specialise the structure of a common data structure to a specific shape of data, or a specific pattern of use. This is useful when a problem has domain-specific characteristics that lend themselves well to these specialisms.

The implementation of RangeSet is inspired by the classic AVL tree, which self-balances to avoid pathological tree shapes. By introducing the notion of ranges to the structure, data can be easily compressed into a node, both reducing the size of the tree and decreasing the worst-case traversal time. This does increase the complexity of the structure, and the additional overhead needed to maintain the more complex structure must be balanced with the potential performance improvements that can be brought: if the data is not well suited and highly contiguous, the benefits of the structure are lost.

A key lesson of this implementation is to recognise that the imposed constraint of `Enum` not only facilitates the minimum interface necessary to ensure that it is possible to check for fusion opportunities in tree, but also allows for the representation of elements themselves to be optimised. The fact that `Enum` is reliant on a mapping to `Int` is important as this monomorphic type can be more readily optimised, especially in a strict data structure like `RangeSet`, and this brought improvements to both `RangeSet` (and indeed `Set`).

In all, RangeSet performs well given favourable conditions (and reasonably even given unfavourable conditions) and can be more performant than Set from `containers`. Thankfully, the parsing domain does offer a wealth of semi-contiguous data in the form of character predicates, so this structure is at its strongest here. However, Patricia trees are faster than RangeSet even in its most favourable conditions; there a couple of advantages RangeSet has, however, which make it suitable for `parsley` in particular:

- The memory requirements of a RangeSet are still several orders of magnitude lower for large sets of characters, such as the set of all characters, which does appear in practice.
- The RangeSet supports a fast and memory efficient complement operation, which is useful for negating character predicates within static analysis.
- The static structure of a RangeSet can be naturally exploited to generate optimised character predicates by applying staging to the member function: this is something done by `parsley` in its code generator (§6.1.3). The underlying structure of a Patricia tree does not reveal anything about the nature of its elements in a usable way.

In addition, RangeSet can be generalised more easily to support infinite datatypes, including an infinite-compatible complement operation.

The discussion in Chapter 6: Analysis and Optimisation will rely on RangeSet as the implementation of static character predicates (§6.1.3).

Chapter 6

Analysis and Optimisation

This chapter builds on the groundwork laid in both chapters 3 and 4; the implementation during those chapters was incremental, and not all parts of the overall code changes were shown. To aid understanding in this chapter most of the relevant parts of the final code from after chapters 3 and 4 can be found in appendix D. Code changes during this chapter will continue to be highlighted.

The implementation of `parsley` seen so far has been focused around establishing a simple parametric translation of high-level combinators to an abstract machine. However, while this produces decent code, there is much that can be improved on: while GHC is very good at optimising and analysing Haskell code, it does not have domain-specific knowledge about parser combinators that it can use to optimise things further. Instead, the staged nature of the `parsley` allows for this information to be exploited by the metaprogrammer instead. This works by trying to move previously dynamic information to compile-time so that it can be eliminated.

There are three primary sources of dynamic content within the generated parsers of Chapter 3: Embedding a Parser Combinator Library: handler bindings, non-terminal bindings, and input consumption. As such, it makes sense to focus attention on improving these aspects of the generated parser:

- First, §6.1 lays some groundwork by improving the static inspectibility of both user-defined values and predicates, as well as ensuring these are generated in an optimal way.
- Then, §6.2 covers some of the more general-purpose optimisations that can be applied across the Combinator and Instr trees. It will leverage the handler idempotency identified in §4.2.1 to provide an improved representation of iterative non-terminals.
- §6.3 focuses on static analysis of input consumption along paths of the parser, to allow for input fetching to be factored out, reducing repetitive backtracking.
- §6.4 exploits information known statically about whether input must have been consumed along a branch to compile away dynamic checks for input equality.

Finally, §6.5 evaluates the performance of the completed system, including the effectiveness of the developed optimisations and a comparison to other popular parser combinator implementations in Haskell.

6.1 Refining User-Defined Functions

Currently, the definition of both Combinator and Instr in Chapter 3: Embedding a Parser Combinator Library take user-defined values and functions as pure Code. The poor static inspectibility of these values limits their use for optimisation. In this section, steps are taken to improve this, which facilitates various static improvements across the rest of the chapter.

6.1.1 Defunctionalisation

The easiest way to make statically uninspectable values inspectable is to render them as a datatype. This is a process known as *defunctionalisation* [Reynolds 1972; Danvy and Nielsen 2001], which provides many of the same benefits as a *deep embedding* provides to a DSL. With an infinite number of values and functions to choose from, deciding what to defunctionalise and what to leave is not necessarily clear-cut; in this case, things that are referred to explicitly within the internal definitions of composite combinators are good candidates. The Defunc datatype describes this set of values:

```
data Defunc a where
  ID      :: Defunc (a → a)
  COMPOSE :: Defunc ((b → c) → (a → b) → (a → c))
  FLIP    :: Defunc ((a → b → c) → (b → a → c))
  EQ      :: Eq a ⇒ Defunc (a → a → Bool)
  LIFTED  :: (Lift a, Typeable a) ⇒ a → Defunc a
  CONST   :: Defunc (a → b → a)
  CONS    :: Defunc (a → [a] → [a])
  NIL     :: Defunc [a]
  LEFT    :: Defunc (a → Either a b)
  RIGHT   :: Defunc (b → Either a b)
  PAIR    :: Defunc (a → b → (a, b))
  (:$:)   :: Defunc (a → b) → Defunc a → Defunc b
  OPAQUE  :: Code a → Defunc a
```

Each of the constructors represents a specific value or function, except for LIFTED, (:$:), and OPAQUE. The LIFTED constructor can represent any value so long as its monomorphised type is known and it is liftable; the OPAQUE constructor represents any unknown uninspectable value, which includes anything the user provides; and the left-associative (:$:) represents higher-order application of one value to another, which can be used to form many useful combinators:

```
FLIP_H :: Defunc (a → b → c) → Defunc (b → a → c)
FLIP_H f = FLIP :$: f

COMPOSE_H :: Defunc (b → c) → Defunc (a → b) → Defunc (a → c)
COMPOSE_H f g = COMPOSE :$: f :$: g
```

The addition of Defunc means that the Combinator tree should change to support the new values, in particular:

```
Pure :: Defunc a → Combinator k a

pure :: Code a → Parser a
pure = In · L · Pure · OPAQUE
```

The Satisfy constructor is the other constructor that uses Code, however a more specialised defunctionalised representation is beneficial for this (§6.1.3). The definitions of the combinators can be changed, however, to make use of these new defunctionalised values. For simplicity assume that there exists an overloading that allows any combinators to be used with either Defunc or OPAQUE wrapped code values[1].

```
(<:>) = liftA2 CONS
(<**>) = liftA2 (FLIP_H ID)
prefix :: Parser (a → a) → Parser a → Parser a
prefix op p = postfix (pure ID) (FLIP_H COMPOSE <$> op) <*> p
```

These are examples of how the combinators might change to make use of the new Defunc abstraction. The use of Defunc for performing inspection will be explored in §6.2. The function defuncToCode :: Defunc a → Code a can be used to exit the defunctionalised representation; however, if defined directly, it is difficult to avoid generating needlessly complex terms. As such, the definition of this function is left to §6.1.2, where a different representation language allows for the generation of optimised terms; this is useful, since while GHC may optimise it, this is not guaranteed – especially on lower optimisation levels – and may "distract" GHC from optimising aspects of the code outside of the metaprogrammer's control.

6.1.2 Staged HOAS Lambda Calculus

While the Defunc type is great for static inspection of different values, generating optimal code for it is an involved process, as there are many special cases to consider. Normally, compilers translate higher-level representations to more primitive representations (like GHC's *Core* language). The core language for both character predicates and other user-defined values is the lambda calculus [Church 1936]. To avoid the burden of dealing with variable binders – normally done by names or De Bruijn indices [de Bruijn 1972]– the Lam type is formulated as higher-order abstract syntax (HOAS) [Pfenning and Elliott 1988], where Haskell's binders are used to represent binders in Lam:

```
data Lam a where
  Abs :: (Lam a → Lam b) → Lam (a → b)
  App :: Lam (a → b) → Lam a → Lam b
  Var :: Code a → Lam a
  Let :: Lam a → (Lam a → Lam b) → Lam b
  If  :: Lam Bool → Lam a → Lam a → Lam a
  T   :: Lam Bool
  F   :: Lam Bool
```

Abs represents lambda abstraction, where a Haskell function is used to represent the lambda, abstracting the names as desired; App is application of one term to another; Var represents meta-variables in the language, which are just Code; Let is an extension that allows for binding of a variable to prevent duplication without a lambda, it corresponds to a Haskell let-binding; If, T, and F are also an extension for simple boolean logic, which is useful for optimising some control flow within the abstract machine.

[1]In practice, this can be done in a non-intrusive way, as briefly outlined at the end of §6.1.

Normalisation by Staging

The role of Lam is to be normalised into an optimised term, which is staged to produce an optimised Haskell term. The term is optimised in the sense that it is as fully reduced as possible without being able to reduce the meta-variables. Since the lambda abstractions are represented by Haskell functions, however, it is only possible to normalise a Lam term into Weak-Head Normal Form (WHNF) [Peyton Jones 1987; Plasmeijer and Eekelen 1993], more specifically β-WHNF, where no β-reductions are possible on the head term. Checking for η-reductions is not possible, as Haskell functions cannot be inspected by the programmer. A Lam term t is considered in β-WHNF if normal t is True:

```
normal :: Lam a → Bool
normal (App (Abs _) _) = False
normal (App f _)       = normal f
normal (If T _ _)      = False
normal (If F _ _)      = False
normal (If _ T T)      = False
normal (If _ F F)      = False
normal (If _ T F)      = False
normal (If c x y)      = normal c ∧ normal x ∧ normal y
normal _               = True
```

This encodes the standard rules for β-WHNF, with the additional rules that if-expressions must be fully reduced and cannot be simplified with boolean logic. With this function, it is possible to define a function normalise for putting a term into β-WHNF:

```
normalise :: Lam a → Lam a
normalise x
   | normal x  = x
   | otherwise = reduce x

reduce :: Lam a → Lam a
reduce (App (Abs f) x)                          = normalise (f x)
reduce (App f x) = case reduce f of f@Abs { } → reduce (App f x)
                                    f         → App f x
reduce (If T t _)                               = normalise t
reduce (If F _ f)                               = normalise f
reduce (If _ T T)                               = T
reduce (If _ F F)                               = F
reduce (If c T F)                               = normalise c
reduce (If c t f)                               = normalise (If (normalise c) (normalise t) (normalise f))
reduce x                                        = x
```

The normalise function makes use of reduce, which defines the single steps taken to normalise the head of the term: applications with an abstraction on the left can reduce by using Haskell function application, otherwise the function is reduced; conditionals are reduced by removing redundant conditionals where possible. However, generating a fully normalised term is the goal, not β-WHNF – i.e., η-β normal form ($\eta\beta$-NF).

As above, the problem is that reduction cannot be applied underneath Abs if there are no arguments that can be provided to them. As it happens, in the process of generating code, staging provides a way to look underneath the lambda abstractions and normalise them. First normalise a term to β-WHNF and then generate code:

```
normaliseGen :: Lam a → Code a
normaliseGen = generate · normalise
```

The claim is that normaliseGen establishes a β-NF for the term (rendered as Code). If this is true then with access to an etaReduce :: Code a → Code a function, the term can be put into full $\eta\beta$-NF by η-reducing the generated code by inspection:

```
normaliseGen :: Lam a → Code a
normaliseGen = etaReduce · generate · normalise
```

The etaReduce function is implemented by untyped Template Haskell, by matching on various constructors that introduce lambda abstractions: its implementation is not important. The missing piece of the puzzle now is the generate function, which establishes a β-NF by structural induction on Lam:

```
generate :: Lam a → Code a
generate (Abs f)   = ⟦λx → $(normaliseGen (f (Var ⟦x⟧)))⟧
generate (App f x) = ⟦$(generate f) $(normaliseGen x)⟧
generate (Var x)   = x
generate (If c t e) = ⟦ if $(generate c) then $(generate t) else $(generate e)⟧
generate (Let b i)  = ⟦ let x = $(normaliseGen b) in $(normaliseGen (i (Var ⟦x⟧)))⟧
generate T         = ⟦True⟧
generate F         = ⟦False⟧
```

The terms provided to generate are in β-WHNF, which means that no redex is possible on the head-most term. As such, a term Abs f is converted directly to a Haskell lambda, since there is no other way of reducing it. This is where the magic happens: when translated to code, the introduction of the binding for x allows the meta-variable Var ⟦x⟧ to be provided to f, which removes the uninspectable function allowing for normaliseGen to be inductively called on the resulting exposed term. This immediately establishes a head normal form, as no further reductions can be done in the lambda body – in fact, by induction the term overall is in β-NF. In the other cases, generate or normaliseGen is used depending on whether the term is known to be already normalised, though there would be no harm in applying normaliseGen uniformly: their definitions are standard structural induction. As such, normaliseGen will generate an irreducible Haskell term from a corresponding Lam term, excluding the meta-variables. Ultimately, this is useful as it reduces the optimisation burden on GHC to establish the normal form itself: a smaller amount of generated code will be more readily optimised.

Translating `Defunc` to `Lam`

The Lam type is designed to be used as the bridge between the high-level Defunc type and raw Haskell code, when no further inspection of the terms is required. It must be converted to Lam before they can be generated into real code, which allows them to benefit from the $\eta\beta$-NF established by Lam during generation.

Converting Defunc to Lam is straightforward, and most constructors are plainly converted to meta-variables, with some exceptions that can make more intelligent use of the Lam type. The function defuncToLam is responsible for the conversion:

```
defuncToLam :: ∀a.Defunc a ⟶ Lam a
defuncToLam ID              = Abs id
defuncToLam COMPOSE         = Abs (λf ⟶ Abs (λg ⟶ Abs (App f · App g)))
defuncToLam FLIP            = Abs (λf ⟶ Abs (λx ⟶ Abs (λy ⟶ App (App f y) x)))
defuncToLam EQ              = Var ⟦(≡)⟧
defuncToLam (LIFTED x)
   | Just Refl ⟵ eqT @a @Bool = if x then T else F
   | otherwise              = Var (lift x)
defuncToLam CONST           = Abs (Abs · const)
defuncToLam CONS            = Var ⟦(:)⟧
defuncToLam NIL             = Var ⟦[ ]⟧
defuncToLam LEFT            = Var ⟦Left⟧
defuncToLam RIGHT           = Var ⟦Right⟧
defuncToLam PAIR            = Var ⟦(, )⟧
defuncToLam (f :$: x)       = App (defuncToLam f) (defuncToLam x)
defuncToLam (OPAQUE x)      = Var x
```

Defunctionalised values that are neither constructors nor type-class methods can be represented directly in the Lam language: ID is an identity abstraction, COMPOSE composes two terms bound by lambda abstractions, and so on for FLIP and CONST as well. The application constructor (:$:) is also converted directly to an App on the recursively converted sub-terms. Other constructors, and EQ, are plainly converted to meta-variables, though LIFTED x :: Defunc Bool can be inspected further to convert it to one of T or F – this is done by doing a type-check on the Typeable instance packaged into the LIFTED constructor.

`Lam` in the abstract machine The Defunc type is used to represent arbitrary user-defined values in the Pure combinator, but Lam is used to represent these same values within the machine. The Push and Red instructions are altered to make use of it:

```
Push :: Lam x ⟶ k (x : xs) n a ⟶ Instr k xs n a
Red  :: Lam (x ⟶ y ⟶ z) ⟶ k (z : xs) n a ⟶ k (y : x : xs) n a
```

Within the compile function, the defuncToLam function is used to convert the defunctionalised values used in Pure, and Lam is used directly otherwise.

6.1.3 Character Predicates

While Defunc is useful for describing a wide variety of user-defined functions in an inspectable way, it is not the ideal representation for handling character predicates (Char → Bool functions). Static knowledge of these predicates unlocks many interesting behaviours and compared with arbitrary values their structure is much more predictable. In fact, unlike arbitrary functions, it is possible to lift static character predicates into Code via "The Trick" (§2.4.1):

```
data CharPred where
  UserPred :: Lam (Char → Bool) → CharPred
  Ranges   :: RangeSet Char → CharPred
```

The CharPred type is a simple datatype but allows for character predicates to be inspected throughout the `parsley` compiler: UserPred represents unanalysable predicates, which may be partially defined via Lam or are simply Var meta-variables; and Ranges represents fully static predicates, which are represented in the compact fully inspectable RangeSet (CHAPTER 5). The following smart constructors can be used to generate CharPreds:

```
defuncPred :: Defunc (Char → Bool) → CharPred
defuncPred (EQ :$: LIFTED c)           = Ranges (RangeSet.singleton c)
defuncPred (CONST :$: LIFTED True)  = Ranges (RangeSet.complement RangeSet.empty)
defuncPred (CONST :$: LIFTED False) = Ranges (RangeSet.empty)
defuncPred pred                        = UserPred (defuncToLam pred)

dynPred :: Code (Char → Bool) → CharPred
dynPred = UserPred · Var

listPred :: [Char] → CharPred
listPred = Ranges · RangeSet.fromList

staPred :: (Char → Bool) → CharPred
staPred p = listPred (filter p [minBound . . maxBound])

setPred :: Set Char → CharPred
setPred = listPred · Set.toList
```

There are a couple of interesting things to note with these: defuncPred can inspect the values to convert them into static predicates, and staPred performs "The Trick" over all characters to apply the predicate exhaustively, allowing it to be converted into a static RangeSet. These can be used by the high-level API to construct the various kinds of predicates for predefined parsers and combinators. The Satisfy combinator and Sat instruction are adjusted to use the CharPred type:

```
Satisfy :: CharPred → Combinator k Char
Sat     :: CharPred → k (Char : xs) (Succ n) a → Instr k xs (Succ n) a
```

The presence of CharPred allows for the analysis and optimisation of character predicates even more than would otherwise be allowed by Defunc.

Predicate Combinators

Character predicates, unlike normal Defunc values, may need to be manipulated during optimisation. There are several combinators that can be used for this purpose, which interact with both Lam values and RangeSets.

```
notPred :: CharPred → CharPred
notPred (UserPred p)   = UserPred (Abs (λx → not_lam (App p x)))
notPred (Ranges rngs) = Ranges (RangeSet.complement rngs)

not_lam :: Lam Bool → Lam Bool
not_lam b = If b F T
```

The notPred function can be used to invert a predicate, which either complements the set or applies a $\text{not}_{\texttt{lam}}$ combinator under an abstraction: this can simplify cases with constants when normalised – as an example UserPred (Abs (const T)) negates to UserPred (Abs (const F)). Since boolean combinators generate slightly simpler code, it is worthwhile to add three special cases to the Lam generation as peephole optimisations:

```
generate (If c T e) = ⟦$(generate c) ∨ $(generate e)⟧
generate (If c t F)  = ⟦$(generate c) ∧ $(generate t)⟧
generate (If c F T) = ⟦not  $(generate c)⟧
```

These ensure that the boolean (∧), (∨), and not operations are all compiled in their optimised forms. Along with this there are two additional Lam combinators:

```
x ∧_lam y = If x y F                    x ∨_lam y = If x T y
```

These two combinators can be used to define andPred and orPred, respectively:

```
orPred, andPred :: CharPred → CharPred → CharPred
orPred (Ranges rngs_1) (Ranges rngs_2) = Ranges (RangeSet.union rngs_1 rngs_2)
orPred p q = UserPred (Abs (λc → App (charPredToLam p) c ∨_lam App (charPredToLam q) c))

andPred (Ranges rngs_1) (Ranges rngs_2) = Ranges (RangeSet.intersection rngs_1 rngs_2)
andPred p q = UserPred (Abs (λc → App (charPredToLam p) c ∧_lam App (charPredToLam q) c))
```

If both arguments are Ranges then they can be merged with the corresponding RangeSet operation, otherwise a new lambda is constructed that performs a disjunction or conjunction or the applications of each predicate to a character. The charPredToLam function is the subject of §6.1.3.

Translate `CharPred` to `Lam`

Like Defunc, converting a CharPred to code is best done via Lam, since it can then piggy-back off the normalisation properties of Lam's conversion to code. In this case, the interesting part is the conversion of a RangeSet Char into a Lam (Char → Bool), which generates a minimised predicate based on the static structure of a fully unrolled RangeSet.

```
RangeSet.fold :: (a → a → b → b → b) → b → RangeSet a → b
```

```
charPredToLam :: CharPred → Lam (Char → Bool)
charPredToLam (UserPred t) = t
charPredToLam (Ranges rngs)
   | rngs ≡ RangeSet.complement (RangeSet.empty) = Abs (const T)
   | otherwise                                   = Abs (λc → RangeSet.fold (fork c) F rngs)
  where fork :: Lam Char → Char → Char → Lam Bool → Lam Bool → Lam Bool
        fork c l u lb rb | l ≡ u        = (c ≡_lam vl) ∨_lam (If (c <_lam vl) lb rb)
                         | l ≡ minBound = (c ≤_lam vu) ∨_lam rb
                         | u ≡ maxBound = (vl ≤_lam c) ∨_lam lb
                         | otherwise    = If (vl ≤_lam c) ((c ≤_lam vu) ∨_lam rb) lb
          where vl = Var (lift l)
                vr = Var (lift r)
        (≤_lam) = App · App (Var ⟦(≤)⟧)
        (<_lam) = App · App (Var ⟦(<)⟧)
        (≡_lam) = App · App (Var ⟦(≡)⟧)
```

The charPredToLam function checks for the obvious cases where all the elements are present and otherwise folds the set using RangeSet.fold, which is a standard shaped fold with one function corresponding to Fork and the other to Tip. A Tip maps to False, and the Fork maps to the fork function, additionally parameterised by the character c introduced by the lambda abstraction surrounding the fold. Other than the first three cases, which are simple optimisations, the last case of fork directly corresponds to the shape of the membership function outlined in §5.2.2:

```
go Tip                          = False
go (Fork _ l u lt rt) | l ≤ x   = x ≤ u ∨ go rt
                      | otherwise = go lt
```

Since RangeSet describes a balanced structure for a set of ranges, and can determine membership in $O(\log n)$, and the folding of the set in charPredToLam follows this structure directly, it follows that the generated predicates are balanced and run in $O(\log n)$.

6.1.4 Updating the Machine

With the changes to the instruction set come changes to the evaluation functions. During evaluation, Lam values are useful to keep around for as long as possible since they can be normalised further when brought into contact with each other. As such, the Moore stack will change to accommodate Lam values directly:

```
data QList xs where
  QNil  :: QList '[ ]
  QCons :: Lam x → QList xs → QList (x : xs)
```

This means that the evalPush function does not need to change (other than its type), but both evalSeek and evalTell need to handle wrapping the input in a meta-variable:

```
evalTell k ctx γ{..} = k ctx (γ {moore = QCons ( Var  input) xs})
evalSeek k ctx γ{..} = let QCons ( Var  input′) xs = moore in k ctx (γ {input = input′, moore = xs})
```

It is safe to assume that all instances of input on the stack will be wrapped in Var, so the unwrapping in Seek is fine. The evalCatch function will also need to be adjusted similarly when it places the input onto the stack for the handler, omitted for brevity. Like evalCatch, evalCall, evalMkJoin, evalGet, and the binding construction for non-terminals need to wrap arguments they place on the stack with Var, these changes are also omitted. The evalRed instruction switches application of two codes for application of Lam terms:

```
evalRed :: Lam (x → y → z) → Eval r s (z : xs) n a → Eval r s (y : x : xs) n a
evalRed f k ctx γ{..} = let QCons y (QCons x xs) = moore
                        in k ctx (γ {moore = QCons ( App (App f x) y ) xs})
```

In contrast, at the boundaries to dynamic code, the Lam values are collapsed down to their code form. This is the latest possible moment for the translation. This is clear in evalRet and evalCase:

```
evalRet ctx γ{..} = let QCons x QNil = moore in ⟦$retCont $( normaliseGen  x) $input⟧
evalCase left right ctx γ{..} = let QCons exy xs = moore in
  ⟦ case $( normaliseGen  exy) of
     Left x   → $(left  ctx (γ {moore = QCons ( Var  ⟦x⟧) xs}))
     Right y → $(right ctx (γ {moore = QCons ( Var  ⟦y⟧) xs}))⟧
```

In both cases, the normaliseGen function is called to create the code for the normalised Lam term. The evalJoin function undergoes similar changes to evalRet but is omitted. Similarly, the evalPut and evalMake undergo similar changes to evalCase. More interesting are the evalSat changes, which naïvely are as follows:

```
evalSat :: CharPred → Eval r s (Char : xs) (Succ n) a → Eval r s xs (Succ n) a
evalSat f k ctx γ{..} = ⟦ case $input of
  c : cs | $( normaliseGen  (charPredToLam f)) c →
         $(k ctx (γ {input = ⟦cs⟧, moore = QCons ( Var  ⟦c⟧) moore}))
  _ → $(evalRaise ctx γ)⟧
```

This version has simply swapped out the $f into a splicing of the normalised generated Lam term for the CharPred. However, this has already missed a trick: the argument c is available, so it can be applied to the Lam and potentially normalised further. For example, suppose the term generated was Abs (const F), then the current implementation would generate the code (λ_ → False) c. However, using $(normaliseGen (App (charPredToLam f) (Var ⟦c⟧))) would allow for β-reduction, resulting in the generated code False. This can be taken further, however, since a guard that always returns False can be pruned away. Instead, the entire branch can be rendered using Lam:

```
evalSat :: CharPred → Eval r s (Char : xs) (Succ n) a → Eval r s xs (Succ n) a
evalSat f k ctx γ{..} = ⟦ case $input of
  c : cs → $(normaliseGen (If (App (charPredToLam f) (Var ⟦c⟧))
                              (Var (k ctx (γ {input = ⟦cs⟧, moore = QCons (Var ⟦c⟧) moore})))
```

```
                                        (Var (evalRaise ctx γ))))
    _     → $(evalRaise ctx γ)⟧
```

By using the If syntax from Lam, the evaluation of the predicate can direct whether the code will be generated since neither If T nor If F are in normal form (§6.1.2). The fact that the code for a dead branch is not even generated when it is not required by the evaluator has been referred to as *staged fusion* [Willis, Wu, and Schrijvers 2022]. Again, this is worthwhile to do since it is straightforward and may help GHC avoid some work.

Summary

This section introduced three new pieces of machinery: Defunc, Lam, and CharPred. The Lam type is used in the backend machinery to optimise terms to normal form before they are spliced into the generated code; CharPred provides a statically inspectable representation of character predicates for use in analysis based on the RangeSet type built in CHAPTER 5: OPTIMISING FOR SEMI-CONTIGUOUS DATA; Defunc allows for the static inspection of in-library values used to build the composite combinators. Already, these have demonstrated a tangible benefit by allowing for dead-code elimination in the evalSat function, which will simplify the item combinator.

6.2 Basic Optimisations

The following section expands on work from "Staged Selective Parser Combinators" [Willis, Wu, and Pickering 2020] (in collaboration with Nicolas Wu and Matthew Pickering).

There is a collection of low-hanging fruit to be picked for optimisation within the existing `parsley` infrastructure. These are relatively cheap to perform and simple to implement and can be done across applied the high-level Combinator tree and low-level abstract machine Instr. Some of these optimisations have clear benefits in performing domain-specific refinements unavailable to GHC itself, whereas others just help reduce either the size of the trees, or the size of the generated code. As mentioned in §6.1, reducing the overall code size and simplifying its structure will allow GHC to focus on more important optimisations. This section is split into two sections: one discussing high-level transformations performed on Combinator (§6.2.1), and another discussing low-level code generation and control-flow simplifications performed during the translation to Instr (§6.2.2).

6.2.1 High-Level Optimisations

The laws that govern parser combinators demonstrated in figs. 2.1 to 2.5 as well as eqs. (2.21) to (2.26) and (4.5) often form a natural optimisation in one direction. These are not necessarily something that can be exploited by a general-purpose compiler, because even if it were made aware of the existence of the laws, it cannot guarantee that an implementation always adheres to them. As a domain-specific compiler, however, `parsley` can leverage these laws to simplify the parsers provided by a user into a more optimised form. This can be done very simply by inspection – some laws require Defunc values (§6.1.1) to be able to recognise. The function optimise shows a selection of the optimisations that are performed:

```
optimise :: Combinator (Fix Combinator) a → Parsley a
  -- eq. (2.1): fmap id u = u
optimise (In (Pure ID) :<*>: u)                       = u
  -- eq. (2.21): satisfy f <|> satisfy g = satisfy (λc → f c ∨ g c)
optimise (In (Satisfy f) :<|>: In (Satisfy g))        = In (Satisfy (orPred f g))
  -- eq. (2.35): look_ (atomic p) = look_ p
optimise (NegLook (In (Atomic p)))                    = optimise (NegLook p)
  -- eq. (4.5): write r (read r) = pure ()
optimise (Write σ1 (Read σ2)) | Just Refl ← σ1 `geq` σ2 = In (Pure (LIFTED ()))
optimise p                                            = In p
```

Each rule encodes a different law as an optimisation. The first optimisation shows how Defunc values can be inspected, in this case allowing for the removal of redundant mapping operations. Given that Lam's machinery will perform this optimisation by normalisation later, it might seem as if this is redundant, however, it facilitates the optimisation of something like modify_ ID r, which would invoke the last optimisation above, allowing for the removal of two reference operations – it is too late to do this when Lam normalises the term. The second optimisation shows how the static information in CharPred can be combined, which is a domain-specific optimisation that GHC may not recognise, and may aid further analysis later on, as well as generating a much simpler structure in the final code. The third optimisation demonstrates how the application of some laws may reveal new optimisations in the underlying parser, which may require optimise to be repeated. The fourth optimisation again demonstrates a domain-specific optimisation, in this case one that GHC would not perform under any circumstances, as it will not optimise away calls to writeSTRef for obvious reasons.

The optimise function can be used in the algebra for *let-insertion* (§3.3.2) by replacing any calls to isequence p with imap optimise (isequence p). In practice, further optimisations would be performed by optimise, matching most of the laws outlined previously, but these do not add much to the discussion.

6.2.2 Low-Level Optimisations

At the other end of the pipeline, there are simplifications that can be easily performed in the code generator, compile, that allows for slightly better instruction sequences, which result in better generated code.

Instruction Selection

The code generation described throughout CHAPTER 3: EMBEDDING A PARSER COMBINATOR LIBRARY has been simplistic in the sense that it only examines a single layer of the Combinator tree – this is evident from the use of cata. While it is not particularly common, some trees can be optimised into better sequences of instructions than the direct translation currently performed. The most obvious example of this is the construction atomic p <|> q, which would currently generate the following partial machine (excluding join points):

```
λk → catch (catch (p (commit (commit k))) (seek raise)) (tell (same (q k)))
```

Instead, this is behaviourally the same as the original PEG handler, which could be generated instead:

```
λk → catch (p (commit k)) (seek (q k))
```

The overhead difference between the pair of handlers and this new one is evident if you consider that each catch will introduce a new let-bound function, which means that two let-bindings and a conditional are now just one simpler let-binding. Applying this optimisation, however, requires a deeper inspection of the Combinator tree than cata allows, which is where histo (§2.6.1) becomes useful:

```
type CodeGen x = ∀xs n a.Fix Instr (x : xs) (Succ n) a → State Word64 (Fix Instr xs (Succ n) a)
compile p = evalState ( histo alg  p ret) 0
  where alg :: Combinator (Cofree Combinator CodeGen) x → CodeGen x
        alg x = fromMaybe (comp (imap extract x)) (opt x)

        opt :: ∀x.Combinator (Cofree Combinator CodeGen) x → Maybe (CodeGen x)
        opt ((_ ◂ Atomic (p ◂ _)) :<|>: (q ◂ _)) = Just $ λk →
          do φ ← freshPhi @x
             liftM2 (mkJoin φ k · catch) (p (commit (join φ))) (liftM seek (q (join φ)))
        opt _ = Nothing

        comp ...
```

The cata used at the top-level of compile is replaced by histo, and the algebra is adjusted to be the composition of the unchanged comp algebra and the opt algebra. The composition first tries to apply opt on the full history of the code generation: if it returns a Just, then an optimisable pattern was found and this is the desired result, otherwise Nothing was returned, and the history is discarded with imap extract and the regular comp algebra is used. The opt algebra has access to the structure of the original parser where each combinator is paired with the code generation function that arises from that node: looking to the left of a (◂) yields the code generation function that arose from the compilation of the combinator on the right. To handle the atomic p <|> q shape, opt looks first for a :<|>: combinator, then looks to the right of the left argument into the history to see if an Atomic node was present; if so, the code generator for the Atomic's argument p is extracted and combined with q found on the right of (:<|>:) using the specialised handler outlined above.

Control-Flow Simplifications

Join-points (§3.2.4) were introduced to prevent code explosion from the same continuation being spliced into the generated code multiple times. This is done by let-binding the continuation manually and calling these bindings instead. However, some shapes of join points are redundant and can be simply optimised out by inlining:

```
let φ₁ x input = φ₂ x input in ...
let φ₁ x input = retCont x input in ...
let φ₁ _ input = nt₂ input handler retCont in ...
```

The first example is a join point that directly calls another join point, the second a join point that just returns, and the third is a *tail-call* to the non-terminal nt_2, where the result of the join-point is ignored. A *tail-call* is characterised as a call to a non-terminal where the return continuation is not altered. Each of these patterns can be identified by inspecting the continuation passed to MkJoin and optimising it if one of the patterns is identified. Instead of calling mkJoin and join functions directly in the compile function, a new function is created:

```
makeΦ k
    | elidable k  = return (id, k)
    | otherwise = do φ ← freshPhi; return (mkJoin φ k, join φ)
```

If a continuation is elidable (explained below) then do not bother to generate a binding and return the continuation k directly; otherwise generate a binding as normal and return the binder and join-point. This can be used in the comp and opt algebras to generate the join points instead:

```
opt ((_ ◂ Atomic (p ◂ _)) :⟨|⟩: (q ◂ _)) = Just $ λk →
    do binder, φ ← makeΦ k
        liftM2 (binder · catch) (p (commit φ)) (fmap seek (q φ))
```

With this change (also applied to the relevant parts of comp), simple enough join-points are elided, simplifying the generated code. A join-point is considered elidable when it follows one of the patterns identified above:

```
elidable (In (Join _))                      = True
elidable (In Ret)                           = True
elidable (In (Pop (In (Call nt (In Ret))))) = True
elidable _                                  = False
```

Note that there are other patterns that could be considered variants of these and can be elided too: in practice, "free" instructions like Pop, Commit, and so on, should be ignored for the purposes of eliding. In fact, elision benefits from full inlining heuristics, but these are omitted since they are not particularly insightful.

Improving Iteration

In §4.2, a combinator called loop was introduced to serve as the prototypical *iterative combinator*:

```
loop :: Parser () → Parser a → Parser a
loop body exit = let go = body *> go <|> exit in go
```

There are two interesting things about this combinator. The first is that the recursive call to go is represented as call $\mathrm{nt}_{\mathrm{go}}$ (commit ret), which is almost tail-recursive modulo handler unwinding; and the second is that it can only return by failing. In fact, if body cannot fail, then this combinator will not terminate (this is a property of PEG parsers). The body must fail at some point, and it can do so by either consuming input or not consuming input. Consider the outcomes of these different failures:

1. If body fails having not consumed input, then exit is performed, and the loop terminates – if exit does.
2. If body fails having consumed input, then the current invocation of go fails, then:
    (a) If this was the outer-most go, loop body exit fails out to the next handler.
    (b) Otherwise, since go failed having consumed input, fail to the next iteration, repeating these steps.

From this, it is clear that one of two things ultimately happens: either the loop fails to the enclosing handler, or it succeeds with exit. This makes the failure handler produced at each iteration idempotent: no matter how many times it is passed through, the outcome is the same. This is interesting, because the handler is not only bound at every iteration of the loop, but successful termination is forced to unwind back through each iteration to discard each of these handlers.

Strictly speaking, the discarding of each iteration's handler is fruitless, since the return continuation does not care about them anyway: call $\mathsf{nt}_{\mathsf{go}}$ (commit ret) can be rewritten to call $\mathsf{nt}_{\mathsf{go}}$ ret, a true tail call, without breaking anything. In fact, commit ret generates the code λx input → retCont x input anyway, witnessing that the commit is fruitless, which can be eta-reduced to the true tail-call anyway. However, the binding of the handler at each iteration incurs some level of overhead, and, on failure of the entire combinator, the failure must cascade needlessly down the entire chain of recursive handlers. Instead, it would be nice to optimise this combinator to remove the inefficiency.

Hoisting handlers Imagine the rough shape of the generated code for loop body exit, supposing body and exit are both let-bound, join-points are elided, and the tail-call for go has been simplified:

```
loop input handler ret =
  let go input handler ret =
    let handler′ input′ = if input ≡ input′ then exit input handler ret else handler input′
    in  body input handler′ (λ_ input′ → go input′ handler′ ret)
  in go input handler ret
```

If the idempotent handler, handler′, were to be hoisted out of the loop, what would the effect be? To see this, lift the handler′ binding one layer out, making sure to explicitly close over the input when lifting handler′ out of go, leaving ret and handler captured, since they are available from loop itself and never change, unlike the input:

```
loop input handler ret =
  let handler′ input input′ = if input ≡ input′ then exit input handler ret else handler input′
      go input handler ret   = body input (handler′ input) (λ_ input′ → go input′ handler ret)
  in  go input handler ret
```

Notice that the ret and handler parameters to go are now not used: both are only called in handler′, which is no longer bound within the scope of the recursive loop itself. As such, they can be removed:

```
loop input handler ret =
  let handler′ input input′ = if input ≡ input′ then exit input handler ret else handler input′
      go input               = body input (handler′ input) (const go)
  in  go input
```

This is simpler than the original code, with a much tighter inner loop: the result is even tidier when body is not let-bound and is instead inlined into the body of go. This is how the loop combinator will be compiled, moving forward, though some new machinery needs to be introduced.

Representing iteration Like most combinators, loop is represented directly as a similarly typed Loop constructor in the Combinator tree. More interesting is the new instruction introduced to represent iteration:

```
Iter :: NT Void → k '[ ] One Void → k (String : xs) n a → Instr xs n a
```

Relating it to the previous example, the first argument to Iter is the name of the go binding, the second is the code for go itself, and the third is the code for the handler handler′. Conspicuously, the type of the return for the binding introduced is Void: this means that the body of the loop must never return "normally", since Void is uninhabited and so no value could ever be provided to return: this allows the ret argument to go to be dropped.

Without adding any more high-level instructions, the comp algebra for compile can be adjusted to compile Loop into Iter assuming that the compile monad stack is augmented to also allow for the generation of fresh NT values – this can be done just like with Φ (§3.2.4), and is omitted:

```
comp (Loop body exit) = λk → do nt_go ← freshNT @x
                                 liftM2 (iter nt_go) (body (pop (call nt_go ret))) (liftM (tell · same) (exit k))
```

This rule simply puts the body, along with a tail call, into the body of iter nt_{go}, and then puts exit followed by the continuation k into the regular tell · same handler.

Staging `Iter` The evaluation and staging of the Iter instruction is much like Catch and the logic for top-level bindings, though iteration is not factored to the top level to allow it to close over the current machine state Γ:

```
evalIter :: NT Void → Eval r s '[ ] One Void → Eval r s (String : xs) n a → Eval r s xs n a
evalIter nt body exit ctx{..} γ{..} =
  ⟦ let handler input input′ = $(exit ctx (γ {input = ⟦input′⟧, moore = QCons (Var ⟦input⟧) moore}))
        loop input = $(body (ctx {nts = DMap.insert nt (NTBound NoRegs ⟦λinp _ _ → loop inp⟧)})
                            (Γ {input = ⟦input⟧, moore = QNil,
                                handlers = Cons ⟦handler input⟧ Nil, retCont = ⟦absurd⟧}))
    in loop $input⟧
```

The structure of the evalIter is like the shape of the transformed definition at the start of this section. The function absurd :: Void → a is provided as the return continuation. The most notable part of this definition is the fact that the binding inserted into the nts map within the body must be a lambda to make the types work. This will produce inefficient code, though GHC will probably optimise it.

Staged Fusion of Continuations Instead of relying on GHC, the same trick from §6.1.4, *staged fusion*, can be used to eliminate these arguments. First, the types of the bindings must be refined:

```
type BaseFunc r s a =
  Code String → Code (String → ST s r) → Code (a → String → ST s r) → Code (ST s r)
type family Func xs r s a where
  Func '[ ]      r s a = BaseFunc r s a
  Func (x : xs) r s a = Code (STRef s x) → Func xs r s a
```

```
data NTBound r s a = ∀rs.NTBound (Regs rs) Func rs r s a

data Γ r s xs n a = Γ {input    :: Code String
                      , moore    :: QList xs
                      , handlers :: Vec n ( Code String → Code (ST s r))
                      , retCont  :: Code a → Code String → Code (ST s r)
                      }
```

Each of the functions in the above types have been broken apart to facilitate staged fusion. However, notice that the arguments to BaseFunc have not been broken up further: this is the true dynamic boundary of these continuations, so they cannot be split up. Obviously, the code will need to be converted to fully dynamic form to invoke a BaseFunc in the Call instruction and made into the static form whenever they are first created:

```
evalIter :: NT Void → Eval r s '[ ] One Void → Eval r s (String : xs) n a → Eval r s xs n a
evalIter nt body exit ctx{..} γ{..} =
  ⟦ let handler input input′ = $(exit ctx (γ {input = ⟦input′⟧, moore = QCons (Var ⟦input⟧) moore}))
        loop input = $(body (ctx {nts = DMap.insert nt (NTBound NoRegs ( λinp _ _ → ⟦loop $inp⟧ ))})
                            (Γ {input = ⟦input⟧, moore = QNil,
                                handlers = Cons ( λinp′ → ⟦handler input $inp′⟧ ) Nil, retCont = noret }))
     in loop $input⟧
   where noret = error "returning from loops is impossible!"
```

This updated evalIter shows the effect of the fusion, where the binding is now a static function ignoring the arguments, which means they will not materialise at runtime. In fact, the return continuation, now noret, will cause an error if it is so much as evaluated, which would imply it was used in a spliced expression. This function also demonstrates how dynamic bindings, like handler, are converted into their static form.

```
evalCall :: NT x → Eval r s (x : xs) (Succ n) a → Eval r s xs (Succ n) a
evalCall nt k ctx{..} γ{..} = applyNTBound (nts DMap. ! nt) regs input h_dyn ret_dyn
   where Cons h _ = handlers
         h_dyn    = etaReduce (dynFunc h)
         ret_dyn  = etaReduce ⟦λx inp′ → $(k ctx (γ {input = ⟦inp′⟧, moore = QCons (Var ⟦x⟧) moore}))⟧
```

On the other hand, evalCall shows the effect of the fusion at the call-site of dynamic bindings: applyNTBound now takes code as arguments and produces code as a result. Here, $\mathsf{h}_{\mathsf{dyn}}$ and $\mathsf{ret}_{\mathsf{dyn}}$ show the explicit construction of dynamic continuations to feed with (dynFunc was defined in §2.4). The etaReduce here has the effect of performing the tail-call simplification of the continuations and eliminating redundant lambdas. Changes are required to evalRet, evalRaise, and evalCatch, as well as letRec, but these offer no additional insights, and are omitted. Join-points do not benefit from staged fusion, and so remain unchanged. The staged fusion does, however, affect the terminal continuations from eval, accept and fail, and optimises away the input argument:

```
accept x _ = ⟦return (Just $x)⟧                    fail _ = ⟦return Nothing⟧
```

Summary

This section introduced optimisations at two different levels in the pipeline: high-level optimisations based on parser laws in §6.2.1, which help reduce the size of the combinator tree and perform some domain-specific optimisations not possible by GHC; and low-level optimisations that focus on improving the generation of the abstract machine instructions in §6.2.2. Notably, the introduction of dedicated support for iteration in the parser brought forward some interesting effects on the whole architecture. By applying *staged fusion*, more aspects of redundant structure can be forgotten at compile-time, leading to simpler code.

6.3 Input Consumption Analysis and Factoring

The following section expands on work from "Staged Selective Parser Combinators" [Willis, Wu, and Pickering 2020] (in collaboration with Nicolas Wu and Matthew Pickering).

Consuming input is the primary dynamic behaviour of a parser and one source of asymptotic slow-down in parser combinators is having to backtrack and consume the same input multiple times. Factoring a grammar to eliminate the re-consumption of input is a common transformation for a parser writer to perform, but it is not always possible to do so in a non-destructive way.

This section aims to augment `parsley`'s existing machinery with analysis and corresponding optimisation that allows for some input factoring to be performed automatically, as well as leveraging the backtracking properties of the parser to make input characters available to the parser ahead of when they will be inspected.

As a first example, consider the generated code for the parser atomic (char 'a') <|> char 'b':

```
aorb inp = runST $ let handler _ = case inp of c : _ → if c ≡ 'b' then return (Just 'b') else return Nothing
                                               [ ]   → return Nothing
                    in  case inp of c : _ → if c ≡ 'a' then return (Just 'a') else handler inp
                                    [ ]   → handler inp
```

While the code for this is quite concise, it pattern matches on the input twice, once in each branch. But since the second branch can only succeed with input available, this read could be factored out ahead:

```
aorb inp = runST $ case inp of c : _ → let handler _ = if c ≡ 'b' then return (Just 'b') else return Nothing
                                        in  if c ≡ 'a' then return (Just 'a') else handler inp
                               [ ]   → return Nothing
```

This is better in the sense that only one pattern match on the input, which is a more expensive operation than character equality, has been performed here: the character is inspected twice, but not wastefully re-consumed.

As another example, consider look_+ (string "123") *> exactly 3 digit. When the positive lookahead is finished, it is clear statically that there are three digits at the start of the input, so the latter three character reads could be considered as "free", not resulting in any code generated at all: this can be manually done by a user as a performance trick.

As a third example, consider atomic (string "abc") <|> atomic (string "ab"), which shares two leading characters: the left-hand side is guaranteed to consume three characters on success, and the right-hand side is guaranteed to consume two. Since atomic (string "abc") is entirely atomic, at most two characters can be factored out in advance. Furthermore, the first two characters of the branches are the same and can be shared, but exceptional care must be taken to not accidentally factor any code that fails having consumed any input past a failure handler's scope: to keep things safe and simple, the a can be shared between the branches, but not the b. The (slightly simplified) code for this parser could be:

```
leading inp = runST $ case inp of
  'a' : c₂ : cs → let handler _ = if c₂ ≡ 'b' then return (Just "ab") else return Nothing
                  in  if c₂ ≡ 'b' then case cs of c₃ : _ → if c₃ ≡ 'c' then return (Just "abc") else handler cs
                                                  [ ]    → handler cs
                                   else handler cs
  _             → return Nothing
```

While this repeats the check for b twice, it will fail fast when not provided with a, and will not re-consume any characters that are not strictly necessary.

Outline The identification of *cut-points* in the parser, used to decide across which boundaries input can be factored, is described in §6.3.1; the process of determining exactly how many characters can be factored across each linear stretch of program is covered in §6.3.2; and the factoring itself is implemented in §6.3.3.

6.3.1 Identifying Cut-Points

Before any factoring can be performed, it is important to understand what limitations there are on factoring throughout a parser. Recall that parsley has a left-biased choice operator (<|>), for which backtracking is only permitted if no input is consumed in the first failing branch; backtracking is explicitly enabled by using the atomic combinator. Care must be taken to respect the semantics of (<|>) outside of the presence of an accompanying atomic combinator: checking for the existence of input may alter whether input is consumed on failure. Consider the following example of an illegal factoring:

```
string "abc" <|> string "def" ⇒ factor 3 *> (string "abc" <|> string "def")
```

For a factoring to be legal, it must preserve the exact same extrinsic behaviour for a given parser fragment. In this case, applying the input "ab" to the unfactored parser fails having consumed input, but applying it to the factored parser will fail factor n, which consumes no input on failure. If, however, there were an atomic around the entire parser, then the factoring does not observably lead to any change in the behaviour. In essence, the parser needs to be annotated with the explicit cut-points, through which backtracking cannot occur: the behaviours of these points must be preserved through any optimisation, and it is only clear how these materialise by considering the scoping of the operations. As such, identifying these cuts should be performed on the high-level Combinator tree.

Representing Cuts

While identifying cuts within the grammar is well-suited to the Combinator tree, further stages of analysis, such as the one described in §6.3.2, are easier to perform in a representation where control flow is more explicit: in this case that means analysis on the Instr datatype instead. Since information about cuts is required for later phases, any information determined now must be preserved within the tree till later. This is the role of a *meta combinator*, which encodes statically determined information about the grammar, but has no equivalent for high-level combinators. This is expressed by the following new AST node:

```
Meta :: MetaCombinator → k a → Combinator k a
data MetaCombinator = Cut
```

When a parser p is tagged with meta Cut, that indicates that a successfully executed p satisfies the requirements for a cut – should one be required – and that no backtracking can occur through it. This is deliberately wider than a single character, as a cut within an atomic only occurs after the atomic is completed, not during.

Finding Cuts

Annotating the parser with cut nodes is performed as a single fold through the parser, with the following carrier:

```
type CutAnalysis a = Bool → (Parsley a, Bool)
```

The boolean parameter represents whether the current combinator being inspected is under the scope of an atomic or not. The values returned are the annotated parser as well as whether this combinator and its sub-parsers performs a cut or not, regardless of whether it was annotated with a cut.

```
cutAnalysis :: Parsley a → Parsley a
cutAnalysis p = fst (cata cutAlg p False)
```

The cutAnalysis function, which will be called for every non-terminal in the grammar, as well as the top-level parser, provides False to the result of the cata to indicate that this non-terminal is conservatively not atomic. The algebra, cutAlg, is defined as follows:

```
cutAlg :: Combinator CutAnalysis a → CutAnalysis a
cutAlg (Pure x) _ = (pure x, False)    -- similarly, for GetReg, Let, and Empty
```

The Pure case, along with GetReg, Let, and Empty, all simply rebuild the tree, ignoring whether backtracking is required, and all do not cut. As mentioned above, the behaviour of any given non-terminal is not known, and so is conservatively treated as doing nothing. The other three combinators are all pure with respect to input consumption, and so all do nothing as well.

```
cutAlg (Satisfy f) underAtomic
    | underAtomic = (satisfy f, True)
    | otherwise   = (meta Cut (satisfy f), True)
```

In the satisfy case, a cut is always performed, so True is returned, but a cut node is only inserted if no backtracking is required: if backtracking is required, then there is something that is assumed to backtrack above, which means the generation of the cut is left to this parent node. One such possible parent is atomic:

```
cutAlg (Atomic p) underAtomic = let (p′, cuts) = p True
                                in if cuts ∧ not underAtomic then (meta Cut (atomic p′), cuts)
                                   else (atomic p′, cuts)
```

Here, atomic always passes True to the child parser; however, depending on whether a cut is required by the parent (i.e., not underAtomic) then the atomic itself can generate a cut if its child is known to have cut-like behaviour. The cut occurs after the atomic, since failing within an atomic consumes no input. On the other hand, both NegLook and Look ignore the cutting properties of their argument parser:

```
cutAlg (Look p)      underAtomic = let (p′, _) = p underAtomic in (look p′, False)
cutAlg (NegLook p) _             = let (p′, _) = p True in (look_ p′, False)
```

For Look, the parser p is allowed to cut if it wishes, but this does not affect the overall parser, which may have to cut after the lookahead: this is because input consumed after a lookahead is rolled back, so the cut is only local to the lookahead itself. Similarly, this is the case with NegLook too, except that never consumes input and so is always atomic. All four of (:<*>:), (:<*:), (:*>:), and MakeReg are implemented in terms of seqCut:

```
seqCut con underAtomic p q = let (p′, pcuts) = p underAtomic
                                 (q′, qcuts) = q (underAtomic ∨ pcuts)
                             in  (In (con p′ q′), pcuts ∨ qcuts)
```

In these cases, if the first parser cuts, then the second is allowed to assume it is atomic since any factoring that happens after a cut cannot possibly alter the backtracking characteristics of the overall parser. A sequencing of two parsers cuts if either of its sub-parsers cut. This is similar for Branch:

```
cutAlg (Branch b p q) underAtomic = let (b′, bcuts) = b underAtomic
                                        (p′, pcuts) = p (underAtomic ∨ bcuts)
                                        (q′, qcuts) = q (underAtomic ∨ bcuts)
                                    in  (branch b′ p′ q′, bcuts ∨ (pcuts ∧ qcuts))
```

As opposed to seqCut, however, the two possible branches p and q must be considered independently and the overall combinator only cuts when both p and q cut: analysis on selective operations must always under- or over-approximate the possible effects. In this case, the more conservative option is to demand they both cut.

```
cutAlg (PutReg σ p) underAtomic = first (putReg σ) (p underAtomic)
cutAlg (Meta m p)   underAtomic = first (meta m) (p underAtomic)
```

The PutReg and Meta combinators are both non-interesting, and just pass through their sub-parser's results unaltered. The PutReg combinator has more interesting interactions and implications in other parts of the analysis. Finally, the (:<|>:) and Loop combinators are similar to one another:

```
cutAlg (p :<|>: q)           underAtomic = let (p′, pcuts) = p False
                                               (q′, qcuts) = q underAtomic
                                           in  (p′ <|> q′, pcuts ∧ qcuts)
cutAlg (Loop body exit) underAtomic = let (body′, _) = body False
                                          (exit′, cuts) = exit underAtomic
                                      in  (loop body′ exit′, cuts)
```

In both cases, False is fed to the left branch (for loops this is the body): this is because the right branch can only be accessed if the left branch fails without consuming input – this in turn requires some form of cut. Like Branch, (:<|>:) requires both branches cut to make the whole combinator cut; however, looping combinators are guaranteed to pass through exit, whereas body is not guaranteed to be executed, so the information can be refined to just whether exit cuts.

6.3.2 Calculating Amount of Factored Input

With cut-points identified, it remains to figure out how many characters can be factored out in each part of the parser. However, calculating the input consumed along a given path through the parser is more involved on the Combinator tree because the control flow is not explicit. As such, this part of the analysis is much more natural to perform on the Instr type instead.

Information about cut points was preserved in the Combinator tree as part of the Meta node: this will also need to be done for Instr, as the information attained now will be needed during the staged evaluation in §6.3.3. As such, an overloaded version of Meta will be used to represent meta-instructions within the Instr datatype: these will control input factoring during the staging. There are three main factoring types:

- Regular factoring of input allows for the existence of *n* characters to be checked as some point.
- After a positive lookahead of size *n* has been performed, *n* characters have already been read: these do not require any existence check and can be reused directly in the rest of the parser.
- Input can be factored across a join point if all callers of the join point perform the necessary existence checks – this can then cascade and continue to factor with other characters on those paths.

The first two are represented by a single instruction each: Credit and Refund respectively. The latter case is represented by two instructions: Liquidate for the callers, and Bursary for the callee. A fifth instruction is also introduced to model Cut at this level, called Fence: during analysis, nothing is allowed to cross a Fence instruction. The semantics of the first four instruction will be discussed later, though Fence does nothing at stage-time. The metaphor here, however, is that a parser is given credit for characters it may read later for free (so long as it does pay up-front!), can receive refunds for characters that have been rolled back, may be required to dump all its credit in one do to pay for something more expensive, or may receive a bursary that it does not have to pay for. Their types are as follows (with an index for the number of handlers required):

```
data MetaInstr n where
  Credit    :: Coins → MetaInstr (Succ n)    -- can fail!
  Refund    :: Int → MetaInstr n
  Liquidate :: Int → MetaInstr (Succ n)        -- can fail!
  Bursary   :: Int → MetaInstr n
  Fence     :: Bool → MetaInstr n
```

In particular, Credit is interested in three different components, rendered as the datatype Coins:

```
data Coins = Coins {checksExistenceOf :: Int, willConsume :: Int, firstPred :: Maybe CharPred}
```

The first component checksExistenceOf, is the total number of characters that are known to be required for the parser to succeed; willConsume, which is possibly less than checksExistenceOf, is the number of characters that the path will directly eat (and so these characters may be cached up-front); and firstPred is the character predicate associated with the first character read of this bunch, should it be known. The reason willConsume may be less than checksExistenceOf is because the difference may be used to pay for a call to a join-point, which does not need caching as these bindings would be out-of-scope. The firstPred is useful since it is possible to always check the first character against this predicate without failing having consumed input – this can help fail faster and reduce ambiguity in the grammar. It is possible to extract more predicates, however it becomes trickier to ensure that they are not factored so that they are outside of a failure handler, which changes behaviour; this is a conservative, but safe, factoring instead.

Counting `Coins`

The first part of the low-level analysis is to be able to calculate the Coins along a specific set of instructions. This is a simple cata along the Instr datatype:

```
coins :: Bool → Fix Instr xs n a → Coins
coins fenceEnabled = fst · cata coinsAlg
  where coinsAlg :: Instr ( {-Const -} (Coins, Bool)) xs n a → (Coins, Bool)
```

Here, the fenceEnabled flag determines whether the Fence meta-instruction has any effect. It will be useful to disable it when considering what happens inside a lookahead (if the boolean flag on the Fence is also False), but everywhere else it will be enabled. The Bool returned by coinsAlg indicates whether the branch ends in unconditional failure or not – this is useful for identifying failure handlers like atomic's Seek Empt.

```
coinsAlg Ret = (Coins 0 0 Nothing, False)
coinsAlg Empt = (Coins 0 0 Nothing, True)
```

The rule for Ret – along with Call, Iter, and Join – just returns no coins and is not a failure. Similarly, Empt returns no coins, but is a failure. Happily, the following instructions all have no effect on the analysis and just return their argument unchanged: Push, Pop, Red, Commit, Tell, Seek, Swap, Make, Get, and Put. Similarly, the rule for MkJoin is:

```
coinsAlg (MkJoin _ _ k) = k
```

This ignores the information about the binding and just transitively returns the scoped portion: the binding will affect the system via the Liquidate instruction later. The Sat instruction is the first interesting one:

```
coinsAlg (Sat p (Coins n m _, fails)) = (Coins (n + 1) (m + 1) (Just p), fails)
```

For Sat, one coin is added to both totals, and the first predicate is set to whatever the Sat matches, ignoring the previous one. This then interacts with Catch and Case, which share a helper function splitCoins:

```
splitCoins :: (Coins, Bool) → (Coins, Bool) → (Coins, Bool)
splitCoins k                     (Coins 0 _ _, True)  = k
splitCoins (Coins 0 _ _, True)   k                    = k
splitCoins (Coins n₁ m₁ p₁, _) (Coins n₂ m₂ p₂, _) = (Coins (n₁ ⊓ n₂) (m₁ ⊓ m₂) (mergePreds p₁ p₂), False)
```

If either of the two branches just fails, then the other branch is just taken as is: this helps avoid a minimum with 0 in the last case, which would remove all benefit of factoring from atomic and filterS. In the last case, however, the minimum is taken from both branches (which would be the characters they have in common) and predicates are merged; the definition of this is omitted for brevity, but it simply combines the predicates, if they are statically known, by checking if one is a subset of the other: if this is true, the larger of the two sets is picked, otherwise Nothing is returned. The final rules of the algebra concern the meta-instructions:

```
coinsAlg (Meta (Credit c) k)                     = k
coinsAlg (Meta (Refund c) (Coins n m _, fails)) = (Coins ((n − c) ⊔ 0) ((m − c) ⊔ 0) Nothing, fails)
coinsAlg (Meta (Liquidate c) _)                 = (Coins c 0 Nothing, False)
coinsAlg (Meta (Bursary _) _)                   = (Coins 0 0 Nothing, False)
coinsAlg (Meta (Fence strong) (c, fails))
    | fenceEnabled, not strong = (Coins 0 0 Nothing, fails)
    | otherwise                = (c, fails)
```

The coins on Credit do nothing as they are reflective of the continuation k anyway; Refund will subtract from both coin counts and remove the known predicate, leaving the remaining required inputs; Liquidate demands c coins, but not any that will be consumed immediately; Bursary does not need to propagate anything forward, as it is always at the front of a binding after which the calculations have already been done; and Fence erases all information from the continuation if it is enabled or is strong and does nothing otherwise.

This is the entire algebra for computing the Coins required for a given stretch of instructions. This is used, in conjunction with the meta-instructions themselves, within the code generation.

Generating Meta-Instructions

The coins function developed in the previous section is used by compile to introduce the meta-instructions into the generated Instr sequence. The most common operation performed is adding credit, which can be done via the following helper function:

```
giveCredit p = meta (Credit (coins True p)) p
```

Given a sequence of instructions p, giveCredit first computes the coins required for p and then adds them as credit.

Several of the existing rules in comp (and opt, but this is omitted for brevity, as it is much the same) need to be modified to interact with the meta-instructions. Those rules not listed below are unchanged, however:

```
comp (p :<|>: q)          = λk → do (binder, φ) ← makeΦ k
                                    liftM2 (binder · catch)
                                           ( liftM giveCredit (p (commit φ)))
                                           (liftM (tell · same · giveCredit ) (q φ))
comp (Branch b l r)       = λk → do (binder, φ) ← makeΦ k
                                    let k′ = red ⟦flip ($)⟧ φ
                                    liftM binder (b ≪ liftM2 kase ( giveCredit (l k′)) ( giveCredit (r k′)))
comp (Loop body exit) = λk → do nt_go ← freshNT @x
                                    bodyc ← body (pop (call nt_go ret))
                                    exitc ← exit k
                                    return (iter nt_go ( giveCredit bodyc) (tell (same ( giveCredit exitc))))
```

When considering any one of (:<|>:), Branch, or Loop, both branches of the parser are given their credit individually. When possible, some of this may have been shared between the branches, but this will be factored earlier in the instructions – the semantics of Credit will account for partial-credit later.

```
comp (PutReg σ p)  = p · put σ · push () · meta (Fence False)
comp (Meta Cut p) = p · meta (Fence False) · giveCredit
```

Both PutReg and Meta Cut must generate a Fence to ensure that factoring cannot commute through them: recall that factoring through a Put may prevent it from being executed. In the Cut case, however, it is useful to give credit for the remainder of the branch since cuts are guaranteed to appear after at least one piece of input was consumed along a branch.

```
comp (Look p)       = λk → do n ← liftM (checksExistenceOf · coins False) (p ret)
                              liftM tell (p (swap (seek meta (Refund n) k)))
comp (NegLook p) = λk → do pc ← p (pop (seek (commit raise)))
                              return (catch meta (Fence True) ( giveCredit (tell pc)) (seek (push () k)))
```

The two lookahead combinators are also adjusted. The change to Look is a little odd in that it now generates p twice, once with a continuation of ret and the other with its regular continuation. This is done so that the input required for p alone can be analysed, which is then refunded before progressing to the regular continuation k. For NegLook, credit can be given to pc, but this must not be factored further out as it is negative lookahead – this is ensured by another use of Fence, this time with its flag set to True so that it cannot be disabled by lookahead.

```
makeΦ k
   | elidable k  = return (id, k)
   | otherwise = do φ ← freshPhi
```

```
let n = checksExistenceOf (coins True k)
return (mkJoin φ (meta (Bursary n) k), meta (Liquidate n) (join φ))
```

Finally, the construction of join points via makeΦ is altered to incorporate the Bursary and Liquidate meta-instructions: bursaries are offered to the binding, and liquidation is demanded before a call.

This concludes all the changes needed to the high-level compiler, and the remaining work of factoring falls to the implementation of these meta-instructions, as well as some changes to Sat, in the staging, covered in §6.3.3.

6.3.3 Performing Factoring

Most of the logic of input factoring at the evaluation level is about the manipulation of static information to avoid generating dynamic code. Static information is kept in the Ctx, not Γ, and this is where information about current and future credit will be kept, as well as pre-fetched characters. It is augmented as follows:

```
data Input = Input {str :: Code String}
data Ctx r s = Ctx {nts        :: DMap NT (NTBound r s), phis :: DMap Φ (PhiBound r s)
                   , regs      :: DMap SigmaVar (QSTRef s)
                   , credit    :: Int, futureCredit :: Queue Coins
                   , netWorth  :: Int, knownChars :: RewindableQueue (CharPred, Code Char, Input)
                   }
```

The credit field is the current number of characters that have been accounted for already, and so can be consumed "for free". The futureCredit is a queue of coins added from Credit instructions but those that have not yet materialised into confirmed credit: this is used when there is still credit available during a Credit instruction. The netWorth field is the sum of the credit and all futureCredit. The knownChars queue contains characters, their corresponding CharPreds if they have been checked already, and the residual input from that point; the queue is rewindable, which means that items popped off the front are kept and can be rewound back onto the queue – this is used for refunding. The type Input is – currently – a simple wrapper around a Code String value: assume that this has been used in Γ too, along with Seek and Tell and so on. The implementation of Queue and RewindableQueue is not important. These new components all start at 0 or empty queues: when creating a new binding like a join-point or a loop, these must be reset back to these default values. There are a handful of helpful functions to worth with these values:

```
giveCredit :: Int → Ctx r s → Ctx r s
giveCredit c ctx   = ctx {credit = credit ctx + c, netWorth = netWorth ctx + c}

refundCredit :: Int → Ctx r s → Ctx r s
refundCredit c ctx = giveCredit c ctx {knownChars = RewindableQueue.rewind c (knownChars ctx)}

storeCredit :: Coins → Ctx r s → Ctx r s
storeCredit c ctx  = ctx {futureCredit = Queue.enqueue c (futureCredit ctx)
                         , netWorth    = netWorth ctx + checksExistenceOf c
                         }
```

```
redeemCredit :: Ctx r s → (Coins, Ctx r s)
redeemCredit ctx = (c, ctx {futureCredit = futureCredit′, netWorth = netWorth ctx − checksExistenceOf c})
  where (c, futureCredit′) = Queue.dequeue (futureCredit ctx)
```

The first, giveCredit adds a set number of free character consumptions onto the current credit, and updates the netWorth to match; refundCredit additionally rewinds the queue of known characters so that these characters can be read again statically without inspecting the input again; storeCredit pushes a Coins value onto the end of the queue so that it can be used later, and also updates the net worth to suit – these values have not been guaranteed to exist yet. On the other hand, redeemCredit pops the first credit off the queue and returns it, removing that value from the net worth. There are also two more helper functions that interact with the knownChars:

```
addChar :: CharPred → Code Char → Input → Ctx r s → Ctx r s
addChar p qc qcs ctx = ctx {knownChars = RewindableQueue.enqeue (p, qc, qcs) (knownChars ctx)}

item :: CharPred
item = Ranges (RangeSet.complement RangeSet.empty)

readChar  ::  Ctx r s → CharPred → ((Code Char → Input → Code b) → Code b)
          → (Code Char → CharPred → Input → Ctx r s → Code b) → Code b
readChar ctx{..} pred fallback k
   | credit ≡ 0                       = error "there must be credit to read a character"
   | RewindableQueue.null knownChars = fallback $ λqc input → pullChar (addChar item qc input ctx)
   | otherwise                        = pullChar ctx
  where pullChar ctx =
          let optimisePred (pred_old, qc, input) = (andPred pred_old pred, qc, input)
              knownChars′ = RewindableQueue.poke optimisePred (knownChars ctx)
              ((pred_opt, qc, input), knownChars″) = RewindableQueue.dequeue knownChars′
              ctx′ = ctx {credit = credit − 1, netWorth = netWorth − 1, knownChars = knownChars″}
          in  k qc pred_opt input ctx′
```

The function addChar enqueues a dynamic character, along with the predicate it has been checked against and the residual input from that point, onto the knownChars queue. Given that this works with dynamic values, it implies that the code to consume the character has been generated and this is called within a continuation of that point. On the other hand, the readChar function allows for a character to be pulled from the queue and returned: some credit is required to do this. If there is a character in the queue, this is dequeued using pullChar, also spending credit in the process. Otherwise, if the queue is empty, which may happen if characters are known to exist but not statically available, the provided fallback function can be used to consume the next token from the input, which is guaranteed to exist, and enqueues it on before using pullChar as before. The character is enqueued in the fallback case so that it can be rewound if necessary. Two different predicates are supplied to the continuation k: the first is the original predicate that was registered for this character, and the second is their intersection. This is done by poking the queue before a dequeue, as this ensures that the refined predicate will be present in the rewind portion of the queue if lookahead occurs.

A new consumption model The other new piece of machinery is a more fine-grained way of consuming characters in bulk. This forms the backbone of an actual factoring. This is implemented by the following function:

```
check :: Int → Int → Code String → Maybe (Code Char → Code a → Code a)
      → (Code String → [(Code Char, Code String)] → Code a) → Code a → Code a
check n m qcs headCheck good bad = go n m qcs mempty headCheck
  where
    go   :: Int → Int → Code String → DList (Code Char, Code String)
         → Maybe (Code Char → Code a → Code a) → Code a
    go 0 _   qcs dcs _         = good qcs (fromCayley dcs)
    go n 0   qcs dcs _         = ⟦ case drop (n − 1) $qcs of [ ] → $bad
                                                             cs  → $good ⟦cs⟧ (fromCayley dcs)⟧
    go n m   qcs dcs headCheck = ⟦ case $qcs of c : cs → $(good′ ⟦c⟧ ⟦cs⟧ headCheck)
                                                [ ]    → $bad⟧
      where
        good′ qc qcs Nothing      = go (n − 1) (m − 1) qcs (diffSnoc (qc, qcs) dcs) Nothing
        good′ qc qcs (Just check) = check qc (go (n − 1) (m − 1) qcs (diffSnoc (qc, qcs) dcs) Nothing)
```

At a high-level, check n m qcs headCheck good bad means the following: a total of n characters need to be verified that they exist, with m of those actually consumed and collected up; qcs is the starting point from which to start reading; headCheck is an optional check to be applied to the very first character consumed only; and good is the continuation to perform when the read of n characters was successful and is provided with the input at that final point as well as the m consumed characters and their corresponding residual inputs too, with bad being the continuation to use if the characters could not be consumed. The function works in term of the helper go, which is inductive on n and m and threads through a difference list of the currently consumed characters: if n is zero, the good continuation should be called with the current input and the finalised difference list; if m is zero then there are still characters to check but these do not need to be explicitly consumed, so the remainder are handled using drop, which will generate less code and a tighter loop; otherwise consume a single character, add it onto the difference list, and continue generating more code if the "uncons" succeeds. When the very first character is consumed in this way, the function within headCheck, should one exist, is applied to that character: in practice this may be the application of some character predicate, allowing the factoring of the leading character.

Implementing `Meta`

With the context adjusted and the new check function in place, it is possible to implement the evaluation semantics for the meta-instructions. The simplest instructions are as follows:

```
evalMeta :: MetaInstr n → Eval r s xs n a → Eval r s xs n a
evalMeta (Fence _)    k ctx γ = k ctx γ
evalMeta (Refund n)   k ctx γ = k (refundCredit n ctx) γ
evalMeta (Bursary n)  k ctx γ = k (giveCredit n ctx) γ
```

As expected, Fence does absolutely nothing. More interestingly, Refund simply calls refundCredit, which has the effect of adding n characters of "free consumption" and rewinds the queue to put those characters at the front of the queue. Simply, Bursary directly adds n characters to the credit value: these are known to exist, but they are not statically available in this branch: they will be consumed in due course.

The Liquidate instruction is trickier: some characters may already have been checked, with the potential that more need to be read. In this case, it is necessary to try and read the rest of the characters from the deepest confirmed input location. In the check function from earlier, this would have been one of the arguments fed to the final good continuation. As such, it is a good idea to track this during the staging: this necessitates a change to the Input type introduced above:

```
data Input = Input {str :: Code String, deepestKnown :: Code String}
```

Here, the deepestKnown component records the furthest place in the input where it is known that a character can be consumed. This will start off as Nothing and will be updated whenever possible in a function defined shortly. The Liquidate evaluation can be defined as follows:

```
evalMeta (Liquidate n) k ctx γ
   | credit ctx ⩾ n = k ctx γ
   | credit ctx ≡ 0  = checkAndRead n 0 Nothing (input γ) str good bad
   | otherwise       = checkAndRead (n − credit ctx + 1) 0 Nothing (input γ) (fromJust · deepestKnown)
                                         good bad
  where bad             = evalRaise ctx γ
        good input′ _ = k ctx (γ {input = input′})
```

If there is enough credit to pay for the n characters now, nothing needs to be done. Otherwise, if there is no credit at all then the checkAndRead function will check for the existence of all n characters starting at the current input with str. Finally, if there is some credit remaining, this is used up immediately, and the leftover demand is read from the deepestKnown point (which is guaranteed to exist here). The checkAndRead function is as follows:

```
checkAndRead :: Int → Int → Maybe CharPred → Input → (Input → Code String)
                → (Input → [(Code Char, Input)] → Code a) → Code a → Code a
checkAndRead n m headCheck input sel good bad =
  flip (check n m (sel input) headCheck′) bad $ λdeepest qccs →
    let input′ = input {deepestKnown = Just deepest}
    in good input′ (map (λ(qc, qcs) → (qc, input′ {str = qcs})) qccs)
  where headCheck′ = fmap checkFor headCheck
        checkFor f qc good = normaliseGen (If (App (charPredToLam f) (Var qc)) (Var good) (Var bad))
```

The checkAndRead function interacts with the check function defined earlier to form the meat of the factoring. The n and m parameters here serve the same purpose as they did in check, with headCheck now being a Maybe CharPred: this is transformed into the expected function for check by mapping checkFor over it. The checkFor function reuses the logic previously used for evalSat in §6.1.4: this is used to validate the very first character consumed against a known common predicate for all the possible branches as established by the analysis in §6.3.2. The good continuation for check is altered to adapt it to work with Input instead of just Code String: this is where the deepestKnown input is updated, and this knowledge is shared with all the characters cached by the function. Of course, the Liquidate meta-instruction never caches any characters – and so ignores the list – but Credit certainly does:

```
evalMeta (Credit c) k ctx γ
   | netWorth ctx ≢ 0 = k (storeCredit (c `minus` netWorth ctx) ctx) γ
   | otherwise          = checkReadAndCache c k ctx γ
  where minus (Coins n m _) net = Coins ((n − net) ⊔ 0) ((m − net) ⊔ 0) Nothing
```

Here, Credit must check whether there is any credit, unredeemed or otherwise, left in the context. If there is, then the processing of the coins c will be delayed until later by using storeCredit. Notably, the current net worth is deducted from this credit before it is stored, since this overlap would have already been factored during analysis. Otherwise, checkReadAndCache is used to complete the logic:

```
checkReadAndCache :: Coins → Eval r s xs (Succ n) a → Eval r s xs (Succ n) a
checkReadAndCache Coins{..} k ctx γ =
  checkAndRead checksExistenceOf willConsume firstPred (input γ) str good (evalRaise ctx γ)
  where onlyStatic (UserPred _) = Nothing
        onlyStatic p              = Just p
        preds = maybe id (:) firstPred (repeat item)
        good input′ cached = foldr (λ((c, input), pred) k ctx′ → k (addChar pred c input ctx′))
                                   (λctx′ → k (giveCredit checksExistenceOf ctx′) (γ {input = input′}))
                                   (zip cached preds)
                                   ctx
```

This final helper function differs slightly from checkAndRead in that it needs to process the resulting characters more carefully, loading them into the context ready to be used later. Firstly, the three components of the Coins type are passed to checkAndRead: checksExistenceOf, willConsume, and firstPred. The input from γ is fed into it too, with str being used to pick which of str and deepestKnown should be used. If anything should go wrong with the input consumption, evalRaise is used to fail with the initial state γ: this means checkReadAndCache will always fail having not consumed input. In the good case, however, the cached characters are first zipped with a list of predicates (the first is the firstPred, should it exist, then the rest are all equivalent to const True) and this is folded, which each in turn being added into the context with addChar. Finally at the base case of this fold, the continuation k is invoked with the updated context given the right credit and the updated γ, which is aware of the deepest consumed input. The result of this operation is that all the characters that were statically collected during check will now be present in the knownChars queue within the context. This will (finally) come into play during the evaluation of the Sat instruction.

Adjusting Sat

The new version of evalSat has a bit of added complexity to handle the static interactions with the context set up by the meta-instructions. There are three cases to consider: where there is no credit remaining, unredeemed or otherwise; where there is credit currently available; where no credit is available, but some may be redeemed from the futureCredit queue. These are implemented as follows (this is a new implementation, so no highlighting):

```
evalSat :: CharPred → Eval r s (Char : xs) (Succ n) a → Eval r s xs (Succ n) a
evalSat f k ctx
   | netWorth ctx ≡ 0 = checkReadAndCache (Coins 1 1 (Just f)) sat ctx
   | credit ctx ⩾ 1    = sat ctx
   | otherwise         = let (c, ctx′) = redeemCredit ctx in checkReadAndCache c sat ctx′
  where sat :: Eval r s xs (Succ n) a
        sat ctx γ{..} = readChar ctx (uncons input) $ λqc f_refined input′ ctx′ →
          let f_opt = optimisePredGiven f f_refined
          in normaliseGen (If (App (charPredToLam f_opt) (Var qc))
                              (Var (k ctx′ (γ {input = input′, moore = QCons (Var qc) moore})))
                              (Var (evalRaise ctx γ)))
        uncons :: Input → (Code Char → Input → Code b) → Code b
        uncons input k = ⟦ let c : cs = $(str input) in $(k ⟦c⟧ (input {str = ⟦cs⟧}))⟧
```

In the first case, the checkReadAndCache function is used to guarantee the existence of a single character, which can then be processed using the continuation sat; in the second case, there is a character ready to be processed immediately with sat; in the last case, credit can be redeemed, and checkReadAndCache can be used, like in evalMeta Credit, to populate the queues with these new characters and apply the credit before again consuming a character with sat. The sat continuation uses the readChar function with an irrefutable pattern match on the input as a fallback continuation (called uncons above). Recall that readChar will feed the first predicate, character, and residual input on the queue and to its given continuation: the f_{refined} returned is the intersection of all predicates known to have been tested against that character, which can be used to optimise the predicate f into f_{opt}. Finally, the same code to apply this predicate to the consumed character and continue as normal is done from the original evalSat. With this change in place, evalSat may be entirely free, and generate no code: this would be the case for sat item, assuming that factoring had been performed, for instance.

Summary

This section started by demonstrating that some parser can benefit from their input being consumed in advance and then presented two layers of analysis and changes to evaluation that makes such optimisation possible. All the examples given in the start of the section are factored by the analysis described. The analysis was split into two phases: the first identifies where in the parser cuts appear, which ensures that no backtracking can occur at the next (<|>) – this relies on scope and the high-level structure of the parser; and the second calculates how many character may be factored out for any given branch – this is complicated by the scoped structure of the combinator, and so is performed on the linearised CPS-transformed machine, where the calculations become trivial. This interplay between the high-level scope information and the low-level control-flow information is interesting because it helps simplify different analyses. However, it is not always clear which parts of an analysis should be performed at which level: it might well be the case that the information that needs to be encoded by the high-level analysis is so complex that the benefits of the low-level's linear control flow is outweighed.

The modifications made to the staged evaluator are more involved than other optimisations seen so far in this chapter. One of the key changes is the introduction of a richer Input type, which can encode additional information about the input state: this will become useful in §6.4, where further information will be stored in Input to help distinguish statically between different Input values.

In the "real" implementation of `parsley`, the input representation is not fixed to be a String (indeed, it also represents String with a companion Int for cheap equality checks): it supports other formats such as `Data.Text` and `Data.ByteString`. This can be done as a zero-cost generalisation since staging can simply insert the correct implementations of the input consumption operation check. With the `Data.ByteString` implementation, checking the existence of *n* characters is particularly cheap if they are not cached, which provides an additional benefit to the checks before bindings from Liquidate.

6.4 Static Refinement of Handlers

At the end of §6.2, the continuation types for BaseFunc, handlers, and retCont were all broken apart to facilitate *staged fusion* of these continuations: this allows for dead-argument elimination for iterative parsers and the terminal continuations. Section 6.3 altered the dynamic state of `parsley` to use an Input record to store more additional information. The failure handler generated by an atomic-less (<|>) combinator will include a conditional check to see whether input has been consumed since the handler has been in scope: this allows for the prevention of backtracking if input is consumed. This section uses the improved static structure of the handlers to perform a light-weight optimisation that can eliminate the condition check within a handler: if it is statically verifiable that no input has been consumed since the handler was created, then the condition check can be skipped, and the correct sub-handler immediately invoked.

Section 6.4.1 starts by outlining some changes to the representation of handlers at the high-level compilation to facilitate a clear static separation between different branches of handlers, or indeed those that have a single path. Then, §6.4.2 augments the Input type to include static information about the origin of a specific input, and how many characters are known to have been consumed from it. Finally, §6.4.3 leverages the improved handler representation and richer information about the input to allow for a static dispatch on the relevant path of a handler if it can be determined.

6.4.1 First-Class Handlers

In §3.1.6, the same handler for (<|>) was shown to be implementable in terms of the selective instructions as follows:

```
same k = red ⟦λnew old → if new ≡ old then Right () else Left ()⟧ (kase raise (pop · k))
```

This works fine but is not inspectable in the way required to ensure the analysis for this section can be performed correctly: the function used to perform the check is not statically inspectable. It is possible to use Defunc (§6.1.1) to make this function inspectable, it would still generate a nested structure that would require eval to be implemented using a histo, which is not ideal. Instead, the Catch instruction is modified to accept a Handler:

```
data Handler k xs n a where
  Same   :: k xs n a → k (Input : xs) n a → Handler k (Input : xs) n a
  Always :: k (Input : xs) n a → Handler k (Input : xs) n a
Catch :: k xs (Succ n) a → Handler k (Input : xs) n a → Instr k xs n a
```

This new Handler type encodes the more complex Same handler, which takes two sub-handlers: the first is the "yes, the input matches" handler, and the second is the "no, the inputs do not match". In the "yes" case, no input is provided on the stack since this is pointless. An advantage of doing this separation here is that the means of checking the equivalence of the Inputs can now be done specifically in the backend, which allows for a cheaper check than the $O(n)$ equality check, and for input to be generalised to other types, like Text, easily. The Always handler deals with all other kinds of handler and has a single sub-handler to defer to. The same k handler helper becomes Same k raise; and compile is adjusted to include Always for the other handlers.

Changes to Analysis By adding the two new handler types into the machinery, the analysis performed in §6.3 needs to be adjusted to accommodate the change. This is straightforward to adapt: any implementations of Catch in the algebras now examine the handlers and their new implementation can be the inlined and simplified definitions for the composition of Red and Case as in the original helper formulation. As such, these changes are omitted for brevity, as they are purely mechanical.

Evaluating `Handler`

The evalCatch function will need to be mostly rewritten to accommodate the different handlers:

```
evalCatch :: Eval r s xs (Succ n) a → Handler (Eval r s) (Input : xs) (Succ n) a → Eval r s xs n a
evalCatch k h ctx γ{..} = evalHandler h (λhandler → k ctx (γ {handlers = handler : handlers}))
  where evalHandler (Always h) k =
              ⟦ let handler input′ = $(h ctx (γ {input = Input ⟦input′⟧ Nothing
                                                   , moore = QCons input moore}))
                in $(k (λqinput → ⟦handler $qinput⟧))⟧
            evalHandler (Same yes no) k =
              ⟦ let yesSame = $(yes ctx γ)
                    notSame input′ = $(no ctx (γ {input = Input ⟦input′⟧ Nothing
                                                    , moore = QCons input moore}))
                    handler input′ | $input ≡ input′ = yesSame
                                   | otherwise      = notSame input′
                in $(k (λqinput → ⟦handler $qinput⟧))⟧
```

Here, the two kinds of handler generate differently shaped code: the Always handler generates a plain binding similar to the original implementation (appendix D); however, Same generates three bindings, one for each of the two outcomes, and then one to perform the dynamic check. In practice, some of these bindings do not need to be generated if the parsers are small enough, though the inlining is omitted from discussion for simplicity. This will be updated in §6.4.3 to leverage the new light-weight static analysis.

6.4.2 Partially-Static Input

To statically determine which branch of a Same handler is taken, different input values need to be statically distinguishable. While the contents of the input is clearly dynamic information, the dynamic origin of input and how many characters are guaranteed to have been consumed off it can be tracked. This information can be added to the Input, making it partially static (§2.4.2).

```
data Offset = Offset {uniqueOrig :: Word, consumed :: Word}
data Input = Input {str :: Code String, deepestKnown :: Code String, staOff :: Offset}
consume :: Offset → Offset
consume off = off {consumed = consumed off + 1}
```

The two new components, uniqueOrig and consumed, both need to be initialised during the parse, and consumed will be updated during each checkAndRead or uncons (§6.3.3). To denote the unique origins of each piece of input, a seed needs to be threaded through the staging in the Ctx:

```
data Ctx r s = Ctx {nts           :: DMap NT (NTBound r s),     phis      :: DMap Φ (PhiBound r s)
                   , regs          :: DMap SigmaVar (QSTRef s), credit    :: Int,
                   , futureCredit  :: Queue Coins,               netWorth :: Int
                   , knownChars    :: RewindableQueue (CharPred, Code Char, Input)
                   , inputUnique   :: Word
                   }
```

The inputUnique field should be initialised to 1 on the creation of the empty context, which only occurs in the main eval function – the initial input is given unique 0. From that point, the context is only threaded through linearly, and due to the presence of Code no values from a sibling path can enter any other sibling statically: the inputUnique is safe to duplicate within siblings. The newUnique function can be used to acquire a unique:

```
newUnique :: Ctx r s → (Word → Ctx r s → a) → a
newUnique ctx{..} k = k inputUnique (ctx {inputUnique = inputUnique + 1})
```

This function simply returns the next unique and then updates it moving forward. It should be used whenever a dynamic boundary is crossed: within handlers for the bound input, for return continuations, and for join points. The relevant changes to Catch are as follows:

```
evalCatch :: Eval r s xs (Succ n) a → Handler (Eval r s) (Input : xs) (Succ n) a → Eval r s xs n a
evalCatch k h ctx γ{..} = evalHandler h (λhandler → k ctx (γ {handlers = handler : handlers}))
```

```
where evalHandler (Always h) k =
          ⟦ let handler input′ = $( newUnique ctx $ λu ctx′ →
                  h ctx′ (γ {input = Input ⟦input′⟧ Nothing u 0, moore = QCons input moore}))
            in $(k (λqinput → ⟦handler $qinput⟧))⟧
      evalHandler (Same yes no) k =
          ⟦ let yesSame = $(yes ctx γ)
                notSame input′ = $( newUnique ctx $ λu ctx′ →
                  no ctx′ (γ {input = Input ⟦input′⟧ Nothing u 0, moore = QCons input moore}))
                handler input′ | $input ≡ input′ = yesSame
                               | otherwise = notSame input′
            in $(k (λqinput → ⟦handler $qinput⟧))⟧
```

The bindings for handler and notSame both now generate a new unique, u, and pass this into the input value fed forward – no input has been consumed from this location yet, so the consumed characters is set to 0. The evalMkJoin and evalCall functions are changed similarly, but this is omitted.

Finally, the changes to checkAndRead and uncons are simply as follows:

```
checkAndRead :: Int → Int → Maybe CharPred → Input → (Input → Code String)
                → (Input → [(Code Char, Input)] → Code a) → Code a → Code a
checkAndRead n m headCheck input sel good bad =
  flip (check n m (sel input) headCheck′) bad $ λdeepest qccs →
    let input′ = input {deepestKnown = Just deepest}
    in good input′ (map (λ(qc, qcs) → (qc, input′ {str = qcs, staOff = consume (staOff input′) })) qccs)
  where headCheck′ = fmap checkFor headCheck
        checkFor f qc good = normaliseGen (If (App (charPredToLam f) (Var qc)) (Var good) (Var bad))
uncons :: Input → (Code Char → Input → Code a) → Code a
uncons input k =
  ⟦ let c : cs = $(str input) in $(k ⟦c⟧ (input {str = ⟦cs⟧, staOff = consume (staOff input′) }))⟧
```

In each function, the input updates the consumed field by 1 whenever a character is known to have been consumed. With this in place, the input at any point in the parser now has static information about both its origins and how many characters are known to have been consumed from it.

6.4.3 Partially-Static Handlers

Now that handlers that check for input equality have been split into three different bindings, and the state of the input is more statically inspectable throughout the parser, the representation of handlers on the stack can be changed to become partially static. The new PSHandler type is as follows:

```
data StaHandler r s = StaAlways (Code String → Code (ST s (Maybe r)))
                    | StaSame Offset                              -- the statically captured offset
                              (Code String → Code (ST s (Maybe r)))  -- full handler
```

```
                                                (Code (ST s r))                          -- yes same
                                                (Code String → Code (ST s (Maybe r)))    -- not same
data PSHandler r s = PSHandler {staHandler :: StaHandler r s
                               , dynHandler :: Code (String → ST s (Maybe r))
                               }
```

The PSHandler type has two components: a StaHandler; and the dynamic representation of the handler, which eases passing it across dynamic boundaries without needing to do etaReduce. The StaHandler type encodes both types of handler as statically broken up continuations, with the StaSame constructor also storing the information about the input bound by the handler. By storing all different possible forms of the handlers, it has the flexibility to choose which should be used in a specific call. This is used in Γ in place of the existing handlers:

```
data Γ r s xs n a = Γ {input    :: Input,                    moore   :: QList xs
                      , handlers :: Vec n ( PSHandler r s ), retCont :: Code a → Code String → Code (ST s r)
                      }
```

The following functions allow for the creation of these PSHandler values (using staFun and dynFun from §2.4):

```
fromDynAlways :: Code (String → ST s (Maybe r)) → PSHandler r s
fromDynAlways dynHandler = PSHandler (StaAlways (staFun dynHandler)) dynHandler

fromStaAlways :: (Code String → Code (ST s (Maybe r))) → PSHandler r s
fromStaAlways staHandler = PSHandler (StaAlways staHandler) (dynFun staHandler)

fromStaSame  ::  Offset → Code (String → ST s (Maybe r)) → Code (ST s r)
             → Code (String → ST s (Maybe r)) → PSHandler r s
fromStaSame offset dynHandler dynYes dynNo =
  PSHandler (Just offset) (StaSame (staFun dynHandler) (staFun dynYes) (staFun dynNo)) dynHandler
```

The fromDynAlways function is used to take dynamic handlers and wrap them up, with no known information about their binding or an Always handler bound in evalCatch; the others are used to wrap up a Same handler statically bound within either evalCatch or the initial handler in eval, which is now replaced with fromStaAlways (const ⟦return Nothing⟧) . In addition to these, invokeHandler performs the static dispatch:

```
invokeHandler :: Input → PSHandler s a → Code (ST s r)
invokeHandler inp (PSHandler (StaAlways h) _) = h (str inp)
invokeHandler inp (PSHandler (StaSame off h yes no))
   | Just True  ← same (staOff inp) off = yes
   | Just False ← same (staOff inp) off = no (str inp)
   | Nothing    ← same (staOff inp) off = h (str inp)
  where same (Offset u₁ n₁) (Offset u₂ n₂)
          | u₁ ≡ u₂     = Just (n₁ ≡ n₂)
          | otherwise = Nothing
```

The invokeHandler function performs the static refinement of the handler: if the handler is a StaAlways, then it is just invoked as normal; otherwise, compare the offset of the input with the bound one of the handler and dispatch based on the result. Comparing the offsets involves first looking at their origins, if these do not match, no comparison can be made; otherwise, if the consumed characters match, then this can be used to pick one of yes or no. With this in place, the evalCatch and evalRaise functions can be adjusted into their final forms:

```
evalCatch :: Eval r s xs (Succ n) a → Handler (Eval r s) (Input : xs) (Succ n) a → Eval r s xs n a
evalCatch k h ctx γ{..} = evalHandler h (λhandler → k ctx (γ {handlers = handler : handlers}))
  where evalHandler (Always h) k =
          ⟦ let handler input′ = $(newUnique ctx $ λu ctx′ →
                  h ctx′ (γ {input = Input ⟦input′⟧ Nothing u 0, moore = QCons input moore}))
            in $(k ( fromDynAlways ⟦handler⟧ ))⟧
        evalHandler (Same yes no) k =
          ⟦ let yesSame = $(yes ctx γ)
                notSame input′ = $(newUnique ctx $ λu ctx′ →
                  no ctx′ (γ {input = Input ⟦input′⟧ Nothing u 0, moore = QCons input moore}))
                handler input′ | $input ≡ input′ = yesSame
                               | otherwise = notSame input′
            in $(k ( fromStaSame (staOff input) ⟦handler⟧ ⟦yesSame⟧ ⟦notSame⟧ ))⟧
evalRaise :: Eval r s xs (Succ n) a
evalRaise ctx γ {..} = let Cons h _ = handlers in invokeHandler input h
```

The other minor adjustment is that fromDynAlways needs to be used for handlers that are bound by continuations, like in return continuations within evalCall, and introduced in other non-top-level bindings within eval.

Summary

Making failure handlers into a partially-static representation allows for the conditional check of input equality to be omitted and the correct sub-handler used in its place. The way this is implemented is relatively light-weight, in the sense that no full analysis has been performed across the parser has been performed to alter what handlers are bound into the Catch instruction, and instead stage-time information is threaded through instead. While this is simple to implement, with a more obviously correct implementation than a global analysis would be, it does mean that the analysis is only approximate.

The optimisation can be improved, however, by performing ahead-of-time analysis on each binding within the parser to establish whether they: consume a fixed amount of input on success, always consumed a varying amount of input, consume no input on success, or behave unpredictably. This would allow for the analysis to propagate more information through return continuations, allowing for failures within to make use of the optimisation too. This may mean introducing a finer-grained representation of the origins of an input value, that can account for same origin but with an unknown amount of consumption on the value: this is future work.

An advantage of a global input-use analysis is that it could eliminate dead handlers before the code is generated for them: now, unused yesSame, notSame, and handler bindings in the generated code must elided by GHC, instead of avoiding generation up-front. As mentioned though, a global analysis would be heavier and trickier to get right.

If a binding could be known to never consume input, then the handler passed to it can also be refined to be the yesSame variant, which could be converted into a dynamic handler using const. By refining the types of the bindings themselves, however, it would be possible to change the type to reflect this knowledge and avoid the const entirely.

Finally, an aspect of the design of the real system omitted here, is that the handlers can specify whether each component is small enough to be inlined directly, which can allow small handlers like the one used for atomic to be inlined and simplified. This is done by providing a Bool argument with each sub-handler that says whether a binding should be generated or spliced directly into the use-sites. This will generate lambdas if these inlined handlers are forced to cross a dynamic boundary.

6.5 Benchmarks

This chapter has presented a variety of different optimisations. This section aims to quantify the effectiveness of some of these optimisations as well as the performance of the completed implementation of Haskell `parsley` itself. All benchmarks are performed using GHC `9.0.2` (the GHC version is limited by the use of the helper `parsley-garnish` library); and make use of the `gauge` benchmarking library, which is a simpler version of `criterion` with the same rigorous benchmarking support.

6.5.1 Comparing with other Libraries

To start, `parsley` is compared against four other libraries: `attoparsec`, `megaparsec`, `parsec` [Leijen and Meijer 2001], and `happy` [Gill and Marlow 1995]. Of these four, `megaparsec` and `parsec` are tested using `String` and `Data.Text`; `happy` is tested with `String` using handwritten threaded lexers; and `attoparsec` is tested with `Data.Text` only.

In each benchmark, the data is presented as relative to an implementation using `parsley` with the corresponding input type: values greater than 1 are slower than `parsley` and values smaller than 1 are faster than it. For fairness, the parsers for each of the parser combinator libraries are all identically written: for example, while `megaparsec` and `attoparsec` both have specialised combinator for reading chunks of text, these are not used as they are not supported in `parsley` (it is future work to either add them, or add analysis to automatically identify and apply these patterns). The `happy` parsers have been implemented to ensure that the relevant optimisation flags and left recursion are used idiomatically..

As is standard for benchmarking Haskell, the results of the parsers are deeply evaluated to ensure they are fully forced so that the benchmark is not executed lazily. To reduce the overhead of this process, each benchmark is parsed into an AST that is fully strict, and smart constructors are employed within the parsers to ensure that constructing a node also deeply forces any other lazy Haskell types, like lists. As such, the result of the parser need only be evaluated to weak-head normal form.

JSON Parsers

The first benchmark is concerned with parsing JSON, a portable format for data storage based off JavaScript's objects. This is a relatively simple non-ambiguous grammar with no expressions or natural left recursion – this means implementation with parser combinators is very straightforward. The benchmark runs on four files: "smalldata.json" is a 550-character file with a single object inside; "mediumdata.json" is an array of 5 objects totalling 6850 characters; "bigdata.json" is an array of 10 larger nested objects totalling 15328 characters; and "hugedata.json" is an array of 511 objects like those in "bigdata.json", totalling 744207 characters.

The results can be seen in fig. 6.1; in the graph, benchmarks using `Text` are distinguished by patterned bars. This shows that `parsley`, at value 1, is outperforming all the other libraries for this parsing task. It is up to 1.8× faster than `happy` for the smallest data, though this advantage starts to drop off as the data size gets particularly big. This is true for the other libraries as well, which seem to fare comparatively better at the larger input size. This is likely due to some residual remaining space-leaks in the generated parser, which will need to be hunted down in due course. To demonstrate that this may be the case, table 6.1 shows the time taken to parse each character normalised against the size of the file itself: for `parsley` this increases more heavily than it does for `parsec`. Since the code for the parser itself does not change and the grammar is non-ambiguous (and so parses in linear time), it is likely that a space-leak somewhere is causing unnecessary additional computation; as `parsec` shares a similar semantics to `parsley`, the scaling between the two should be similar.

WACC Parsers

The WACC language, used for Imperial's undergraduate compiler project, is a fully functioning programming language loosely based on the educational While programming language. The language contains conditionals; functions; looping constructs; expressions; statements; and basic types like integers, booleans, pairs, and strings. There are several ambiguities in the grammar, and many instances of left recursion. The parser combinator implementations have employed idiomatic techniques to resolve the left recursion (using iterative combinators), and most of the ambiguities (except for two tricker ones) have been factored out with techniques discussed

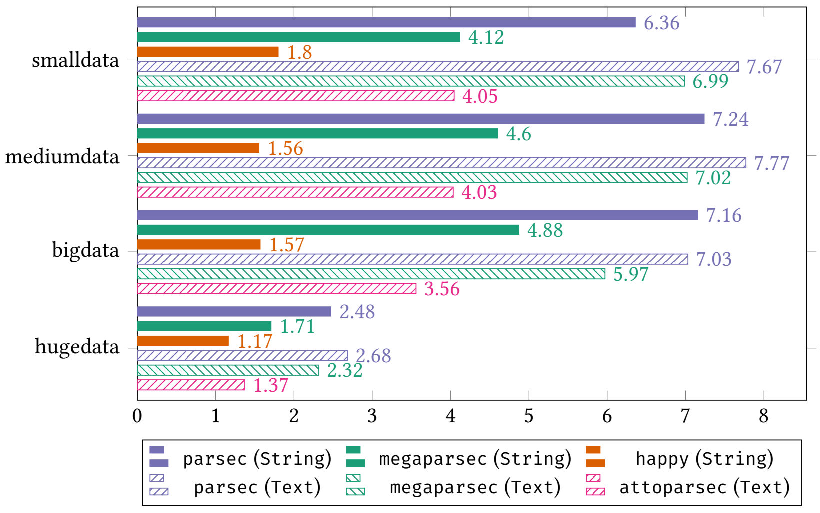


Fig. 6.1: Performance of libraries parsing JSON, time relative to `parsley`.

Total characters	550 chars	6850 chars	15328 chars	744207 chars
`parsley`	9.1	12.5	13.8	41.4
`parsec`	57.8	90.9	99.03	102.4

Table 6.1: Time to parse each character (ns/char).

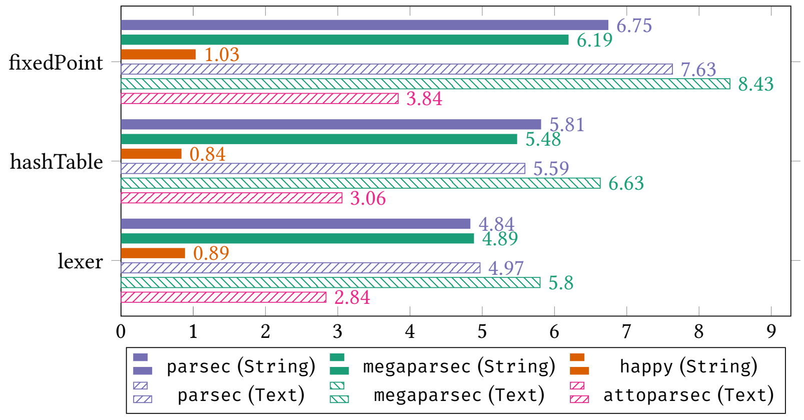


Fig. 6.2: Performance of libraries parsing WACC, time relative to `parsley`.

in §7.1.3. WACC is a keyword heavy language, which means backtracking will be performed if keywords and identifiers overlap; however, the threaded lexer used for `happy` will also read these characters twice. The three files tested are an implementation of fixed-point arithmetic, which is 123 lines long; an implementation of a hash table, which is 321 lines long; and an implementation of a lexer for WACC itself, which is 2693 lines long.

Fig. 6.2 shows the results of the benchmarking. Similarly to the JSON benchmarks, `parsley` out-performs the other parser combinator libraries by 3 to 4× for `attoparsec` and 4 to 8× with `parsec` and `megaparsec`. The fact that `parsec` and `megaparsec` are so close here is interesting, with GHC `8.10.7`, `megaparsec` pulls much further in front of `parsec` (more in line with `attoparsec`): it is not clear what changed across the GHC versions to cause this. However, `happy` in this case comes either level with or up to 15% faster than `parsley`. This is likely due to the differences between the recursive descent paradigm and `happy`'s $LALR(1)$ parsing algorithm: these are known to be linear time and very efficient bottom-up parsers. It is likely that `happy` is avoiding additional recursion during parsing, which is aiding its performance.

JavaScript Parsers

The final cross-library benchmark is parsing the JavaScript programming language. There are a few programs being parsed: "fibonacci.js" is a small implementation of the Fibonacci function, a mere 18 lines long; "heapsort.js" is a 50-line implementation of the heapsort algorithm on an array; "game.js" is a 167-line implementation fragment of a small browser game; and "big.js" is a big-number library totalling 925 lines. The implementations of the parsers are relatively ambiguity-free, with many expression precedence levels. Similarly to WACC, the parser combinators have addressed this in an idiomatic way, and `happy` is using its precedence resolution and left recursion to resolve shift-reduce conflicts. Compared with WACC, JavaScript is a less keyword heavy language, which may impact the performance of handling keywords.

The results of the benchmark are shown in fig. 6.3. As with the WACC benchmark, `attoparsec` performs better than both `megaparsec` and `parsec` but is trailing behind `happy` and `parsley`. The relative speedup of `parsley`

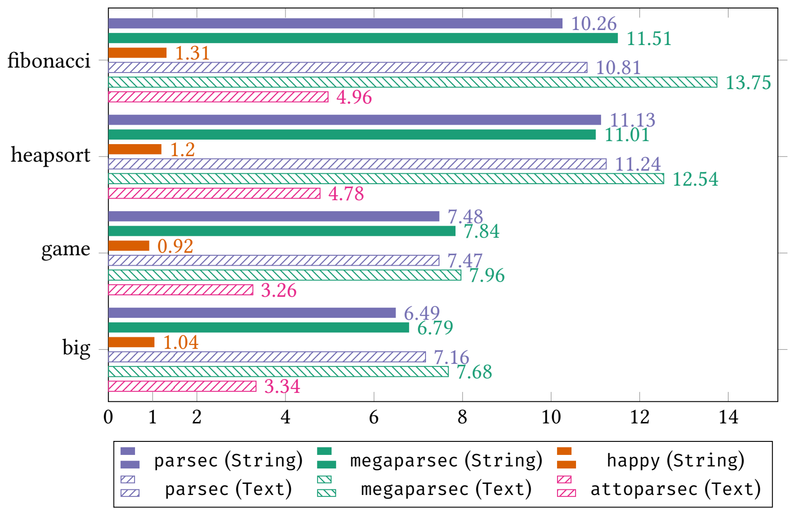


Fig. 6.3: Performance of libraries parsing JavaScript, time relative to `parsley`.

against these slower parser combinator libraries ranges from 6.5 to 13×, depending on the input type and file size. Like the JSON benchmark, `parsley` seems to be exhibiting some slowdown as the file increases in size, probably as a result of similar overly lazy code. Compared with the WACC benchmark, `happy` is up to 30% slower than `parsley`, reaching a peak of 8% faster in the "game.js" case. This may be due to the less ambiguous grammar, or that more iterative combinators are used to handle a deeper precedence table: in `parsley`, iterative combinators are noticeably more efficient than recursive ones (§6.5.2).

6.5.2 Effect of Optimisations

This section evaluates the effectiveness of the some of the optimisations developed within this chapter. The three most notable ones are: the idempotent handler optimisations for iterative combinators; the consumption analysis and input factoring; and the static dispatch of handlers based on whether input has been consumed or not.

Iterative Optimisation and References

Iterative combinators, defined using the loop combinator, like many and some, can have been optimised to make use of references and prevent recursive invocation of handlers, instead re-using the same one for each iteration. This section explores how beneficial these are.

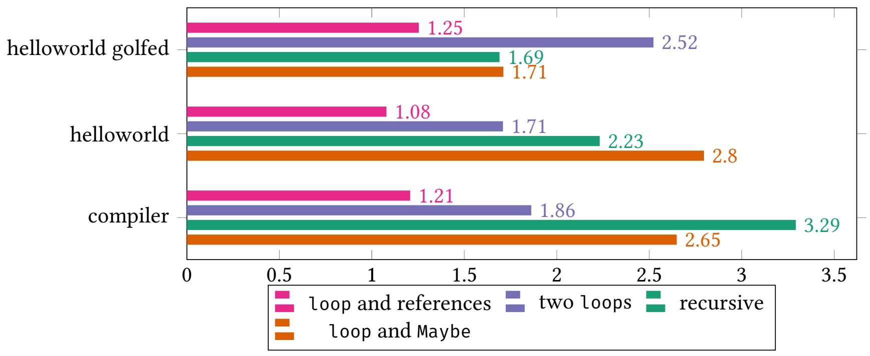


Fig. 6.4: Performance of iterative *Branflakes* parsers relative to the handwritten implementation.

For this, the *Branflakes* language is used (first seen in CHAPTER 4). The structure of this language is simple, with eight core operators (two of which forming a recursive pair) and everything else is a comment. Five different parsers for this language have been implemented:

- A tail-recursive handwritten implementation that uses a difference list to collect up the results.
- A naïve recursive `parsley` implementation, that just recurses when a comment character is encountered – the results are built recursively as the parser unwinds.
- A `parsley` implementation that reads operators with many reading any residual comment characters in a skipMany loop after each is consumed – the result is built using a difference list in a reference.
- A `parsley` implementation using a single many loop, which wraps operators in Just and comments as Nothing – the resulting list is flattened with catMaybes.
- A `parsley` implementation using a single loop combinator with non-abstracted use of references: here, a difference list is also used, but comment characters are just ignored on each iteration – no Maybe, secondary loop, or recursion.

These five parsers are tested against three files: a commentless implementation of "hello world!", which is 111 characters long; a version with comments, 869 characters long; and a compiler, which is 28618 characters long.

The results are shown in fig. 6.4. As the file size increases, the recursive implementation gets relatively slower compared to the iterative versions, which remain more stable: this is likely pointing to the source of the performance slowdown in the cross-library benchmarks being as a result of recursion. The version using a single loop but wrapping the results in Maybe is mostly slower than the others: this is likely because of the extra traversal over the result list, or the allocation of more Just values. The two loops and reference implements are generally the fastest: however, when the input contains comments, the two loops are the slowest, due to the repeated switching between different hot loops. By far the most stable and fastest implementation of the `parsley` version is the manual use of references: here, there is one clean tight loop, with no redundant handler allocation or traversals, and the results are collected in an efficient way. In fact, this implementation is only within 8% to 25% slower than the handwritten optimal version: an inspection of the generated GHC Core suggests that the only difference is the use of the STRef as opposed to accumulation parameter – future reference analysis would optimise this to the same code!

In all, this demonstrates the scalability and effectiveness of the iterative pattern in the `parsley` combinators and shows that effective use of references can help build very efficient parsers.

Consumption Analysis

This section explores the effect of input factoring on both the `String` and `ByteString` backends for `parsley`. For context, the implementation seen in this dissertation has focused on `String`, but other input types are supported – including `ByteString` and `Text`. In particular, `ByteString` is very well suited to input factoring optimisations, as the check is very cheap – just a length check on an array. This contrasts to `String`, where pattern matching must occur for each factored character. To assess the effect of the optimisation, two small parsers are given:

Input	`String`		`ByteString`	
	on	off	on	off
`abcdef`	11.33	10.01	5.36	8.19
`uvwxyz`	11.88	12.91	5.89	9.36
`012345`	8.97	8.49	5.03	7.12
`abcde`	8.18	9.57	4.49	8.05

(a) parser `p` with input factoring on and off.

Input	`String`		`ByteString`	
	on	off	on	off
`abc0123456789`	25.22	19.97	6.61	7.77
`uvw0123456789`	24.67	20.45	6.81	7.96
`xyz0123456789`	13.39	6.52	4.77	5.23
`abcx123456789`	16.51	16.01	4.86	6.46
`abc012345678`	12.73	14.93	4.13	5.42

(b) parser `q` with and without factoring over join-points.

Table 6.2: Time taken to parse given input (ns) for

```
p = string "abcdef" <|> string "uvwxyz"
q = (string "abc" <|> string "uvw") *> string "0123456789"
```

The parser `p` is used to test how general input factoring performs against a variety of different inputs, and `q` specifically demonstrates how well input factoring across join-points works – recall that the string that follows the initial `<|>` will be let-bound in the parser. These two parsers are compiled with and without these optimisations on, against both `String` and `ByteString` – specifically, for the let-bound factoring, "off" means input factoring is enabled, just not across the let-boundary.

Table 6.2 shows the effect of the analysis and its associated optimisations. In table 6.2a, it is apparent that input factoring for `String` is less effective when no backtracking would have otherwise occurred (first row) and that it slow down parsing when neither alternative will match – this is because all characters must be acquired before the first one can be matched. However, when the input would have been deconstructed multiple times (row 2), it does provide a benefit, as the previously fetched character is reused. When not enough input is available, it is also faster since it avoids any comparisons of the characters. This means that the optimisation is probably best disabled or limited to single-character factoring for `String`. However, with `ByteString`, the optimisations provide a universal benefit, as length checks are coalesced into a single point: this can result in around a 35% to 45% improvement for even these small strings – this would be particularly useful for parsing keywords!

Table 6.2b tells a similar story, with factoring across a join being harmful for the `String` parser in all but the last case, where not enough input is present. Here, the effect is amplified, because the input must still be read twice – the factoring ensures existence, but this just results in irrefutable pattern matches on the other side of the binding, not pre-available characters. However, for `ByteString`, this is always beneficial even accounting for the existing improvement from the base factoring: this is resulting in a single length check instead of two checks.

Static Handler Resolution

By tracking whether input has been consumed statically, `parsley` can remove the dynamic check for whether or not input has been consumed. In the real implementation, this check is cheap, as an integer offset is carried through the parser with the input as opposed to doing an equality check on the input string. This benchmark establishes what benefit the static resolution of a handler that checks for input consumption has in practice. For

	`fixedPoint.wacc`	`hashTable.wacc`	`lexer.wacc`
Improvement	4.26%	12.79%	1.68%

Table 6.3: Performance improvement from statically deducing input consumption.

this, the WACC parser and benchmarks input are reused, with and without static resolution turned on.

The results are summarised in table 6.3: in each case, the optimisation has been effective, providing a good speed-up across all cases. It is not clear why the benefit is so much higher for the hash table example, but it is likely that this parser fails more often than the other two examples on its way to a successful parse.

Discussion

In all, `parsley` exhibits very competitive performance against `happy`, and offers improvements against other parser combinator libraries. The optimisations performed offer a performance benefit, however this may depend on the type of input consumes, as well as the subtleties of the grammar itself. To overcome this, the user can fine-tune what optimisations are turned on by passing a configuration object to the parse function. Clearly, the consumption analysis can be improved to avoid the pathological behaviours for the `String` backend moving forward, however it is already effective for `ByteString`. Further improvements to the static analysis of handlers across recursion boundaries will help improve its effectiveness further.

Summary

This chapter introduced explicit defunctionalisation of internally-defined values and user-defined predicates. This is useful, as it unlocks extra analysis and optimisation opportunities: this is particularly relevant for static error message generation, where error content depends on predicate domains.

High- and low-level optimisations have been employed to simplify the AST and leverage some domain-specific knowledge that GHC cannot employ. In addition, more aggressive analysis like input factoring and static application of failure handlers is performed. Again, these exploit information that GHC is unaware of, and input factoring has a good benefit for `ByteString`. Optimising handlers exposes a need for further analysis: more aggressive specialisation of continuations is possible with more information and more type-level information.

The benchmarks are promising, demonstrating `parsley`'s performance is competitive against other parser combinator and generator libraries, and that the proposed optimisations are effective.

Chapter 7

The Design Patterns of Parser Combinators

The following chapter has been adapted from both "Design Patterns for Parser Combinators (Functional Pearl)" [Willis and Wu 2021] and "Design Patterns for Parser Combinators in Scala" [Willis and Wu 2022] both in collaboration with Nicolas Wu.

While the implementation and overall interface for a parser combinator library has been well-studied, not much has been said about how to structure the parsers we write to keep them maintainable; or how to make good APIS that complement these parsing design ideas. The need for these ideas stems out of observing how parsers were written by students undertaking Imperial's WACC project: the same sticking points appear very often, and the patterns of this chapter were developed to help address these issues, additionally forming part of our marking criteria – this has proven effective.

As parsers are first-class values in the combinator model, the full high-level abstractions of the host-language are available to both the writer of parsers and the writer of libraries. This means that the regular rules of good practice for software development apply to the construction of parsers in a way that they might not for parsers written in the parser generator style, which is less prescriptive. The patterns of use for parser combinators leverage the first-class nature of parsers to help create clean abstractions that encapsulate problems and leave the underlying grammar and parser intact.

This chapter starts by first introducing and discussing: the different problems, anti-patterns, patterns, and their impact on an overall parser (§7.1); before introducing the ways in which library design can be informed by these principles, illustrating how libraries can facilitate the use of these patterns in both Haskell and Scala (§7.2).

7.1 How to Structure and Design Parsers

Design patterns are a tool for programmers that allow them to structure large bodies of code in an organised and maintainable way. The perhaps most well-known literature on design patterns is *Design Patterns: Elements of Reusable Object-Oriented Software* [Gamma et al. 1995], which talks about twenty-three different patterns for structuring Object-Oriented (OO) code. Interestingly, many of the patterns introduced by the so-called "Gang of Four" in an OOP languages, can be plainly implemented with higher-order functions in a functional one like Haskell. A criticism of design patterns is that they are a substitute for missing language features: in more modern languages, the classic design patterns find less and less use.

Aims The goals of the parser combinator design patterns are as follows:

- Improve how parsers are organised and structured.
- Create a clearer distinction between the different roles of the sub-parts of a parser.
- Cleanly decouple extraneous logic from the "syntactic" part of the parser itself, keeping its shape as close as possible to the plain, underlying, BNF grammar

- Help constrain the implementation of the parser such that it can be reasonably assumed to be correct using stronger typing guarantees.

The presentation and discussion of the patterns introduced in this chapter will be centred around a running example, which will get more complex as required to help illustrate key points and will assume the regular `parsec`-style API and semantics offered by both Scala and Haskell `parsley`. For simplicity, the Template Haskell noise of Haskell `parsley` will be omitted, and instead will be treated more like an API iteration on `megaparsec`.

Running Example

The language used to illustrate the design patterns and associated problems will start with a simple expression language and build out to one with statements as required (fig. 7.1). This grammar consists of simple tokens like identifier, numbers, and some literals like '`+`', '`negate`' – the overall *lexical* structure of the language is described in fig. 7.1a; and these can be used to construct simple arithmetic expressions where multiplication binds tighter than addition and subtraction, precedence can be overridden with explicit parentheses (fig. 7.1b), and all arithmetic operators are explicitly left-associative – denoted in the grammar via left-recursion. The distinction between the lexical structure and grammatical structure, and its importance, will be discussed in §7.1.2.

Instead of a parse tree, parser combinators naturally specify the desired output of the parser as a *semantic action*: often, this is an abstract syntax tree (AST) representing the structure of the parsed expression without encoding unnecessary additional information about the parse. A simple AST for this language would be:

```
data Expr = Add Expr Expr | Sub Expr Expr | Mul Expr Expr
          | Neg Expr
          | Val Integer | Var String
```

Each operation in the grammar is expressed using a single constructor each, with numbers and identifiers each getting their own constructor. In this representation, the entire tree is formed from a single type, so parentheses are normalised out of the datatype – it suffices to write Mul (Add (Val 5) (Val 3)) (Var "x") to express overriding of the natural precedence ordering. A first attempt at writing a parser for the language generating a value of the Expr datatype might be as follows:

```
  -- Lexical structure
digit  :: Parser Char
digit  =  oneOf ['0'..'9']
```

```
⟨digit⟩  ::= '0' .. '9'
⟨letter⟩ ::= 'a' .. 'z' | 'A' .. 'Z'
⟨number⟩ ::= ('+' | '-')? ⟨digit⟩+
⟨ident⟩  ::= ⟨letter⟩ (⟨letter⟩ | ⟨digit⟩)*
```

(a) Lexical structure

```
⟨expr⟩ ::= ⟨expr⟩ '+' ⟨term⟩ | ⟨expr⟩ '-' ⟨term⟩ | ⟨term⟩
⟨term⟩ ::= ⟨term⟩ '*' ⟨neg⟩ | ⟨neg⟩
⟨neg⟩  ::= 'negate' ⟨neg⟩ | ⟨atom⟩
⟨atom⟩ ::= '(' ⟨expr⟩ ')' | ⟨number⟩ | ⟨ident⟩
```

(b) Grammatical structure of expressions

Fig. 7.1: The description of the expression language

```
letter   :: Parser Char
letter   = oneOf (['a'..'z'] ++ ['A'..'Z'])
number   :: Parser Integer
number   = (negate <$ char '-' <|> id <$ char '+' <|> pure id)
         <*> (foldl addDigit 0 <$> some digit)
  where addDigit n d = n * 10 + digitToInt d
ident    :: Parser String
ident    = letter <:> many (letter <|> digit)
  -- Grammatical structure of expressions
expr     :: Parser Expr
expr     = Add <$> expr <*> (string "+" *> term) <|> Sub <$> expr <*> (string "-" *> term) <|> term
term     :: Parser Expr
term     = Mul <$> term <*> (string "*" *> neg) <|> neg
neg      :: Parser Expr
neg      = Neg <$> (string "negate" *> neg) <|> atom
atom     :: Parser Expr
atom     = string "(" *> expr <* string ")" <|> Val <$> number <|> Var <$> ident
```

This parser is an almost one-to-one translation of the grammar, with the extra parts for handling the semantic content of the combinators. In particular, number uses a fold over the parsed digits to create an Integer result with arbitrary precision, ident combines the leading letter with the trailing letters and digits to form a single String, and each of the operators combine their results with the corresponding AST constructors using the Applicative combinators. A key problem with this parser is the ambiguity of several branches, requiring the atomic combinator, but this is a minor concern given the other problems present and will be trivially fixed in due course.

The Problems While the above parser models the grammar precisely, it exhibits several problems that the design patterns seek to resolve:

Left-recursion (§7.1.1) The most pressing issue with the parser as it stands is the use of *left recursion*. This has been done to faithfully represent the left-associativity of the operators as defined by the grammar, but for `parsley`, and indeed most parser combinator libraries, left-recursion will result in unproductive infinite recursion.

To fix this, instead of using *left-factoring* algorithms to modify the grammar, so-called *chain* combinators are employed to abstract the factoring, and additional type-safety can introduced into the parser by using a heterogenous version of the AST. Further improvements can be made by making use of a precedence combinator.

Poor lexing (§7.1.2) Another issue with the parser is that there is no attempt at dealing with whitespace. In particular, the lexical and grammatical portions of the parser are intermingled, which does not provide a clean separation of the two distinct ideas.

For this, combinators are designed to help enforce conventions on whitespace, and these are exposed sparingly to ensure the parser does not need to be concerned with whitespace handling – this is not a property encoded by the BNF and is instead part of lexing itself. Furthermore, other combinators are developed to deal with symbols including keywords and operators.

Tightly coupled AST construction (§7.1.3) The current parser has a very tight coupling between the grammar itself and the way that the AST should be constructed as a result. This coupling seems benign with the current requirements of the AST, but even still it is obscuring the underlying structure by introducing noise into each rule. The real problem comes when further impositions are made on the parser by the AST, for example position information, or data invariances: with the current coupling, this impacts the clarity of the parser.

For this, the two tasks of parsing and AST construction are detached from each other and the interaction of their behaviours are managed more effectively by *parser bridges* – a term borrowed from the OOP design patterns [Gamma et al. 1995].

7.1.1 Expression Parsing and Left-Recursion

Expressions usually consist of operators of varying fixities and associativities at various levels of precedence. Grammars can usually encode this by using a clear and standardised shape: higher-precedence operators appear as the atoms to weaker precedence levels, with same precedence operators appearing in the same rule; and then the fixities and associativities are usually encoded via the specific shape of the rule (fig. 7.2).

The presence or absence of recursion to the left or right of an infix operator indicates what kind of associativity it has: recursion indicates the side of the operator that other versions of that same operator may appear on. Left-associative operations are therefore implemented using left-recursive grammar rules (fig. 7.2a): this is problematic because recursive-descent parsers, which most parser combinator libraries produce, do not handle left-recursion in a productive way.

> ***Problem 1: Left-Recursive Expressions*** *Expressions with left recursion cannot be encoded by recursive-descent parsers and will diverge [Willis and Wu 2021].*

One approach to solving this problem is to use grammar left-factoring algorithms such as Paull's algorithm [Moore 2000; Aho et al. 2006; Hopcroft, Motwani, and Ullman 2001], which re-associates the grammar so that it is purely right-associative. This problem with this approach is that it also affects the associativity of the generated parse-tree, which then requires correction, and it also obscures the original intention of the grammar instead exposing implementation details.

$\langle expr\rangle ::= \langle expr\rangle$ ‘-’ $\langle term\rangle \mid \langle term\rangle$

(a) Left-associative infix operator

$\langle expr\rangle ::= \langle term\rangle$ ‘&&’ $\langle expr\rangle \mid \langle term\rangle$

(b) Right-associative infix operator

$\langle expr\rangle ::= \langle term\rangle$ ‘<’ $\langle term\rangle \mid \langle term\rangle$

(c) Non-associative infix operator

$\langle expr\rangle ::= \langle expr\rangle$ ‘+’ $\langle expr\rangle \mid \langle term\rangle$

(d) Associative infix operator

Fig. 7.2: Standard forms for infix operators

The Homogeneous Chain Combinators

Instead, using parser combinators, a more effective approach can be taken by abstracting such algorithms behind combinators that preserve the original semantic action encoded by the naïve left-recursive parser. Such combinators were first described by Hutton and Meijer [1996] and are called the "chain" combinators: they take an iterative approach to parsing the operations and combine the results via folding.

```
chainl1 :: Parser a → Parser (a → a → a) → Parser a
chainr1 :: Parser a → Parser (a → a → a) → Parser a
```

For each of these combinators, chainx1 p op will parse many ps separated by ops and apply the results together x-associatively. Crucially, neither is implemented using left recursion. In particular, the nature of their types means that they only operate on values of type a: this means they are *homogeneous*.

> ***Pattern 1a: Homogeneous Chains*** *For binary operators where the associativity is not specified, use chainl1 or chainr1 to combine operands with their operators [Willis and Wu 2021].*

With these two combinators on hand, it is possible to fix the left-recursion in the original example parser:

```
expr :: Parser Expr
expr  = chainr1 term (Add <$ string "+" <|> Sub <$ string "-")
term :: Parser Expr
term = chainl1 neg (Mul <$ string "*")
```

While this can now productively parse expressions, it no longer adheres to the original grammar (fig. 7.1b): notice that chainr1 has been erroneously used instead of chainl1, but the shape of the original grammar specified addition and subtraction as left-associative! Importantly, the description of the pattern says that the chainx1 combinators are best suitable when the associativity of the operator is not specified. Concretely, this means that the operator should have the shape described by fig. 7.2d: the parser must decide about how the expression should associate as an implementation detail, and the chainx1 combinators allow that decision to be changed without any additional work. But, as mentioned, the example grammar in 7.1b has explicitly encoded addition and subtraction as being left-associative – while addition is fully associative, subtraction certainly is not – so the flexibility afforded by the homogeneous chains has translated into a bug.

The Heterogenous Chain Combinators

The combinators introduced in §7.1.1 are useful for eliminating left-recursion within a parser, but they do not help ensure the grammar's intended associativity in a type-safe way. The key to seeing why is to compare the type of the operator given to a chain and the recursion structures found in fig. 7.2: the type of '+' in ⟨*expr*⟩ ::= ⟨*expr*⟩ '+' ⟨*expr*⟩ is precisely Expr → Expr → Expr, which matches the type a → a → a demanded by the chains. Instead, the type of '-' in ⟨*expr*⟩ ::= ⟨*expr*⟩ '-' ⟨*term*⟩ is Expr → Term → Expr, which has the more general shape of b → a → b. These more specific types can be used to formulate two new chains, whose definitions are as follows:

```
infixl1 :: (a → b) → Parser a → Parser (b → a → b) → Parser b
infixl1 wrap p op = (wrap <$> p) <**> rest
  where rest = flip (·) <$> (flip <$> op <*> p) <*> rest <|> pure id
infixr1 :: (a → b) → Parser a → Parser (a → b → b) → Parser b
infixr1 wrap p op = p <**> (flip <$> op <*> infixr1 wrap p op <|> pure wrap)
```

Here, the more specific shapes of operators are employed, clearly denoting where recursion is allowed to appear in the expression represented by the "infix" chains: the side with the b is the side where the recursion lies. Using these definitions, it is also possible to define the homogeneous chains:

```
chainl1 = infixl1 id
chainr1 = infixr1 id
```

Here, both a and b are the same type, so the wrapping of one layer into the other is just id. To complete the set, a trivial non-recursive infixn1 combinator can also be built to handle the associativity pattern of fig. 7.2c too:

```
infixn1 :: (a → b) → Parser a → Parser (a → a → b) → Parser b
infixn1 wrap p op = p <**> (flip <$> op <*> p <|> pure wrap)
```

Unary chains In addition to the infix versions of the heterogeneous chain combinators, it is also useful to define versions that can be used for unary operators in both prefix and postfix form:

```
postfix :: (a → b) → Parser a → Parser (b → b) → Parser b
postfix wrap p op = (wrap <$> p) <**> rest
  where rest = flip (·) <$> op <*> rest <|> pure id
prefix :: (a → b) → Parser (b → b) → Parser a → Parser b
prefix wrap op p = op <*> prefix wrap op p <|> wrap <$> p
```

The handling of unary operators has less information to constrain the types compared with infix operators, so these combinators are less "safe," though they can have a reversed argument order to make the intent clear. In fact, the shape of the postfix combinator is similar enough to that of infixl1 that, in fact, it is possible to define infixl1 in terms of postfix:

```
infixl1 wrap p op = postfix wrap p (flip <$> op <*> p)
```

However, it is not possible to define infixr1 in terms of prefix, because parsing p <**> op will fail having consumed input if there is no op, preventing a proper recovery in `parsec`-like libraries.

Subtyping The purpose of the wrap function is to allow the layer below (represented by the Parser a) to serve as a Parser b in the case that no operators can be parsed. This makes the combinators less ergonomic (though this will be improved in §7.1.1) but note that this is a subtyping relation between the a and b layers: the wrap function often achieves a simple form of explicit *upcasting*. This can be exploited by languages with true subtyping, like Scala, and a more ergonomic type can be given to infixl1 and infixr1:

```
def infixl1[A, B >: A](p: Parsley[A], op: =>Parsley[(B, A) => B]): Parsley[B]
def infixr1[A, B >: A](p: Parsley[A], op: =>Parsley[(A, B) => B]): Parsley[B]
```

Here the wrap function present in the Haskell version has disappeared and is replaced by the explicit subtyping relation that `A <: B`. The definitions of both prefix and postfix can be made more ergonomic as well, but in a different way:

```
def prefix[A](op: Parsley[A => A], p: =>Parsley[A]): Parsley[A]
def postfix[A](p: Parsley[A], op: =>Parsley[A => A]): Parsley[A]
```

It might seem that this formulation has dropped the fact that the layer below can subtype into the current layer, however the application of a `Parsley[A]` to a `Parsley[B => B]` where `A <: B` is trivial by covariance. Indeed, Scala will happily upcast `A` into a `B` anyway, leaving all the types the same: there is simply nothing added by distinguishing between `A` and `B` in this case. That said, Scala does provide a natural way to extend of these combinators, called `prefix1` and `postfix1`:

```
def prefix1[A, B <: A](op: Parsley[A => B], p: =>Parsley[A]): Parsley[B]
def postfix1[A, B <: A](p: Parsley[A], op: =>Parsley[A => B]): Parsley[B]
```

Both combinators require at least one `op` to be parsed, as opposed to the zero required by the regular `prefix` and `postfix`. This is made clear in their types: both return a `B`, and there is no way to upcast from `A` to `B` (notice that the subtyping relation works the other way around here). Instead, by guaranteeing that `B` is a subtype of `A`, any value provided to the `op`, which will either be the initial `p` or a recursive operator, will be either of type `A` already or is upcastable to `A`; and there must be at least one operator to be able to return the `B`.

> ***Pattern 1b: Heterogeneous Chains*** *For associative operators where operand types may differ, use infixl1 or infixr1 to combine operands with their operators, in conjunction with strongly-typed semantic actions [Willis and Wu 2021].*

Using heterogeneous chains To use heterogeneous chains, the type of the semantic actions being performed within the chain also need to reflect the stronger typing guarantees. Recall the example AST:

```
data Expr = Add Expr Expr | Sub Expr Expr | Mul Expr Expr
          | Neg Expr
          | Val Integer | Var String
```

This tree exhibits the exact problem we are trying to avoid when using the new chains, namely that, Add :: Expr → Expr → Expr, which means that it has not properly encoded the extra guarantees desired. Instead, a stricter AST must be defined to leverage the safer chain interface:

```
data Expr  = Add Expr Term | Sub Expr Term | OfTerm Term
data Term  = Mul Term Neg                  | OfNeg  Neg
data Neg   = Neg Neg                       | OfAtom Atom
data Atom  = Val Integer   | Var String    | OfExpr Expr   -- or Parens
```

In this version of the tree, each layer of precedence is explicitly encoded as a separate datatype, with the transitivity between different layers encoded using an OfX constructor. As mentioned above, a more natural way of encoding these relations would be using sub-typing apart from OfExpr, which ties the knot and would always need to be explicitly represented. The overall shape of this hierarchy of datatypes matches the grammar far more precisely, which will make the parser more likely to be correct if it type-checks. Since these constructors match with the types demanded by the infixx1 chains the parser can be reworked as follows:

```
expr :: Parser Expr
expr = infixl1 OfTerm term (Add <$ string "+" <|> Sub <$ string "-")
term :: Parser Term
term = infixl1 OfNeg  neg  (Mul <$ string "*")
neg  :: Parser Neg
neg  = prefix OfAtom (Neg <$ string "negate") atom
atom :: Parser Atom
atom = string "(" *> (OfExpr <$> expr) <* string ")" <|> Val <$> number <|> Var <$> ident
```

With this formulation, it is not possible to (a) change the associativity of the operators or (b) re-order them incorrectly without getting a type-error. However, this increased safety has an ergonomic cost with respect to any processing that happens post-parse – although, the stronger types can have advantages when writing equally very structured code like pretty printers. It may be desirable to flatten the datatype down into a homogeneous one after the expression itself has been parsed to make working with it easier. In a language where this has been encoded by subtyping, however, this disadvantage is less evident.

Generalising to Precedence Tables

In §7.1.1, the chain combinators were introduced as a clean way of handling the parsing of associative operators without falling afoul of left recursion. However, the shape of an expression parser making use of chains is still very formulaic. It is possible to abstract this more into something that is more akin to a *precedence table*.

Precedence combinators [Fokker 1995] exist in a few different forms across common parser combinator libraries and the literature [Danielsson and Norell 2011; Hill 1996; Hill 1994]. The idea is to root the table with a given "atom" and then build out the layers of the table from this. Like the existing chain combinators found in the wild, however, these precedence combinators are homogeneous, usually working with a [Ops a]-like structure. Since heterogeneous chains also exist, it would be ideal to be able to generalise this kind of machinery to work with those instead.

```
data Fixity a b sig where
  InfixL  :: Fixity a b (b → a → b)
  InfixR  :: Fixity a b (a → b → b)
  InfixN  :: Fixity a b (a → a → b)
  Prefix  :: Fixity a b (b → b)
  Postfix :: Fixity a b (b → b)
```

Firstly, the relation between the different fixities and the types of the semantic actions is encoded by a Fixity type that will force the unification of a and b with sig when required.

```
data Op a b where Op :: Fixity a b sig → (a → b) → Parser sig → Op a b
```

This is then used by Op to encode a single layer of the precedence table. Here, the type variable sig is existential, which is why the tying of sig to a and b in Fixity is important. A value of type Op a b indicates that the layer adapts values of type a into values of type b.

```
data Prec a where
  Level  :: Prec a → Op a b → Prec b
  Atom :: Parser a → Prec a
infixl 5 >+
(>+) = Level
infixr 5 +<
(+<) = flip (>+)
```

Finally, the heterogeneous list Prec can be defined, where each layer is tied to the next by an Ops object. The (>+) and (+<) are then convenience operators that allow for the list-like structure to be formulated in either strongest-to-weakest or weakest-to-strongest ordering: these operators are greedy; their "mouths" point towards higher precedence levels. This structure is what is then folded by the precedence combinator:

```
precedence :: Prec a → Parser a
precedence (Atom atom) = atom
precedence (Level lvls ops) = con (precedence lvls) ops
  where con p (Op InfixL wrap op)   = infixl1 wrap p op
        con p (Op InfixR wrap op)   = infixr1 wrap p op
        con p (Op InfixN wrap op)   = infixn1 wrap p op
        con p (Op Prefix wrap op)   = prefix wrap op p
        con p (Op Postfix wrap op) = postfix wrap p op
precHomo :: Parser a → [Op a a] → Parser a
precHomo atom = precedence · foldl (>+) (Atom atom)
```

The precedence combinator itself is just a simple recursive folding of the heterogeneous list, feeding each layer into the next and using the corresponding heterogeneous chain to formulate the layer. However, constructing Op values by-hand is not ideal, especially when the relationship between a and b can be leveraged to simplify construction – as seen in precHomo.

```
gops :: Fixity a b sig → (a → b) → [Parser sig] → Op a b
gops fixity wrap = Op fixity wrap · choice

ops :: Fixity a a sig → [Parser sig] → Op a a
ops fixity = gops fixity id
```

```
class sub < sup where
  upcast   :: sub → sup
  downcast :: sup → Maybe sub

sops :: a < b ⇒ Fixity a b sig → [Parser sig] → Op a b
sops fixity = gops fixity upcast
```

Instead, smart constructors can be made that capture these common patterns of use: gops is the root, and it allows the user to specify more than one operator at the same level by using choice; ops captures the basic case with a homogeneous tree, recovering the classic precedence behaviour – it does this by providing the id function as the layer wrapper; and sops captures the case where each layer of the tree is in a subtyping relationship with the next, except in-lieu of real subtyping, a typeclass is used to provide explicit upcast and downcast operations – more on that in §7.2.1.

> ***Pattern 1c: Precedence Tables*** *For expressions, use the precedence combinator to deal with both fixity and precedence concisely [Willis and Wu 2021].*

If each layer of the refined AST developed in §7.1.1 has been given an instance of the (<) typeclass, a more concise definition of the expr parser can be given as follows:

```
expr :: Parser Expr
expr = precedence $
  sops InfixL [Add <$ string "+", Sub <$ string "-"] +<
  sops InfixL [Mul <$ string "*"]                    +<
  sops Prefix [Neg <$ string "negate"]               +<
  Atom atom
```

By making use of the sops smart constructor, the OfX wrappers that existed in the previous solution are no longer required, as they are being implicitly summoned by the typeclass resolution in Haskell. As before, trying to reorder the layers in the table, remove one, or change the fixities, without first changing the corresponding datatype will result in a type error, as desired.

Generic Parsing Folds

The chain combinators are not just useful for expression parsing, in fact, they can be used to build generic combinators that deal with iteration. The definitions of the chains in terms of monadic (>>=) – which is best avoided, as this can be an expensive operation – demonstrate clearly how the chains are the embodiment of a fold with accumulation (this is also true of the applicative-style version, but it is less obvious in that presentation):

```
chainl1 p op = p >>= rest
  where rest acc = do f ← op; y ← p; rest (f acc y)
                   <|> pure acc
```

In each iteration of parsing op and p, the results are combined with the current accumulation and passed forward onto the next iteration. Compare this with the original naïve definition using foldl:

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

The version using direct recursion and monadic (>>=) is doing much the same work but is doing so in a *deforested* style [Gill, Launchbury, and Peyton Jones 1993; Wadler 1988] where the list of intermediate values does not need to be constructed. This may be more efficient than the folding version, which was found to be true during the original development of such combinators in `parsley` [Willis 2018], where using a deforested implementation of the combinator provided an improvement of 25%.

There are examples of parsers that have this kind of pattern, where results are combined on each iteration, usually implemented by using a fold over the many or some combinators, in fact the number parser from §7.1 is one such example. Using the postfix, prefix, and infixl1 combinators, it is possible to build deforested combinators:

```
manyl  :: (b → a → b) → b → Parser a → Parser b
manyl f k p = postfix id (pure k) (flip f <$> p)

manyr  :: (a → b → b) → b → Parser a → Parser b
manyr f k p = prefix id (f <$> p) (pure k)

somel  :: (b → a → b) → b → Parser a → Parser b
somel  f k p = infixl1 (f k) p (pure f)

somer  :: (a → b → b) → b → Parser a → Parser b
somer  f k p = f <$> p <*> manyr f k p
```

In these definitions, the folding function f is partially applied to each element and then chained together. In particular, the use of infixl1 is interesting, as the use of a non-id wrap function means that somel cannot be modelled as a chainl1. These are more primitive than the many and some combinators, which can both be defined as identities on these combinators:

```
many = manyr (:) [ ]
some = somer (:) [ ]
```

Using somel, it is possible to improve the number definition in the parser so that it does not need to generate an intermediate list of digits:

```
number  ::  Parser Integer
number  =  (negate <$ char '-' <|> id <$ char '+' <|> pure id)
              <*> somel addDigit 0 digit
  where addDigit n d = n * 10 + digitToInt d
```

7.1.2 Lexing and Whitespace Abstraction

In §7.1.1, the parser for the expression grammar was restructured to remove the problem of left recursion. This ensures that the parser can be ran without diverging, but the parser as written is still not correct. This is because the parser has made no attempt at distinguishing between different categories of tokens, and no attempt at handling the whitespace between tokens. Here are some concrete issues that will be tracked during development[1]:

[1]the wrapping constructors are omitted for clarity. A broken case is represented by "✗", and a fixed one with "✓".

```
parse expr "x + 7"           ≡ Just (Var "x")          -- ✗
parse (expr <* eof) "x + 7"  ≡ Nothing                 -- ✗
parse expr " x"              ≡ Nothing                 -- ✗
parse expr "x a"             ≡ Just (Var "x")          -- ✗
parse expr "negatex"         ≡ Just (Neg (Var "x"))    -- ✗
```

It is possible to fix most of these problems by adding whitespace consuming parsers throughout the main parser itself, but this is messy and intrusive.

> ***Problem 2: In-Place Lexing*** *Dealing with tokens while parsing is intrusive to the overall structure of the parser and introduces clutter [Willis and Wu 2021].*

If the parser writer had decided to go with a parser generator-based approach, it is more likely that they would have concretely separated the parsing into two passes: a lexing pass to generate a stream of tokens, followed by a parser that operates purely on the tokens. In this model, the lexer is usually tasked with the handling of whitespace, as it is whitespace that delimits the tokens of the grammar. However, this has the disadvantage that the token generation is treated in complete isolation from the grammar: this can lead to token ambiguities where two symbols are treated the same by the lexer where they appear in entirely different positions within the grammar and could therefore be made distinct. An example of this in the grammar of fig. 7.1 is the ambiguity of the (sub)-token '`-`', which appears twice: it can form part of a negative integer literal, or it may serve as a binary subtraction operator. The difficulty here, is how should the text fragment `"-68"` be treated by the lexer: at this point, the information previously lexed has been forgotten (and is otherwise has no true structure yet anyway), so how should this be tokenized? It would be entirely obvious within the parser, which will either have just parsed an expression and know this is a subtraction, or will otherwise know it is a negative literal, but the lexer has no such context. This can, in part, be addressed with a *threaded* lexer, where tokens are lexed on-demand from the parser, but still, this is not how parser combinators usually tackle this problem.

> ***Anti-pattern 2: Lexing then Parsing*** *Pre-processing the input with a dedicated lexer has no contextual awareness with which to selectively construct tokens [Willis and Wu 2021].*

Parser combinators still represent tokenizing machinery using the same set of combinators as for general parsing: to best fix the issues with lexing, however, there should be a clear separation between the two kinds of parser, and a separation of their individual concerns. I make these distinctions when deciding which things should be lexer tokens and which should be parser rules:

1. A parsing rule, except for whitespace-sensitive languages, should never be concerned with, or even aware of, whitespace.
2. A lexical token should not be able to contain whitespace: it is *effectively* atomic.
3. A lexical token should not interact with AST nodes or other semantic actions, but instead return basic, primitive types.

For the expression grammar, the lexical components are all the plain symbols, which should return () as there is no reason for a literal symbol to have a more complex result; the numbers, which cannot have spaces after the sign; and identifiers. The parsing rules are then everything left, which all construct AST nodes using either other rules or tokens from the lexer. From this point, the lexical tokens and combinators will be kept in a separate module from the parsing rules.

A Convention for Whitespace

The first step to tackling the whitespace problem is to settle on a uniform convention for how whitespace should be consumed within the parser. There are three different choices:

1. Consume whitespace uniformly before and after every token. (✗)
2. Consume whitespace uniformly before every token. (✗)
3. Consume whitespace uniformly after every token. (✓)

For both cases (1) and (2), consider a parser that tries to parse one of two arbitrary tokens t_1 and t_2, t_1 <|> t_2, and observe what would happen if just leading whitespace ws was consumed before the tokens: in this case ws *> t_1 <|> ws *> t_2 would be the resulting parser and suddenly an ambiguity has been introduced requiring backtracking on every single (<|>) to resolve – this degrades parsing time complexity to $O(2^n)$. This leaves the only sensible option as approach (3), which will be used throughout this section.

> ***Pattern 2a: Whitespace Combinators*** *Build a lexeme combinator dedicated to consuming trailing whitespace after a given lexeme. Build a fully combinator to consume initial whitespace and the end of input [Willis and Wu 2021].*

Everything the lexing module exposes must have handled whitespace using the convention outlined above and should also avoid exposing this convention so that the parser cannot be whitespace aware. To handle all the symbols within the main parser, the lexing module exposes a symbol combinator:

```
-- private                          -- public
ws :: Parser ()                     symbol :: String → Parser ()
ws = skipMany (space <|> ...)       symbol sym = lexeme (void (string sym))

lexeme :: Parser a → Parser a       number = lexeme ...
lexeme p = p <* ws                  ident  = lexeme ...
```

By simply substituting out the string combinator from the main parsing rules for the new symbol combinator, the state of the problematic cases is now as follows:

```
parse expr "x + 7"   ≡ Just (Add (Var "x") (Val 7))  -- ✓
parse expr " x"      ≡ Nothing                       -- ✗
parse expr "x a"     ≡ Just (Var "x")                -- ✗
parse expr "negatex" ≡ Just (Neg (Var "x"))          -- ✗
```

So far, this fixes the issue where whitespace was not being processed by the parser, but this has not worked for the second case: the convention is that trailing whitespace is always consumed by tokens, but that leaves the initial leading whitespace unhandled; for the third case, the parser cannot handle the `'a'` at the end of the input, but it also has not been forced to try consuming all the input in the parser. Both tasks are the job of the fully combinator, which should be used exactly once around the top-level portion of the grammar:

```
-- Lexer.hs
fully :: Parser a → Parser a
fully = ws *> p <* eof
```

```
-- Parser.hs
lang :: Parser Expr
lang = fully expr
```

```
parse lang "x + 7"    ≡ Just (Add (Var "x") (Val 7))  -- ✓
parse lang "  x"      ≡ Just (Var "x")                -- ✓
parse lang "x a"      ≡ Nothing                       -- ✓
parse lang "negatex"  ≡ Just (Neg (Var "x"))          -- ✗
```

The fully combinator has been used to define a lang parser wrapping around expr. Now, the only remaining problem is with handling conjoined tokens requiring refining the handling of identifiers and keywords.

Building Token Parsers

With whitespace properly handled, the distinctions between various kinds of symbolic tokens, and their relation to each other needs to be established.

It is worth elaborating on one point from §7.1.2: "A lexical token should not be able to contain whitespace: it is *effectively* atomic." The meaning of *effectively atomic* is that the token is internally indivisible, so that it must not be possible to succeed in parsing half of the token but may fail having consumed part of token without backtracking. Often, however, the backtracking is actually desirable: symbols are likely to overlap with other symbols, so should be marked with atomic to make them truly atomic; however, there are some tokens with unambiguous leading characters, string literals perhaps, where backtracking would be counter-productive. In the example grammar, however, everything can be handled uniformly with atomic: to help enforce this a private token combinator can be introduced:

```
-- private
token :: Parser a → Parser a
token = lexeme · atomic
```

```
-- public
symbol sym = token (void (string sym))
number = token ...
ident = token ...
```

The ordering of backtracking and whitespace consumption in the token combinator is important: any backtracking from the failed token should be done before whitespace consumption, otherwise a parse error in the whitespace would cause a needless token backtrack instead of an immediate failure.

Pattern 2bi: Tokenizing Combinators *Annotate terminals with a token combinator, built on lexeme, to atomically parse them with whitespace consumed [Willis and Wu 2021].*

Keywords and Identifiers The last issue in the parser is parse lang `"negatex"` ≡ Just (Neg (Var `"x"`)): instead, this should produce the identifier Var `"negatex"`. The prefix negation operator sees a `"negate"` at the start of the input, and consumes it without verifying that this is not, part of an identifier. While it does not materialise in this grammar, since negation always appears before identifiers, it is also an issue that identifier is able to parse keywords – this should only be allowed in the presence of so-called "soft keywords."

> ***Pattern 2bii: Keyword Combinators*** *Avoid using string with token for keywords. Use a keyword combinator that enforces that the keyword does not form a valid prefix of another token [Willis and Wu 2021].*

To fix this issue, it suffices to use negative-lookahead with keywords, and filtering with identifiers:

```
keys      :: Set String
keys      = Set.fromList ["negate"]

keyword   :: String → Parser ()
keyword k = token $ string k *> look_ (letter <|> digit)

ident     :: Parser String
ident     = token $ filterS (∉ keys) (letter <:> many (letter <|> digit))

symbol    :: String → Parser ()
symbol sym
   | sym ∈ keys = keyword sym
   | otherwise = token (void (string sym))
```

By ensuring that a keyword k is not followed by a valid identifier letter before consuming the whitespace, the keyword combinator will ensure that only legal keywords are parsed. Keywords are almost always just valid identifiers in a language, so this will also work for keywords where one is a prefix of another. The last step to fixing the parser is to hook the keyword combinator into the symbol combinator, so that string tokens are treated entirely uniformly, with no special casing – if soft keywords existed in this grammar, they would still use keyword explicitly, as they would not be in the keys set. At this point, all the issues with the parser are fixed:

```
parse lang "x + 7"   ≡ Just (Add (Var "x") (Val 7))  -- ✓
parse lang "  x"     ≡ Just (Var "x")                -- ✓
parse lang "x a"     ≡ Nothing                       -- ✓
parse lang "negatex" ≡ Just (Var "negatex")          -- ✓
```

Operators One last point of discussion for token parsing is about parsing operators: many languages do not permit user-defined operators and, as such, the technique of ensuring an operator is not followed by a valid operator letter does not necessarily work well to prevent ambiguities between different operators. Instead, operators should be handled by a mixture of user-defined operator letter checks (if available) as well as checks to ensure that the other "hard operators" cannot be parsed.

The most efficient way of handling this is to store the set of operators as a Trie data structure [Fredkin 1960; Okasaki and Gill 2000; Morrison 1968]:

```
data Trie   = Trie {present :: Bool, children :: Map Char Trie}
suffixes    :: String → Trie → Trie    -- O(n log m), n is length of key, m is branching factor of nodes
member      :: String → Trie → Bool
member k = present · suffixes k
foldTrie    :: (Bool → [b] → b) → (Char → b → b) → Trie → b
delete      :: String → Trie → Trie
```

A Trie is a structure that encodes a set of strings by sharing common prefixes of keys in the set, supporting a suffixes operation that can efficiently find the sub-trie of all the keys in the set that share a common given prefix. A fold over a Trie can yield an optimal parser for recognising elements of that set, requiring no backtracking [Ljunglöf 2002]:

```
trie :: Trie → Parser ()
trie = foldTrie layer child
  where layer True _   = pure ()
        layer False cs = choice cs    -- simplifies to empty when cs is empty.
        child c p      = char c *> p
```

This combinator, which does not generate the best error messages, traverses each layer of the Trie by parsing each child character followed by the parser that recognises that suffix Trie; if at any point, the end of a legal string is reached, the parser terminates with a success.

Using this combinator, an operator can be defined with the desired behaviour:

```
ops :: Trie
operator :: String → Parser ()
operator op = token $ string op *> look_ (trie (Trie.delete "" (Trie.suffixes op ops)))
```

This combinator first tries to parse the given op, then will find all the suffixes of that op that would form valid larger operators: the empty string must be removed from the resulting Trie, since this will be present if given op is a member of the ops set, and subsequently always succeed, failing the look_. With the operator combinator, symbol can be modified one final time to accommodate the new operator parsing:

```
symbol :: String → Parser ()
symbol sym
   | Set.member sym keys = keyword sym
   | Trie.member sym ops  = operator sym
   | otherwise            = token (void (string sym))
```

Abstracting Behind String Literals

With all the problems in the parser fixed, attention can be turned to ergonomic issues. The focus of the past two sections has been on the lexing file itself, the attention now shifts to the parsing rules, which are as follows:

```
lang = fully expr

expr  :: Parser Expr
expr  = precedence $
  sops InfixL [Add <$ symbol "+", Sub <$ symbol "-"] +<
  sops InfixL [Mul <$ symbol "*"]                     +<
  sops Prefix [Neg <$ symbol "negate"]                +<
  Atom atom

atom :: Parser Atom
atom = symbol "(" *> (OfExpr <$> expr) <* symbol ")" <|> Val <$> number <|> Var <$> ident
```

The remaining issue with the parser at this point is the noise introduced by the symbol combinator. Some languages, Scala and Haskell included, have support for overloading the literals of the language to mean something else. In Scala's case, anything can make use of an implicit conversion to change one thing into another where required; in Haskell's case, certain kinds of literals, like integers, strings, and lists, can be repurposed to return a different type.

> ***Pattern 2c: Overloaded Strings*** *Hide tokenizing logic by allowing string literals to serve as parsers [Willis and Wu 2021].*

A natural step to take is to make string literals in the host language serve as parsers. Ideally, these string literal parsers should work with the tokenizing logic already defined in the lexer: in languages like Haskell, where instances must be confluent, this means that the library should refrain from defining the instance itself or keep it in an importable module away from main functionality, even if that means the defined instance is orphan. Haskell has the IsString typeclass and `OverloadedStrings` language extension that can be used to make strings into parsers:

```
  -- Lexer.hs
instance u~() ⇒ IsString (Parser u) where fromString = symbol
```

With this simple instance in place, parser can be modified by just enabling the language extension and ensuring the instance has been brought into scope.

```
 {-# LANGUAGE OverloadedStrings #-}
lang = fully expr

expr  :: Parser Expr
expr  = precedence $
  sops InfixL [Add <$ "+", Sub <$ "-"] +<
  sops InfixL [Mul <$ "*"]             +<
```

```
    sops Prefix [Neg <$ "negate"]        +<
    Atom atom
  atom :: Parser Atom
  atom = "(" *> (OfExpr <$> expr) <* ")" <|> Val <$> number <|> Var <$> ident
```

In Scala, implicit conversions can be used for this instead (appendix A.4.2): here there is no issue of global coherence [Peyton Jones, Jones, and Meijer 1997], so if the implicits are kept in well-defined scopes, a library can also provide a conversion for just the `string` combinator:

```
given implicitSymbol: Conversion[String, Parsley[Unit]] = symbol(_)  // in the Lexer object
given stringLift: Conversion[String, Parsley[String]]   = string(_)  // in a library object
```

7.1.3 Decoupling Semantic Actions from Plain Syntax

Parser combinators interleave the semantic actions of the parser with the syntactic description directly, as each fully complete parser returns a user-directed result. This contrasts with parser generator-based approaches, where the semantic actions are often kept to the side, or even handled as fully separate visitors [Gamma et al. 1995] over a bespoke parse tree [Parr 2013]. Having tightly coupled code, however, is not considered best engineering practice, as it makes the code harder to maintain. In this case, there are three main ways in which changing requirements of the semantic actions can interfere with the syntactic description of a grammar:

- The structures produced change shape in a way that necessitates restructuring of the parser even if the syntax of the language does not change: this might be that the data is canonicalised, say (§7.1.3).
- Metadata about the parse needs to be extracted and associated with the produced data (§7.1.3).
- Additional constraints or invariances are imposed on the generated data that need to be verified within the context of the parser itself, without being part of the grammar explicitly (§7.1.3).

In each of these cases, the maintainability and clarity of the parsers decrease. For example, when building a compiler, the semantic action of parsing is usually an AST, and this AST is usually passed forward to some semantic analysis phase: this may generate errors, and so position information from the parsing needs to be known and encoded into the tree.

> ***Problem 3: Bookkeeping*** *ASTs built during parsing occasionally require parser metadata [Willis and Wu 2021].*

To demonstrate this, the grammar described in fig. 7.1b is extended to support a new ⟨*let-expr*⟩ node, which is also added into ⟨*atom*⟩ instead of ⟨*expr*⟩, the new or updated grammar rules are shown in fig. 7.3, with the rest of grammar remaining unchanged. To support this, the AST used up to this point is also altered, and the parser is patched up as follows (continuing from the final implementation in §7.1.2):

```
  data LetExpr = Let String LetExpr LetExpr | OfExpr Expr
   ...
  data Atom    = Val Integer | Var String    | OfLetExpr LetExpr    -- or Parens
```

```
letExpr :: Parser LetExpr
letExpr = Let <$> ("let" *> ident) <*> ("=" *> letExpr) <*> ("in" *> letExpr)
          <|> OfExpr <$> expr
  ...
 atom   :: Parser Atom
 atom   = "(" *> (OfLetExpr <$> letExpr) <* ")" <|> Val <$> number <|> Var <$> ident
```

This new fragment of parser still retains the nice similarity with the original BNF grammar apart from the usual noise of the (<*>) combinator. Now assume that the writer of this parser has gone on to implement a full compiler for this little language. Now suppose they want to do scope analysis on a syntactically valid program and report any shadowed or declaration-less variables. Position information now needs to be encoded into the AST:

```
data Pos = Pos {line :: Int, col :: Int}
data LetExpr = Let String LetExpr LetExpr Pos | OfExpr Expr
 ...
data Atom    = Val Integer | Var String Pos     | OfLetExpr LetExpr   -- or Parens
```

Here, both Let and Var have been associated with a position represented by the type Pos. The problem now, however, is that this position information needs to be injected into the parser, which is not expressed in BNF, and injected into the correct part of the constructors. This is complicated by the fact that position information needs to be parsed just before a token and then placed at the end of the constructor:

```
pos = Pos <$> line <*> col
letExpr :: Parser LetExpr
letExpr = (λp v e b → Let v e b p) <$> ("let" *> pos) <*> ident <*> ("=" *> letExpr) <*> ("in" *> letExpr)
          <|> OfExpr <$> expr
 ...
atom    :: Parser Atom
atom    = "(" *> (OfLetExpr <$> letExpr) <* ")" <|> Val <$> number <|> pos <**> (Var <$> ident)
```

Here, the placement of the position within the Let (and the fact it must be parsed just before the ident) makes for an awkward dance involving the Let constructor –Pos could have been moved to the first argument to mitigate this, but the structure of the AST should not be dictated by parsing implementation detail[2]– and the same goes for the Var case, though this is made simpler by the use of the (<**>) combinator. Perhaps a cleaner way of doing this would have been to package up the identifier with the position instead:

[2]This is particularly true for Scala, where it is desirable to have defined `case class Let(v: String, e: LetExpr, b: LetExpr)(val pos: Pos) extends LetExpr` to avoid the need to include the position information in every pattern match: in Scala only the first set of arguments is considered for both pattern matching and equality.

⟨*let-expr*⟩ ::= ‘`let`’ ⟨*ident*⟩ ‘`=`’ ⟨*let-expr*⟩ ‘`in`’ ⟨*let-expr*⟩ | ⟨*expr*⟩

⟨*atom*⟩ ::= ‘(’ ⟨*let-expr*⟩ ‘)’ | ⟨*number*⟩ | ⟨*ident*⟩

Fig. 7.3: New rules for let-expressions

```
data Ident = Ident String Pos
```

But this would now introduce extra indirection into the AST, which may not be desirable for the future work the compiler writer will do. Again, it is forcing the AST's shape to be tied to the specific implementation details of the parser, which is not ideal: the focus of this section is to explore ways of mitigating this, regardless of which approach is taken to resolve the underlying problem.

> ***Anti-pattern 3: Inline Bookkeeping*** *Incorporating metadata inline into the parser is intrusive and brittle [Willis and Wu 2021].*

Abstracting with Parser Bridges

The main problem identified with the parser defined in §7.1.3 is that the tight coupling of semantic actions and syntactic description is preventing the requirements of these two distinct parts of the parsing from varying independently from one another: a change to one necessitates a change to the other. While the last example focused on position information being one way to damage the parser's structure, it also demonstrated a point that the structure of the AST in general cannot vary independently from the way the parser expects it to be built: by addressing this issue in a general way, the remaining issues can be resolved as special instances of this technique. *Design Patterns: Elements of Reusable Object-Oriented Software* presents a pattern known as a **Bridge** that is designed to address this exact problem:

> ***Bridge****: Decouple an abstraction from its implementation so that the two can vary independently [Gamma et al. 1995].*

Fully-Saturated Bridges This lends its name to the parser combinator design pattern called a *Parser Bridge*. The idea is to create a minimalistic function that combines various opaque parsing components to build a specific constructed value. The parser then uses this directly instead of data constructors and the bridge can adapt the real AST to the interface exposed to the parser without needing to change the parser itself. In Haskell, instances of this pattern are known as *Lifted Constructors* [Willis and Wu 2021], though in Scala it is more appropriate to refer to them as *Parser Bridges* given Scala's relationship with the OOP world. This technique is simple but effective. At its simplest, a *Lifted Constructor* for a position-less Let would be as follows:

```
  -- This should be kept in a separate module, perhaps alongside the AST itself
mkLet :: Parser String → Parser LetExpr → Parser LetExpr → Parser LetExpr
mkLet = liftA3 Let
letExpr = mkLet ("let" *> ident) ("=" *> letExpr) ("in" *> letExpr) <|> OfExpr <$> expr
```

In this case, the implementation is trivial, making use of the basic applicative liftA3 combinator, which is where the name *Lifted Constructor* comes from. The first insight, though, is that just by abstracting in this straightforward way, it is possible to rework the AST node and just change the function provided to liftA3 to patch the changes in. The parser now has a stable interface with which to construct the Let node. The second advantage is that it also removes some of the noise from the parser that results from the (<*>) combinator.

This technique can already be applied across several of the nodes in the AST. In particular:

```
mkVal :: Parser Integer → Parser Atom          mkVar :: Parser String → Parser Atom
mkParens :: Parser LetExpr → Parser Atom       mkExpr :: Parser Expr → Parser LetExpr
```

Except for mkVar, which, like mkLet, requires position information (addressed fully in §7.1.3), each of these can be implemented trivially in terms of the "lift" combinators. But as soon as the requirements on this AST change, these will still retain their stable API, which still makes them valuable.

Unsaturated Bridges Conspicuously, no bridges have been postulated for the arithmetic operators in the expression grammar. This is because the formula of lifting a constructor to work on parsers only works when the application of the bridge is *fully saturated*: all its arguments are provided at the call-site of the function. This is not true of the arithmetic operators: their arguments are not applied immediately, instead this application is performed within a chain combinator due to the precedence combinator. The *Parser Bridge* pattern, however, is flexible enough that this is not an issue, it suffices to alter the external shape of the bridge to adapt to the pattern-of-use required of it. In the Scala formulation, this is known as a *Singleton Bridge*, where the bridge acts as a parser all on its own: by leveraging Scala's objects, this can be done in a clean way via a purpose built combinator – this is demonstrated in §7.2.3. In Haskell, for the trivial case, it suffices to just use pure:

```
mkAdd :: Parser (Expr → Term → Expr)           mkMul :: Parser (Term → Neg → Term)
mkAdd = pure Add                               mkMul = pure Mul
mkSub :: Parser (Expr → Term → Expr)           mkNeg :: Parser (Neg → Neg)
mkSub = pure Sub                               mkNeg = pure Neg
```

With the types of all the bridges defined, these can decouple all the semantic actions across the parser:

```
letExpr = mkLet ("let" *> ident) ("=" *> letExpr) ("in" *> letExpr) <|> mkExpr expr
expr    = precedence $
  sops InfixL [mkAdd <* "+", mkSub <* "-"] +<
  sops InfixL [mkMul <* "*"]               +<
  sops Prefix [mkNeg <* "negate"]          +<
  Atom atom
atom    = "(" *> mkParens letExpr <* ")" <|> mkVal number <|> mkVar ident
```

Now that the semantics actions have been decoupled from the parser, they are free to vary in many interesting ways – including adding the desired position information to Let and Var – without changing this parser.

Metadata Bridges

In §7.1.3, the introduction of position information into the AST resulted in an invasive and disruptive transformation of the reasonably clean initial parser. However, §7.1.3 introduced a simple technique for decoupling the AST construction, the semantic action, from the syntactic description of the parser itself. With this abstraction in place, it is now possible to tweak the bridges so that they satisfy the metadata requirements of the altered AST:

```
mkLet :: Parser String → Parser LetExpr → Parser LetExpr → Parser LetExpr
mkLet v e b = pos <**> (Let <$> v <*> e <*> b)

mkVar :: Parser String → Parser Atom
mkVar v = pos <**> (Var <$> v)
```

By adding in the pos combinator to the relevant positions within the bridge, the position metadata has been incorporated into the AST. It is easy to move the position to a different argument in the constructors. In section §7.1.3, it was mentioned that an Ident datatype could be used to package up the positions with identifiers, hiding them from the grammar. With the bridges in place, this solution can be explored without changing the AST itself by adapting the constructor to fit the arguments. In this case, the types of the bridge do change:

```
mkLet :: Parser Ident → Parser LetExpr → Parser LetExpr → Parser LetExpr
mkLet = liftA3 let′ where let′ (Ident v pos) e b = Let v e b pos

mkVar :: Parser Ident → Parser Atom
mkVar = liftA var′ where var′ (Ident v pos) = Var v pos
```

In these definitions, the sequencing part of the bridge is left untouched, but the function provided is performing some canonicalisation of the structure by flattening out the information.

Verifying Bridges

In addition to metadata extraction, bridges can also be used to enforce data invariances from the semantic domain back onto the syntactic domain. As an example, the current type of Val in the AST is an arbitrary-sized Integer: there is no harm in using this lossless type in the construction of number tokens, but suppose the expression language was changed to only support the 64-bit Int datatype and, in addition, exceeding this limit is supposed to be a syntactic error. In this case, it is possible to encode this overflowing property directly in the grammar, but that is complex and obscures the intention of the grammar. Instead, the bridges can be used to perform this validation instead, leaving the parser and lexer untouched.

```
mapMaybeP :: (a → Maybe b) → Parser a → Parser b

mkVal :: Parser Integer → Parser Atom
mkVal = mapMaybeP integerToInt
  where max = toInteger (maxBound :: Int)
        min = toInteger (minBound :: Int)
        integerToInt x | min ⩽ x, x ⩽ max = Just (Val (fromInteger x))
                       | otherwise        = Nothing
```

To achieve this, the mapMaybeP combinator, which combines the act of filtering and mapping into a single combinator, will try and convert the result of the Parser Integer into an Int, assuming that it falls within the 64-bit range, and wrapping it into a Val; or otherwise fails (making better error messages for this is left to CHAPTER 8). This kind of filter-like operation inside a bridge makes that bridge into a *Verifying Bridge*: once again, the parser did not need to change to implement the new semantic constraint.

Disambiguator Bridges

The final, notable, type of bridge is called a *Disambiguator Bridge*. Instead of decoupling data-construction tasks away from the parser and leaving it to purely describe the syntax, this kind of bridge interacts with the syntax in a way that can optimise away backtracking by exploiting common structure between two different ambiguous parsers.

The current grammar does not have any instances of ambiguity in it that a *Disambiguator Bridge* can exploit; to demonstrate this idea, the grammar is temporarily extended with an ⟨*atom*⟩-level vector literal, of the form ⟨*vec*⟩ ::= '(' ⟨*let-expr*⟩ (',' ⟨*let-expr*⟩)* ')', along with a new constructor Vec :: [LetExpr] ⟶ Atom. To start with, a naïve definition can be given that relies on backtracking to disambiguate between ⟨*vec*⟩ and parenthesised expressions, which both share a leading '(' and ⟨*let-expr*⟩:

```
mkVec :: Parser [LetExpr] ⟶ Parser Atom

atom = "(" *> (atomic (mkParens        letExpr        <* ")")
                <|>        mkVec (sepBy1 letExpr ",") <* ")")
       <|> mkVal number <|> mkVar ident
```

This version of atom does successfully parse both vectors and parenthesised expressions but will have to read the leading expression twice in the case of a parenthesised expression: this is not ideal, since the expression could have been large. Left factoring has clearly already been employed here to factor out the common "(" to both branches, but it cannot be so easily applied to the inner bits of the parser: certainly, such factoring would make the parser less readable. There is information here that can be exploited, however: parenthesised expressions take priority over vectors, so a single-element vector cannot exist. So, if vectors always have two or more sub-expressions otherwise it is just parenthesised, a bridge can be made to arbitrate between the two:

```
mkParensOrVec :: Parser [LetExpr] ⟶ Parser Atom
mkParensOrVec = liftA disambiguate
  where disambiguate [e]             = OfExpr e
        disambiguate es@(_ : _ : _) = Vec es

atom = "(" *> mkParensOrVec (sepBy1 letExpr ",") <* ")" <|> mkVal number <|> mkVar ident
```

By inspecting the result list, the mkParensOrVec bridge can pick which of the two different nodes it wants to generate: the effect on the atom parser is that the ambiguity is entirely removed, and this parses in a single pass. This technique can be used to resolve a lot of different ambiguities, or even abstract the busywork of constructing the results of manually left-factored grammars. This technique has been put to good use by WACC students at Imperial to factor two of the four ambiguities in WACC's grammar cleanly.

7.2 Integrating Patterns into API Design

The discussion of the patterns in §7.1 focusses on the way that users should be structuring their parsers; but not what library designers should be doing to facilitate these patterns, or indeed how a library can be designed to push users down the route of using the patterns in a natural way.

One obvious way in which library authors should engage with this material would be to simply provide the given definitions from §7.1 in their APIs as is, however, there is more nuance to it than this:

- Many improvements can be made to the ergonomics of the expression parsing functionality discussed in §7.1.1 to either remove boilerplate from the practical use-case or improve the type-inference aspect so that the user avoids writing extra unneeded type ascriptions (§7.2.1).
- The lexical combinators introduced in §7.1.2 can be better abstracted into a configurable framework that abstracts choice away from the users and automatically enforces the desirable lexical conventions. There are pitfalls here that a library author can easily fall into as well, which ideally are avoided (§7.2.2).
- The construction of parser bridges detailed in §7.1.3 entails writing large amounts of systematic boilerplate on the user's end. Identifying the generic portions of bridges and introducing ways to abstractly produce, or generate, them is something a library should strive to provide (§7.2.3).

7.2.1 Implementing Expression Combinators

When writing expression parsers, the best way of abstracting the left-recursive elements is with chain combinators (§7.1.1 and §7.1.1), and precedence hierarchies are best abstracted using precedence combinators (§7.1.1). However, the implementations of this functionality previously provided do not account for the user experience:

- The presentation of precedence in §7.1.1 assumes that the language has a natural support for subtyping when encoding the sops smart constructor: in languages like Haskell this creates the burden of boilerplate on the user, the library can ideally deal with this (§7.2.1).
- In Scala, the uncurried versions of the chain combinators, as shown in §7.1.1 suffer from type-inference issues that should ideally be side-stepped by good API design (§7.2.1).
- In Scala, the lack of existentials to tie together layers of the precedence framework outlined in §7.1.1 would cause an intolerable amount of type-ascription: advanced language features can be used to tackle this problem more elegantly (§7.2.1).

Handling Subtyping in Haskell

When designing parsers, lightweight guarantees about parser correctness can be gained by leveraging more strongly-typed semantic actions: the use of multi-layered hierarchies for encoding a syntax tree for expression parsers can help prevent the parser from deviating from the prescribed associativities and precedence ordering. In a language like Scala, which supports subtyping, this structure follows very naturally:

```scala
sealed trait Expr
case class Add(x: Expr, y: Term) extends Expr
case class Sub(x: Expr, y: Term) extends Expr
sealed trait Term extends Expr
case class Mul(x: Term, y: Neg) extends Term
sealed trait Neg extends Expr
case class Negate(x: Neg) extends Neg
```

```
sealed trait Atom extends Neg
case class Val(n: BigInt) extends Atom
case class Var(v: String) extends Atom
case class Parens(x: Expr) extends Atom
```

However, in Haskell, the lack of subtyping introduces the need for explicit type-casting constructors like OfTerm, OfNeg, OfAtom (§7.1.1): when working with the hierarchy in other parts of the code, these become very obtuse to manage, and writing examples of the data structure is painful, consider the different ASTs for '`6 * 4 + negate 4`' in both Scala and Haskell:

```
// Scala
Add (        Mul (                Val(6),           Val(4)),          Negate (        Val(4)))

-- Haskell
Add (OfTerm (Mul (OfNeg (OfAtom (Val 6))) (OfAtom (Val 4)))) (OfNeg (Negate (OfAtom (Val 4))))
```

There is clearly a lot of extra noise introduced by the wrapping constructors for each level in the Haskell code. Regardless, this situation is improved by the (<) typeclass, introduced in §7.1.1:

```
class 🥚 < 🐔 where
  upcast    :: 🥚 → 🐔
  downcast :: 🐔 → Maybe 🥚
sops :: a < b ⇒ Fixity a b sig → [Parser sig] → Op a b
sops fixity = gops fixity upcast
```

This typeclass allows for the generalisation of the wrapping constructor pattern so that each type can specify its relation to the others. It may be possible to define transitive instances that allow for ease-of-use elsewhere. The problem for the user, however, is having to implement the instances for their own types:

```
instance Term < Expr where
  upcast              = OfTerm
  downcast (OfTerm x) = Just x
  downcast _          = Nothing
```

This is straightforward, but boring. If a parser combinator library wants to support such a convenience, then it would be nice to give the user some assistance in generating these instances. As it happens, Untyped Template Haskell can help with this, allowing the library author to provide a Template Haskell based (<) derivation:

```
deriveSubtype :: Name → Name → Q [Dec]
deriveSubtype 🥚 🐔 = determineWrap 🥚 🐔 >>= deriveSubtypeUsing 🥚 🐔
```

This function should first try and find a Con :: 🥚 → 🐔, and then use this to instruct the instance programmatically, such that the instance Term < Expr above can be more quickly written as:

```
$deriveSubtype ''Term ''Expr
```

When finding wrap constructors to serve as the candidate for upcast, it is important to only proceed if there is a single, unambiguous, constructor that matches the type, otherwise the derivation would have to make choices. To track this, a datatype called WrapCon is given to handle each of three cases:

```
data WrapCon = Single Name | Missing | Duplicates [Name]

instance Monoid WrapCon where
  mempty = Missing
  cs1 • cs2 = fromList (toList cs1 • toList cs2)
```

The good case here is the Single case, which indicates a unique constructor exists, and Missing and Duplicates indicate there is either ambiguity, or no way to convert directly between the two types. The Monoid WrapCon instance provides a convenient way of merging WrapCon values together, such that merging Missing has no effect, and merging anything else results in Duplicates: the fromList and toList functions are omitted for brevity but form a straightforward monoid isomorphism between WrapCon and [Name]. This is used in determineWrap:

```
determineWrap :: Name → Name → Q Name
determineWrap 🥚 🐓 = do    -- precondition: 🥚 ≢ 🐓
  TyConI decl ← reify 🐓
  case decl of
    DataD _ _ _ _ cons _      → findRightCon cons
    NewtypeD _ _ _ _ con _    → findRightCon [con]
    TySynD { }                → fail $
      "type synonym " ++ show 🐓 ++ " cannot be used for Subtype deriving"
    _                         → fail $
      "Subtype derivation only works for data and newtype"
  where
    findRightCon :: [Con] → Q Name
    findRightCon = getWrappedConOrFail · foldMap checkCandidate

    checkCandidate :: Con → WrapCon
    checkCandidate (NormalC  con   [(_, ConT τ)])                | τ ≡ 🥚          = Single con
    checkCandidate (RecC     con   [(_, _, ConT τ)])             | τ ≡ 🥚          = Single con
    checkCandidate (GadtC    [con] [(_, ConT τ)]    (ConT τ′))   | τ ≡ 🥚, τ′ ≡ 🐓 = Single con
    checkCandidate (RecGadtC [con] [(_, _, ConT τ)] (ConT τ′))   | τ ≡ 🥚, τ′ ≡ 🐓 = Single con
    checkCandidate _                                                               = Missing

    getWrappedConOrFail :: WrapCon → Q Name
    getWrappedConOrFail Missing           = fail $ missingConstructor 🥚 🐓
    getWrappedConOrFail (Duplicates cons) = fail $ tooManyConstructors 🥚 🐓 cons
    getWrappedConOrFail (Single con)      = return con
```

The determineWrap function first demands the type information about the desired super-type, 🐓, and then establishes what kind of type it is: a regular data, newtype, or GADT definition is what is required, with type synonyms explicitly ruled out. Then findRightCon searches through the constructors of the type, looking for a constructor with a single argument of the sub-type, 🥚. For GADT constructors, the return type of a candidate constructor must also be checked to ensure it returns 🐔. Any candidate constructors are bundled up using the WrapCon monoid, which will result in a Duplicates if more than one is found. The WrapCon value is then inspected, failing if either no candidate was found, or too many were found, generating errors as follows:

```
missingConstructor :: Name → Name → String
missingConstructor 🥚 🐔 = unwords
  ["type", show 🐔, "does not have a constructor Con ::", pretty 🥚, "->", pretty 🐔]
tooManyConstructors :: Name → Name → [Name] → String
tooManyConstructors 🥚 🐔 cons = unwords
  ["type", show 🐔, "has conflicting constructors that could wrap", show 🥚,
   "namely", conjunct cons]
pretty    :: Name → String
conjunct :: [Name] → String
```

When a candidate wrapping constructor wrap is found, the 🥚 < 🐔 instance itself can be synthesised using standard quasi-quotation with the "declaration" quotes:

```
deriveSubtypeUsing :: Name → Name → Name → Q [Dec]
deriveSubtypeUsing 🥚 🐔 wrap = do   -- precondition: 🥚 ≢ 🐔
  x ← newName "x"                    -- new pattern variable for the downcast
  ⟦d instance
     {-# OVERLAPPING #-} $(conT 🥚) < $(conT 🐔) where
    upcast                     = $(conE wrap)
    downcast $(conP wrap [varP x]) = Just $(varE x)
    downcast _                 = Nothing⟧
```

In the instance, the wrap name is turned into a constructor for the upcast case with the conE combinator and turned into a pattern match constructor with conP in the downcast case. In downcast, a new variable x needs to be bound, so a new name must be generated. This is all the machinery needed to derive instances of the subtyping class (<), removing the work of the user for at least the uniquely determinable cases – if the user has multiple candidates, they will need to specify the instance themselves.

Currying the Chains in Scala

Scala is a language with only local type inference: types can be inferred for local value declarations as well as function return types, but not for parameter types. In addition, the extent to which inferred types of sibling arguments within a function call influence the inference of a parameter is limited. When designing a parser combinator library in Scala, it is important to take this into account, because it can worsen the user experience if they are forced to needlessly ascribe types to satisfy the compiler.

To illustrate the problem, consider the following parser, given `p: Parsley[Int]` and `q: Parsley[Int]`:

```
val r = (p <~> q).map {
    case (x, y) => x + y
}
```

This compiles fine, but it is both a little inefficient, needlessly constructing and deconstructing a pair, and a bit bulky. A user looking around for a better way might stumble across `lift2`, which can combine two parsers with a given function: sounds perfect!

```
val r = lift2(_ + _, p, q) // ✗
```

Unfortunately, the user is met with the following error: `missing parameter type for expanded function ((<x$1: error>, <x$2: error>) => x$1.$plus(x$2))`. Ignoring the mangling of the names inside the error, the compiler is telling us that it does not know what the types of the arguments to the function `_ + _` are supposed to be – even when this is obvious from the types of `p` and `q`. The user would either be forced to write `lift2[Int, Int, Int]` (as Scala does not yet have partial application of type parameters), or would have to write `(x: Int, y: Int) => x + y`: both options are not ideal. The question is: why does this work with `map` but not with `lift2`? The answer is that, with `map`, the receiver of the method's type can be used to immediately fix what the type of the function's argument should be. In this case, `map[B]` only has one type-parameter, so no inference occurs on the argument to the function.

However, a similar trick can be pulled with *currying*, where a single set of arguments is replaced by multiple independent sets. Here is how `lift2` can be adjusted to make the example work:

```
def lift[A, B, C](p: Parsley[A], q: Parsley[B])(f: (A, B) => C) = lift2(f, p, q)

val r = lift(p, q)(_ + _) // ✓
```

In this case, the types of `p` and `q` fix the types `A` and `B`, so that inference is attempted on the next set of parentheses only for the type `C`. Happily, Scala will perform an uncurrying phase that turns the curried `lift` back into one with the same shape as `lift2`, so there are no performance implications for this trick.

The definition of the heterogeneous chain combinators as a direct port of the Haskell-style definition (§7.1.1) runs into the exact same problem as `lift2`:

```
def infixl1[A, B >: A](p: Parsley[A], op: =>Parsley[(B, A) => B]): Parsley[B]

val r = infixl1(p, "+" #> (_ + _)) // ✗
```

Again, in this case it is best to curry the arguments to the chain just like with `lift`[3]:

```
def infixl1[A, B >: A](p: Parsley[A])(op: =>Parsley[(B, A) => B]): Parsley[B]

val r = infixl1(p)("+" #> (_ + _)) // ✓
```

This is an important consideration to make, as the less ascriptions the user must make, the cleaner the resulting parser will look. In general, functions should be kept in argument groupings further to the right, maximising the chance their arguments will not require inference.

[3]Alternatively, this could be offered as an extension method (appendix A.4.3), allowing for `p.infixl1("+" #> (_ + _))` instead.

Avoiding Existentials in Scala

The precedence combinator introduced in §7.1.1 relies on existentials to relate the type of a fixity with the type of an operator within a level. This existential is more evident if the datatype is given in the non-GADT syntax:

data Op a b = ∀sig.Op (Fixity a b sig) (a ⟶ b) (Parser sig)

Here the existential parameter is sig, which is fixed by the specific provided Fixity. This is fine for Haskell, with its support of existentials, but is trickier in Scala, where existentials are either (a) clumsy and verbose or (b) no longer exist, depending on the lanaguage version. The "obvious" fix would be to just shift the type sig to the left-hand side of the definition for Op: this is not ideal, for the same reasons discussed in §7.2.1. Instead, it is possible to leverage one of Scala's more unique features: *path-dependent types*.

Path-Dependent Types In Scala, a path-dependent type [Amin et al. 2016] is a type that is associated with an object, or more generally a *path*, which can be a series of accesses within an object or other scope. Amin et al. [2016] show that all generic parameters to a class can instead be encoded in this form, such that `class Foo[A]` can be encoded as:

```
class Foo { type A }
```

Unlike generic parameters, however, it is possible given a `x: Foo` to write `x.A` as a type – this can then be used in interesting ways as a restricted form of dependent typing. An argument defined in one argument set can have its associated types referred to within the next set.

Writing `Fixity` with Associated Types The way that the Haskell Fixity a b sig type has been defined is that the types a and b are "inputs" to the fixity and combined with the specific constructor result produce sig as an "output." This works because of Haskell's global type inference: the types of the actual operator parsers can be used to infer the type of sig when the specific constructor is known. In Scala, as mentioned in §7.2.1, local type inference, where type information is only propagated across curried argument sets from left-to-right, means that using the same style of type as Haskell will result in an awkward bi-directional type dependency forcing the user to ascribe the types no matter what. Instead, the types `A` and `B` should be decoupled from the `Fixity` itself and instead associated more directly with `Sig`. Couple that with the desire to eliminate `Sig` from the type `Fixity` itself to avoid the existential, and the best translation for the type is:

```
sealed trait Fixity { type Sig[A, B] }
```

In this representation `Fixity` contains an abstract type constructor of kind $* \to * \to *$, which can be saturated with the correct `A` and `B` when they are known. Objects that extend `Fixity` will be required to complete this "type function" definition individually:

```
object InfixL  extends Fixity { type Sig[-A, B] = (B, A) => B }
object InfixR  extends Fixity { type Sig[-A, B] = (A, B) => B }
object InfixN  extends Fixity { type Sig[-A, B] = (A, A) => B }
object Prefix  extends Fixity { type Sig[A, B]  = B => B       }
object Postfix extends Fixity { type Sig[A, B]  = B => B       }
```

Defining `Op` With `Fixity` defined, it is now possible to define an equivalent version of the Haskell Op type without making use of an existential. Instead, the previously existential nature of sig is instead explicitly a type constructor associated `Sig` with the given `Fixity`. This `Sig` can be used within the `Op` by leveraging path-dependent arguments in curried arguments:

```
class Op[A, B](val fixity: Fixity)(val op: Parsley[fixity.Sig[A, B]])(using val wrap: A => B)
```

Given a value `fixity: Fixity` and types `A` and `B`, the path-dependent type `fixity.Sig[A, B]` can only be known statically if the value of `fixity` itself is known statically: when a known value is statically provided, then the compiler will be able to establish what the type is. Type-checking will otherwise assume nothing about the type. So, when using `Fixity` to define a value for `Op`, it is important that a statically known value of type `Fixity` is used – this burden is on the user, however. The use of `using` in the above definition is to allow for the `wrap` function to be determined implicitly by the compiler (appendix A.4.1): this allows for the compiler to automatically find `wrap: A => A` and `wrap: A => B` where `A <: B`, which provides an ergonomic benefit. This `Op` class can be used to define smart constructors `gops`, `sops`, and `ops` in a similar way to in Haskell:

```
def gops[A, B](fixity: Fixity)(ops: Parsley[fixity.Sig[A, B]]*)(using A => B) =
    new Op(fixity)(choice(ops: _*))
def sops[A, B >: A](fixity: Fixity)(ops: Parsley[fixity.Sig[A, B]]*) = gops(fixity)(ops: _*)
def ops[A](fixity: Fixity)(ops: Parsley[fixity.Sig[A, A]]*)          = gops(fixity)(ops: _*)
```

Avoiding `asInstanceOf` Unfortunately, when implementing the `precedence` combinator itself with the `Op` type, it becomes apparent that pattern matching on the `Fixity` does not refine the type of the `op` to be fully known. Scala does not support dependent case classes, where one set of parentheses can support path-dependent types between arguments (see `Issue #8073`), so upon pattern matching on the `Fixity`, the `asInstanceOf` method must be used to cast the operator parser to its real type. This problem, however, can be fixed by associating a new method to `Fixity`:

```
sealed trait Fixity:
    type Sig[A, B]
    def chain[A, B](p: Parsley[A], op: Parsley[Sig[A, B]])(using wrap: A => B): Parsley[B]
```

In this case, each of the five fixities will define the `chain` method in term of the heterogeneous chains. The definition of `precedence` can then be as follows:

```
def precedence[A](lvls: Prec[A]): Parsley[A] = lvls match
case Atoms(atoms: _*) => choice(atoms: _*)
case Level(lvls, lvl) => lvl.fixity.chain(precedence(lvls), lvl.op)(using lvl.wrap)
```

This surprisingly concise definition works perfectly for Scala 3, and Scala 2 with some slight modifications to `Op` since path-dependencies between arguments in class parameters are disallowed.

7.2.2 Implementing Lexical Combinators

The main contributions of §7.1.2 were to introduce a standard convention for whitespace consumption along with a rationale for why it is the right choice (§7.1.2); discussed how to formulate different kinds of primitive tokens (§7.1.2); and demonstrated how to clean up the noise from lexing by using overloaded string literals (§7.1.2).

From a library author's perspective this raises a couple of interesting questions:

- Should I provide implicits to help the user write string recognisers or leave that to them (§7.2.2)?
- If I support a system for configurable lexer combinator generation, does it make sense to have non-lexeme variants of the tokens (§7.2.2)?

To Implicit, or not to Implicit?

In §7.1.2, it was shown that the use of implicits – in the form of Haskell's `OverloadedStrings` extension – could be used to abstract away the lexing logic of the parser and results in something looking closer to BNF. One may wonder if this works just as effectively for dealing with string (or character) parsing before a lexer is developed, in a similar way that parser bridges can be employed before any additional metadata is even required (and the library can provide these generic bridges, see §7.2.3).

In practice, it depends on the language being worked with. In Haskell, a library that provides the user with a "default" IsString instance that just parses string and no more might seem attractive, but actively stops the user from defining their own without resorting to **newtype** hackery. This is especially bad if the instance is not *orphaned*, which is considered best practice: Haskellers tend to want instances to be defined alongside the definition of the relevant type, otherwise it becomes an *orphan instance*, and this may lead to incoherence. However, in this case, the user has no ability to hide the instance if they want to define their own: keeping the instance orphaned in a dedicated module with only the instance is more helpful – but not ideal. So, Haskell libraries should really refrain from implementing such instances, lest they step on the users' toes!

In Scala, however, the implicits mechanism works differently, and while non-orphaned implicits are found more readily by the compiler – improving the performance of implicit search – they are more closely tied to specific scopes, and not a globally coherent system. Importantly, implicits that are orphaned are not imported by default, and require explicit importing to bring into scope. If ambiguities arise between implicits are the same priority, this can be resolved explicitly by the names given to each implicit. The takeaway here is that it is much safer to provide a default implicit conversion, which `parsley` does in the form of `parsley.syntax.character.{stringLift, charLift}`.

Not only can Scala `parsley` provide an implicit conversion for string literals without stepping on the users' toes, but it also can synthesise the implicit conversion for a user using its own `Lexer` framework:

```
abstract class Symbol:
    def apply(sym: String): Parsley[Unit]
    def apply(sym: Char): Parsley[Unit]

    object implicits { given implicitSymbol: Conversion[String, Parsley[Unit]] = apply(_) }

class Lexer(desc: LexicalDesc):
    object lexeme:
        val symbol: Symbol = new ConcreteSymbol(desc.symbolDesc, ...)
        ...
```

In this case, the user-provided `LexicalDesc` is used to instantiate the various fields of the `Lexer` with pre-baked configured parsers for lexing. Part of this is to define an object `lexer.lexeme.symbol.implicits` that provides a scope for an implicit conversion on strings that hooks into the symbol logic: this is exactly the instance provided in §7.1.2. By switching the import from `parsley.syntax.character` to this, the user gets the desired lexing behaviours of string literals for precisely their configured lexing ruleset. This is possible because implicits can be defined as regular first-class constructions, as opposed to the dedicated global machinery within Haskell.

Should Non-Lexeme Tokens Exist?

The previous section alluded to the idea that a parser combinator library can support a configurable lexing framework. The full design of such a framework is out of scope for this dissertation, as it is mostly organisational and mechanical. However, there are a couple of concrete issues that can be discussed: how implicits should be supported by such a framework is one, and whether the lexical convention around whitespace should always be enforced is the other.

It is clear from §7.1.2 that all tokens should be careful to consume trailing whitespace. However, an overzealous library author may believe that, as a result, it is only worth exposing tokens to the user that are guaranteed to consume whitespace. The problem is two-fold:

1. The library author will never be able to implement every possible token for the user. In some cases, compound tokens can be built by concatenating two other tokens without consuming whitespace between. A good example of this might be Scala or Python's string modifiers, where a sequence of characters can be prepended directly onto a string literal, as in `s"$x + $y"`: to achieve this, the prefix characters need to be consumed without consuming whitespace, just before a string literal is parsed.

   Another use-case would be tokens that need to perform some contextual validation: in language where '-' is both a unary operator and part of negative integer literals, there is a difference in meaning between `-10` and `- 10`. In such languages, a `NEGATE` token should be defined as:

   ```
   val NEGATE = lexer.lexeme(atomic(lexer.nonlexeme.symbol('-') *> notFollowedBy(digit)))
   ```

   Here, whitespace cannot be consumed until after the check for a non-trailing digit, but it is still important to make use of the `symbol` logic for ruling out user-defined operators or other larger key operators as well. As a result, a non-lexeme `symbol` should be used, with an explicit `lexeme` around the whole token.

2. There may be other legitimate uses for non-lexeme tokens: one use may be to try and exact a meaningful error token when a parser fails (§8.1.3). In this case, the position before and after a successful token read may be used to specify how wide the error caret should be. If whitespace were consumed when parsing the token, it would result in carets that are far too wide.

7.2.3 Implementing Bridges

To properly decouple the semantic actions from the parser itself, §7.1.3 introduced the idea of *parser bridges*. However, while a variety of techniques can be implemented with these bridges, the act of writing them is boilerplate heavy. In this section, ways of generically deriving this functionality will be explored in both Scala (§7.2.3) and Haskell (§7.2.3).

Generic Bridges using Subtyping

Compared with Haskell, the use of parser bridges in Scala is slightly more boilerplate heavy, but with a cleaner look within the main parser. This is because the use of the `.apply` method can be sugared into looking like regular function application (`f.apply(x)` can be written as `f(x)`), and every class can have a companion object that can define extra `.apply` methods in addition to its regular constructors. As an example:

```
-- Haskell
data LetExpr = Let String LetExpr LetExpr | OfExpr Expr

mkLet :: Parser String -> Parser LetExpr -> Parser LetExpr -> Parser LetExpr
mkLet = liftA3 Let
```

```
// Scala
sealed trait LetExpr
case class Let(v: String, binding: LetExpr, body: LetExpr) extends LetExpr
object Let:
    def apply(v: Parsley[String], binding: =>Parsley[LetExpr], body: =>Parsley[LetExpr]) =
        lift3(Let(_, _, _), v, binding, body)
```

Here, the straightforward partial application and function definition with Haskell makes for a very concise implementation. For Scala, there is a much heavier overhead for the creation of the companion object, and the definition of the bridge itself requires more work. However, the Scala implementation does start to show benefit when used in the parser itself:

```
-- Haskell
letExpr = mkLet ("let" *> ident) ("=" *> letExpr) ("in" *> letExpr) <|> OfExpr <$> expr
```

```
// Scala
val letExpr = Let("let" *> ident, "=" *> letExpr, "in" *> letExpr) <|> expr
```

Here, the structure is simplified in two ways: the first is the lack of wrapping constructor for the `expr` case due to subtyping, and the second is the more consistent form of the bridge without the `mk` prefix (and more obvious argument symmetry). However, the real advantage comes from the definition of the bridge above. By looking at two other bridges, and then desugaring them, some structural similarities can be clearly seen:

```
case class Var(v: String) extends Atom
case class Num(n: BigInt) extends Atom

object Var:
    def apply(v: Parsley[String]): Parsley[Atom] = v.map(Var(_))
object Num:
    def apply(n: Parsley[BigInt]): Parsley[Atom] = n.map(Num(_))
```

These two bridges look like each other, but make use of different types, and different constructors in the `.map` combinator. De-sugaring the methods, however, offers an alternate perspective:

```
object Var:
    def apply(v: Parsley[String]): Parsley[Atom] = v.map(this.apply(_))
object Num:
    def apply(n: Parsley[BigInt]): Parsley[Atom] = n.map(this.apply(_))
```

As it happens, both `apply` methods have exactly the same shape. When using case classes in Scala, the compiler will automatically synthesise an `apply` method on the companion object with the exact same shape as the primary constructor, so that `new Var("hi")` can be written as `Var("hi")`. Since an overloaded `apply` is being defined on the companion object, `Var(_)` can be first desugared to `Var.apply(_)`, and then to `this.apply(_)`. This simple fact provides an opportunity to exploit the shared structure of the bridges and abstract; in fact, the OOP design pattern *Template Method* can be employed here [Gamma et al. 1995]:

> ***Template Method***: *Define the skeleton of an algorithm in an operation, deferring some steps to subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure [Gamma et al. 1995].*

Generic Lifted Constructors The *Template Method* pattern allows for the defining of a configurable *templated method* that is configured in terms of a so-called *hook method*, which is left abstract. The insight is to define a trait that demands the generated `apply` and synthesises the parser bridge from that:

```
trait ParserBridge1[-T1, +R]:
    def apply(x1:         T1):          R                            // The "hook" method
    def apply(p1: Parsley[T1]): Parsley[R] = x1.map(this.apply(_)) // The "template" method
```

The `ParserBridge1` trait concretely describes the shared structure of both the `Var` and `Num` bridges above. It is easy to expand this pattern to work for any number of different arguments: on the JVM, the usual limit would be arity-22. With this trait (and others) in hand, the definitions of all three bridges shown above can be simplified by exploiting the compiler synthesised `apply` method:

```
object Var extends ParserBridge1[String, Atom]
object Num extends ParserBridge1[BigInt, Atom]
object Let extends ParserBridge3[String, LetExpr, LetExpr, LetExpr]
```

Generic Singleton Bridges The next step is to expand the generic behaviours of the `ParserBridgeN` traits so they can also handle the unsaturated singleton bridges (used in precedence combinators). First, it is useful to see what the shape of singleton bridges in Scala looks like:

```
case class Add(x: Expr, y: Term) extends Expr
object Add:
    def <#(op: Parsley[_]): Parsley[(Expr, Term) => Expr] = op #> this.apply(_, _)
```

Compared with Haskell, where a use of such a bridge looks like mkAdd <* "+", in Scala, it is implemented with a combinator on the object itself: `Add <# "+"`. It might be immediately tempting to pull a similar trick to `ParserBridgeN` and define a bunch of `ParserSingletonBridgeN` traits, each providing a `<#` combinator. However, it is possible to hook into the existing `ParserBridgeN` traits instead:

```
trait ParserSingletonBridge[+A]:
    def con: A                                        // the "hook" method
    final def <#(op: Parsley[_]): Parsley[A] = op #> con // the "template" method

trait ParserBridge2[-T1, -T2, +R] extends ParserSingletonBridge[(T1, T2) => R]:
    def apply(x1:         T1,  x2:         T2):          R
    def apply(p1: Parsley[T1], p2: Parsley[T2]): Parsley[R] = lift2(this.con, x1, x2)
    override final def con: (T1, T2) => R = this.apply(_, _)
```

This means that, like the other bridges, the `Add` bridge just needs to extend `ParserBridge2[Expr, Term, Expr]` and gets both fully saturated and unsaturated application patterns with no extra boilerplate.

There is one special case for the singleton bridges that needs to be accounted for. This is the case where the result itself is already a singleton object:

```
trait ParserBridge0[+R] extends ParserSingletonBridge[R] { this: R =>
    override final def con: R = this
}
```

This makes use of a *self-type*, stating that whomever implements this trait must themselves have type `R`. An example of this might be a bridge to represent the parsing of the Haskell value `()`: `case object HsUnit extends ParserBridge0[HsUnit]`[4], which can now be parsed with `HsUnit <# "()"`. This is important to allow seamless transition between these generic bridges and ones which, for instance, handle position information.

Generic Position-Tracking Bridges Before discussing what bridge support a library author should consider adding to their library, it is instructive to see the translation of the bridges from §7.1.3 into this generic scheme:

```
trait ParserSingletonPosBridge[+A]:
    def con(pos: (Int, Int)): A
    final def <#(op: Parsley[_]): Parsley[A] = pos.map(this.con(_)) <* op

trait ParserBridgePos2[-T1, -T2, +R] extends ParserSingletonPosBridge[(T1, T2) => R]:
    def apply(x1: T1,  x2: T2)(pos: (Int, Int)): R
    def apply(p1: Parsley[T1], p2: Parsley[T2]): Parsley[R] =
        lift3[(Int, Int), T1, T2, R](this.con(_)(_, _), pos, p1, p2)
    override final def con(pos: (Int, Int)): (T1, T2) => R = this.apply(_, _)(pos)
```

Simply, the singletons are adapted by adding a position argument to the `con` method and mapping onto the `pos: Parsley[(Int, Int)]` combinator to fill it. For the saturated application, the position is parsed just ahead of all the other arguments, as usual, and applied to `con` within the `lift3`.

A design decision had to be made here: in particular, `pos: (Int, Int)` was added as a curried argument to the `apply` hook. The reason for this is because in Scala the second set of brackets in a case class is not considered for pattern matching or equality, which makes it an ideal place to store the metadata. However, this is only one possible design choice that could be made: ideally this choice should not be imposed onto the users.

For a library, it is valuable to include the plain generic bridges, including the `ParserSingletonBridge` and all the other arities. This gives users a chance to start using the *Parser Bridge* pattern early on in their parser development without implementing the boilerplate themselves. Then, when they are ready to start interacting with their own metadata, they can feel free to develop their own more specialised generic bridge traits.

[4]In practice, this is not legal, as it is a cyclic reference, but `ParserBridge0[Expr]` works.

Verifying and Disambiguator Bridges The other two kinds of bridges, the *Verifying* (§7.1.3) and *Disambiguator Bridges* (§7.1.3) can often be cleanly implemented by overriding a single part of the generic bridges:

```
case class VarNoun(noun: String)
object VarNoun extends ParserBridge1[String, VarNoun]:
    val nouns = Set("bee", "apple", "car", "dog")
    override def apply(p: Parsley[String]) = super.apply(p).filter(v => nouns(v.noun))
```

This bridge verifies a constraint that variables must be nouns, which can be done by overriding the template `apply` so that it calls the original template and then filters to enforce the constraint. In this case, the hook `apply` can be retained from the companion object. By hooking into the generic machinery, the logic of piecing together the data itself is avoided – including metadata collection – though unsaturated application will not do verification.

```
object ParensOrVec extends ParserBridge1[NonEmptyList[LetExpr], Atom]:
    def apply(es: NonEmptyList[LetExpr]): Atom = es match
    case NonEmptyList(e, Nil) => Parens(e)
    case es                   => Vec(es)
```

In this example, the `ParensOrVec` bridge relies on the generic machinery to provide the template method but provides an implementation of the hook to perform the disambiguation. Again, this retains the unsaturated application, and avoids any meta-data collection parsing logic, which is still handled by the generic bridge.

However, the verified bridge presented in (§7.1.3) involving bounded integer literals cannot be so cleanly ported over because it does a simultaneous mapping and filtering operation. This would have to be implemented in full, with both the hook and the template.

Template Haskell for Bridge Generation

In Scala, traits can be used to hook into compiler generated functionality and cleanly implement different forms of bridge in a unified way (§7.2.3). In Haskell, though, the situation is different, as there is no compiler generated code to hook into, and the equivalent abstraction to traits, type-classes, cannot define instances on a per-constructor level, because constructors themselves are not their own types. Instead, as in §7.2.1, Untyped Template Haskell can be used to programmatically generate the lifted and deferred constructors for a datatype. Unlike in Scala, it is more feasible to also support the position handling metadata with this approach without imposing too many constraints on the user. Given its size and complexity, some of the definitions will be omitted for brevity, however, the full code can be found in appendix E.

To handle position information in the constructor, it needs to be the case that the templating has a notion of position to latch onto. The user can specify this type, but then they would also have to specify the combinator to produce it. For simplicity, the library itself exposes a simple type alias Pos:

```
type Pos = (Int, Int)
```

If this is used, then the templating can pick it up and integrate the position information into it; if they do not, then a regular lifted constructor would be produced, and the user may have to implement the right constructor for themselves if there is metadata to incorporate. If a unique Pos is found within a constructor, then the following combinator can be used to change the generated bridge into one that handles the position:

```
posAp :: Bool → Q Exp → Q Exp
posAp False p = p
posAp True  p = ⟦e pos <**> $p⟧
```

Generating either type of bridge requires two steps: the first should deconstruct the type information of the chosen constructor finding out all the relevant information, including where a Pos can be found – if any; and the second step synthesises the function by reshuffling the parts, applying the posAp combinator if necessary.

Deriving *Lifted Constructors* A *Lifted Constructor* is a function that takes several parsers as arguments, sequencing them and combining them with a specific constructor. They can be derived by providing a prefix string, which should be prepended to the constructor's name to form the bridge name, and a constructor to analyse – for convenience this can be provided as a list of constructors to allow for simultaneous synthesis:

```
data Foo a = Foo Pos a | Bar Int String
  -- makes mkFoo :: Parser a → Parser (Foo a) and mkBar :: Parser Int → Parser String → Parser (Foo a)
deriveLiftedConstructors "mk" ['Foo, 'Bar]
```

By calling deriveLiftedConstructors at the top-level, the functions are generated and put into scope for use, with the expected types. If there is a single Pos present, it does not appear in the generated signature, as expected.

```
deriveLiftedConstructors :: String → [Name] → Q [Dec]
deriveLiftedConstructors prefix = fmap concat · traverse deriveCon
  where
    deriveCon con =
      do (con′, τ, func, posFound, nargs) ← extractMeta True prefix (funcType · map ofParser) con
         args ← replicateM nargs (newName "x")
         sequence [sigD con′ τ
                  , funD con′ [clause (map varP args)
                      (normalB (posAp posFound (applyArgs ⟦e pure $func⟧ (map varE args)))) []]
                  ]
    applyArgs :: Q Exp → [Q Exp] → Q Exp
    applyArgs = foldl′ (λrest arg → ⟦e $rest <*> $arg⟧)
```

The deriveLiftedConstructors synthesises definitions for each provided constructor in turn, then flattens that down into a single list of declarations: this can then be spliced into the top-level to produce the definitions. The deriveCon function is then responsible for each individual constructor: it first needs to extract the meta-data about the constructor with the function extractMeta:

```
extractMeta :: Bool → String → ([Q Type] → Q Type) → Name → Q (Name, QType, Q Exp, Bool, Int)
```

The definition of this function is discussed later, but effectively, the first argument says whether a position is at the start or the end of the function (True for lifted constructors and False for deferred ones), the second is the prefix, the third is how to adapt the constructor type to the bridge type, and the fourth is the constructor itself. It returns the name of the bridge, its type, the position-aware lambda form of the constructor, whether a Pos was found, and then the arity of the function. For lifted constructors, the way to transform the type of the constructor into the bridge's type is to first map Parser across every type component before folding into a function type, done with the following simple combinators:

```
funcType = foldr1 (λτ σ → ⟦t $τ → $σ⟧)
parserOf τ = ⟦t Parser $τ⟧
```

With the metadata obtained, new variable names are created for each of the bridge's arguments, then a list of type declaration and definition is returned, with the body of the bridge constructed by applying the lambda representing the constructor to each argument in turn using (<*>).

Deriving *Deferred Constructors* A *Deferred Constructor* is a function that takes no arguments and returns a parser that results in a function. They are derived similarly to lifted constructors:

```
deriveDeferredConstructors :: String → [Name] → Q [Dec]
deriveDeferredConstructors prefix = fmap concat · traverse deriveCon
  where
    deriveCon :: Name → Q [Dec]
    deriveCon con =
      do (con′, τ, func, posFound, _) ← extractMeta False prefix (parserOf · funcType) con
         sequence [sigD con′ τ
                  , funD con′ [clause [ ] (normalB (posAp posFound ⟦e pure $func⟧)) [ ]]
                  ]
```

The main differences are the way that the type of the bridge is constructed – in this case, the function type is built before wrapping it in the Parser type – and how the body is constructed, which in this case just uses pure of the constructor's lambda, with a (pos <**>) applied to the front if required.

Extracting the Metadata The construction of both kinds of bridge require various pieces of information to be extracted from the type of the given constructor. This is done via the extractMeta function, the arguments to which described above. The definition is as follows:

```
extractMeta :: Bool → String → ([Q Type] → Q Type) → Name → Q (Name, QType, Q Exp, Bool, Int)
extractMeta posLast prefix buildType con = do
  DataConI _ τ _ ← reify con
  (forall, τs)     ← splitFun τ
  posIdx           ← findPosIdx con τs
  let τs′           = maybeApply deleteAt posIdx τs
  let nargs         = length τs′ − 1
  let func          = buildLiftedLambda posLast con nargs posIdx
  return (mkName (prefix ++ pretty con), forall (buildType (map pure τs′)), func, isJust posIdx, nargs)
```

First, the information about the constructor is retrieved using the reify operation in the Q monad, and the type signature is split up into the individual types as well as a function that can reconstruct the type binders if there was ∀· quantification on the constructor. If a position can be found within the types τs, then it should be deleted from those types as it will not form part of the type of the generated bridge. A lambda function is built to apply the constructor, with the position argument – if necessary – placed either first or last to match the bridge's requirements. Finally, the information is all combined and returned. There are three helpers used for this:

```
splitFun :: Type → Q (Q Type → Q Type, [Type])
splitFun (ForallT bndrs ctx τ) = return (forallT bndrs (pure ctx), splitFun′ τ)
splitFun τ                     = return (id, splitFun′ τ)
  where splitFun′ (AppT (AppT ArrowT τ) σ)               = τ : splitFun′ σ   -- regular function type
        splitFun′ (AppT (AppT (AppT MulArrowT _) τ) σ) = τ : splitFun′ σ   -- linear function type
        splitFun′ τ                                       = [τ]
```

The splitFun function is tasked with matching function (and linear function) types and collecting up each of the parameter types as well as the final return type. This is done by straightforward recursion on the type[5].

```
findPosIdx :: Name → [Type] → Q (Maybe Int)
findPosIdx con τs = case elemIndices (ConT ''Pos) τs of
  [ ]    → return Nothing
  [idx] → return (Just idx)
  _      → fail $ unwords   -- more than 1 index, which is ambiguous
    ["constructor", pretty con, "has multiple occurrences of Pos"]
```

The function findPosIdx looks through a list of types to see if the special type Pos can be found. This is done by collecting all the indices of the occurrences of Pos: if there is a unique index, this can be factored out; otherwise, there is no position, which is fine; or there are too many and this is an error.

```
buildLiftedLambda :: Bool → Name → Int → Maybe Int → Q Exp
buildLiftedLambda posLast con nargs posIdx = do
  args    ← replicateM nargs (newName "x")
  posArg  ← newName "pos"
  let pargs  =  if | isNothing posIdx → map varP args
                   | posLast          → map varP args ++ [varP posArg]
                   | otherwise        → varP posArg : map varP args
  let eargs  =  maybeApply (flip insertAt (varE posArg)) posIdx (map varE args)
  lamE pargs (foldl′ (λacc arg → ⟦$acc $arg⟧) (conE con) eargs)
```

To construct the final lambda to lift across the parsers in the bridge, buildLiftedLambda first creates all the required new names, and, if required, adds in the position argument pos into them into the right place. To construct the body of the lambda, the position is inserted at the correction position for the constructor and the argument list is folded up into a nested application. The freshly created lambda is returned.

[5]In practice, it is important to also check whether `KindSignatures` is enabled: if not, TH will generate non-compiling bound variables bndrs – as they will each be given a concrete kind signature – and the added kind-signatures must be removed (see appendix E).

Discussion

Throughout this chapter several patterns have been presented, along with implementations of niceties that can be provided by a library's API to help facilitate them. Given some datatypes, the library can derive machinery:

```
data LetExpr = Let String LetExpr LetExpr Pos  | OfExpr  Expr
data Expr    = Add Expr Term | Sub Expr Term   | OfTerm  Term
data Term    = Mul Term Neg                    | OfNeg   Neg
data Neg     = Neg Neg                         | OfAtom  Atom
data Atom    = Val Integer | Var String Pos    | Parens  LetExpr

$(deriveSubtype ''Expr ''LetExpr)
$(deriveSubtype ''Term ''Expr)
$(deriveSubtype ''Neg ''Term)
$(deriveSubtype ''Atom ''Neg)
$(deriveSubtype ''LetExpr ''Atom)

$(deriveLiftedConstructors "mk" ['Let, 'Val, 'Var, 'Parens])
$(deriveDeferredConstructors "mk" ['Add, 'Sub, 'Mul, 'Neg])
```

This makes use of the functionality from §7.2.1 and §7.2.3 to generate the boilerplate code to use both bridges, including position information, from §7.1.3; and also the subtyped precedence machinery from §7.1.1. This has alleviated the burden on the user and allows them to adjust their implementation with only small, if any, changes. Lexing logic is kept separate from the main parser, meaning that whitespace handling and tokenization can be cleanly varied without intruding on the main parsing logic.

```
letExpr = "let" *> mkLet ident ("=" *> letExpr) ("in" *> letExpr) <|> upcastP expr
expr    = precedence $ sops InfixL [mkAdd <* "+", mkSub <* "-"] +<
                       sops InfixL [mkMul <* "*"]               +<
                       sops Prefix [mkNeg <* "negate"]          +<
                       Atom atom
atom    = "(" *> mkParens letExpr <* ")" <|> mkVal number <|> mkVar ident
```

The parser itself leverages several patterns: the *Overloaded Strings* pattern (§7.1.2) is used, keeping the lexing of symbols cleanly abstracted whilst handling whitespace and keyword concerns; the *Precedence Combinators* pattern (§7.1.1) is being employed to cleanly handle left-recursion in the grammar, as well as providing a clear, type-safe, description of precedence ordering and associativity of operators; and the *Parser Bridge* pattern (§7.1.3) is used to abstract the handling of position information away from the parser and reduce "operator noise" that would otherwise arise from the use of the (<$>) and (<*>) combinators.

The category of patterns not discussed in this chapter are those that concern error messages, which is deferred to CHAPTER 8: IMPROVING ERROR MESSAGES. This is because these patterns can be better discussed within the context of `parsley`'s error system and combinators. As with some of the patterns found here, some of these new patterns can benefit from library level support.

In the context of the library developed in Chapter 3: Embedding a Parser Combinator Library, the bridge patterns have a new role to play. The api of a staged parsing library naturally comes with the burden of interacting with the meta-language and, in this case, the additional machinery introduced by Chapter 6: Analysis and Optimisation as well. This can reduce the usability of the library for the casual user, however, the parser bridges help to mitigate this by hiding the Template Haskell noise behind the bridge; furthermore, the machinery developed in §7.2.3 can be easily adapted to relinquish the user of the burden entirely – at least until they have to do non-trivial work, by which point they should have found their feet.

Related work on parsing design patterns and how they arise in other parsing tools is discussed in Chapter 9: Related Work, as well as how design patterns impact parsing tools in general.

Chapter 8

Improving Error Messages

So far, the discussion of this dissertation has been focused on the design and implementation of Haskell `parsley`. For this chapter, however, the focus shifts to Scala `parsley` to discuss how error messages generated by parser combinators can be improved. While this error system will be ported back to Haskell, and staging applied to it to statically refine error messages, this remains as time-consuming, but straightforward, future work.

The error system `parsley` adopts is an iteration over `megaparsec`'s error system, which in turn is an improved version of the original `parsec` system. As discussed in §2.3.1, Partridge and Wright [1996] describe how error messages for parser combinators can be improved by only reporting errors at the deepest position, and aggressive pruning of alternatives that are known not to succeed help improve the error process – this is countered by using atomic to allow (<|>) to backtrack. To justify more concretely why the absence of the atomic combinator is important to help error messages, here is a common mistake made when writing a parser for Imperial's WACC:

```
val func = Func(ty, ident, "(" *> sepBy(argument, ",") <* ")", "is" *> stmt <* "end")
val decl = Decl(ty, ident, "=" *> expr)
val stmt = ... <|> decl <|> ...
val prog = Prog(many(atomic(func)), many(stmt))
```

Here, both `func` and `stmt` (via `decl`) share a common leading prefix, namely a type and an identifier. The WACC grammar states that all functions must come before any statements are parsed; the first statement, however, will be perceived as the start of a function, so `atomic` used here to ensure that the parser fails out of `func` without consuming input to proceed to the statements. This will result in a parser that will correctly parse valid input as well as reject invalid input, but the error messages that may occur from this are sometimes uninformative and misleading. Consider the following input:

```
begin int f(int x, int) is ...
```

This function has a problem in that the last parameter is missing its name; however, the parser outlined above will provide the following error:

```
Syntax error (1, 12):
  unexpected opening parenthesis
  expected "="
  |begin int f(int x, int) is ...
  |           ^
```

This error instead blames the open parenthesis for the problem, suggesting that an assignment is needed. This is because the backtracking allows the `many` to succeed, which discards the good error generated by `func`. The declaration now generates a poor error in its place. The fix is to limit the scope of the `atomic` so that it does not backtrack past the opening parenthesis: if the parser gets past this point, it is unambiguously parsing a function and never a declaration. The point here is that over-use of `atomic` can diminish the effectiveness of errors, even if it does not make a parser incorrect.

The purpose of this chapter is: to describe `parsley`'s error system and new combinators, and how they help to control the errors (§8.1); to outline how this can be implemented in a fast way that minimises unnecessary work (§8.2); and to demonstrate a new set of design patterns for helping construct informative errors (§8.3).

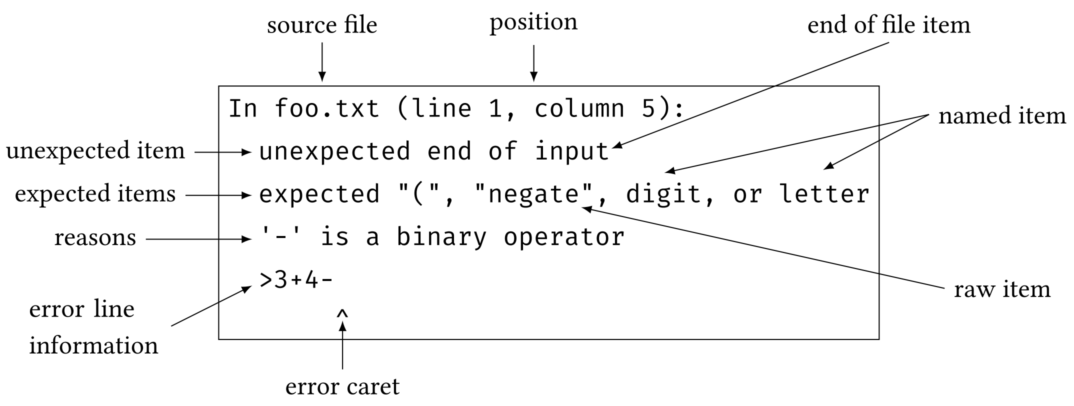
8.1 Error System Design

The error system of `parsley` is inspired by the systems of `megaparsec` and `parsec` before it, but with a richer set of primitives and less manual control over the error construction. Several of the combinators presented in this section may be implementable using `megaparsec`'s `observing` combinator; however this combinator necessitates the exposing of the internal error system to the user-facing API, a decision I have deliberately avoided to help ensure that my system can continue to evolve and be optimised without impacting source or binary compatibility for my users. However, this does not mean that my system is any less flexible: in fact, the core set of combinators has been easy to understand and use for our students at Imperial undertaking the WACC compilers project, who anecdotally seem to have found more success taming the errors of `parsley` than those of `megaparsec`.

Error kinds There are two kinds of error within `parsley`'s system: *vanilla* errors, and *specialised* errors. The vanilla errors are those that arise from regular use of the library and have several core components: an *unexpected* message, a set of *expected* labels, and a set of *reasons* (fig. 8.1). On the other hand, a specialised error simply has an ordered list of raw messages (fig. 8.2): these errors arise from specific user intervention, though the user can control and create vanilla errors too. Both kinds of error then additionally carry with them the position of the error, contextual lines of input surrounding the error, and have an error caret to highlight the error with a non-fixed length. The error system in `parsec` solely maps to the vanilla errors within `parsley`: the messages introduced by `parsec`'s `fail` combinator correspond more precisely to the reasons within a vanilla error, as opposed to messages within a specialised one[1]. However, the `megaparsec` system also distinguishes between *trivial* and *fancy* errors but does not have any notion of reasons within its trivial errors, nor does it have fine-grained control over the unexpected tokens themselves. The fancy errors of `megaparsec` also support custom error types, but this is avoided in `parsley` as to not introduce extra type parameters: in Scala, this is more likely to be annoying than it is helpful given the lack of global type inference or partial type application.

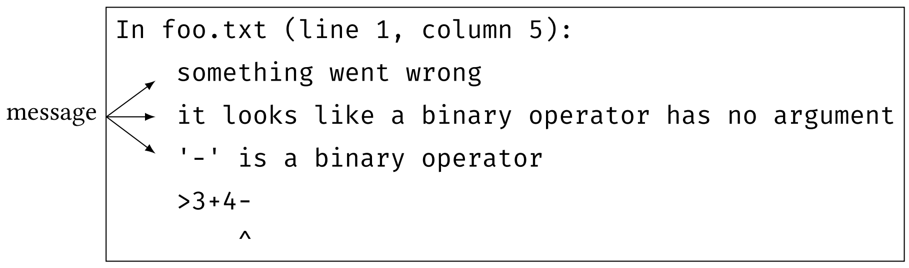
Error hinting Partridge and Wright [1996] suggest that to generate good error messages within parser combinators, a library should report the error that happens at the deepest offset. However, this does not necessarily mean that the only error messages that are relevant are those that arise from sibling `<|>` combinators occurring at the same offset. Given `p *> q`, say, errors from `p` may be relevant and at the same offset as those generated within `q`; this is the case when `p` succeeded having not consumed input, but where there was some internal failure. Consider the parser `optional('+') *> natural`, which should read natural numbers preceded by an optional plus sign; if the input `"a"` is given, it is logical to report an error suggesting "`expected "+" or natural`." To achieve this, when a vanilla error is recovered from by another branch, this error will become a *hint* (a term borrowed from `megaparsec`), carrying some of the information of the error with it: this information may

[1] In fact, `fail "error message"` in `parsec` is the equivalent to `empty.explain("error message")` in `parsley`, a combinator defined later.



```
In foo.txt (line 1, column 5):
  unexpected end of input
  expected "(", "negate", digit, or letter
  '-' is a binary operator
  >3+4-
      ^
```

Fig. 8.1: Structure of a *vanilla* error.



```
In foo.txt (line 1, column 5):
  something went wrong
  it looks like a binary operator has no argument
  '-' is a binary operator
  >3+4-
      ^
```

Fig. 8.2: Structure of a *specialised* error.

get incorporated into a future vanilla error if the new error occurs at the same offset that the hint was registered at.

Observable input consumption Deviating from both `parsec` and `megaparsec`, `parsley` defines two different ways of viewing input consumption: the input that the parser has consumed at any given point during the parse is independent from the *observably consumed* input seen in an error (called the *presentation offset*). To be concrete, the `atomic` combinator rolls back input consumption so that when a parser fails, it fails having consumed no input. This is important for interactions with `<|>`, but the error message will still be pointing at the deepest point – this is not the same position! Combinators that interact with error messages must all work with observable input consumption because that is where the error will be interpreted from. To see why this distinction is important, consider the following parser implemented using `megaparsec`, which does not make this distinction:

```
negInt = try (char '-' *> (negate . read <$> some digit)) <?> "negative number"
```

This parser tries to read negative numbers, with a required leading minus sign. For whatever reason – there is another kind of token starting with a minus, say – this parser has been marked with `try`, `megaparsec`'s `atomic`. This is problematic, as the label combinator, `<?>`, applies a label only when input has not been consumed and it is outside of the `try`. To illustrate how this is bad, consider parsing on the input `"-"`:

```
1:2:
1 | -
  |  ^
unexpected end of input
```

```
 expecting negative number
```

The error message claims that the parser failed because there is no more input, fine, and that after a minus sign, a negative number is expected: this is wrong! Indeed, the fix here is to invert the ordering of the `try` and the `<?>` so that the label is applied (or rather, ignored) before input consumption is rolled back. While this seems benign, this might start having problems in a parser with shape `(try p <|> q) <?> label`, where again the `try` and `<?>` need to be inverted resulting in `try (p <?> label) <|> (q <?> label)`: this transformation duplicates the label in two places. In contrast to this system, `parsley` will only apply labels when the parser has not observably consumed input, meaning that in this case, since the user sees input has been consumed since negative number started parsing, the label is not applied. This section discusses various aspects of the system as outlined above.

8.1.1 Error Primitives

There are five primitive combinators that can produce errors: `empty`, `satisfy`, and `notFollowedBy` (look_), which have been seen already; and two new primitives, `fail`, and `unexpected`.

`empty` Recall that `empty` is the zero for `<|>`, so that `empty <|> p = p = p <|> empty` (eqs. (2.8) and (2.9)); for this to be true, it must be the case that `empty` has no effect on the error messages generated by `p`, nor can errors that arise from `empty` merge with another error and alter it in any way. As such, `empty` produces a contentless error message, such that `empty.parse("abc")` yields the error in fig. 8.3a.

`satisfy` The `satisfy` combinator can also fail, and it does so by interacting with the unexpected component of a vanilla error message. Namely, it will produce the next character as a *raw* unexpected item (denoted by the quotes), or “end of input” should one not exist. This can be seen in fig. 8.3b, with `satisfy(_ == 'x').parse("abc")`; on its own, this is not a particularly informative combinator, though there is no information available to the library to refine this further. The error caret for `satisfy`’s errors is one character wide; it is possible to also make `string` a primitive, so that it can benefit from a wider, multi-character caret spanning the entire width of the string.

`notFollowedBy` The `notFollowedBy` combinator also interacts with the unexpected component of a vanilla error message. It will produce a raw unexpected item that is as wide as the input that its argument parsed successfully. If it consumes no input, it will not produce an unexpected item: this makes it equivalent to `empty`, as expected by eq. (2.30). The error from `notFollowedBy(item *> item).parse("abc")` is shown in fig. 8.3c.

```
(line 1, column 1):
  unknown parse error
  >abc
```

(a) `empty`

```
(line 1, column 1):
  unexpected "a"
  >abc
   ^
```

(b) `satisfy(_ == 'x')`

```
(line 1, column 1):
  unexpected "ab"
  >abc
   ^^
```

(c) `notFollowedBy(item *> item)`

Fig. 8.3: Errors generated by primitive combinators

```
(line 1, column 1):
  unexpected number
  >123
   ^
```

(a) `unexpected("number")`

```
(line 1, column 1):
  unexpected number
  >123
   ^^
```

(b) `unexpected(2, "number")`

Fig. 8.4: Errors generated by `unexpected`

New Error Producing Primitives

The three primitives seen so far have only been capable of producing *raw* or *end of file* unexpected items. This does not account for *named* items, which do not have quotes and have a specific programmer assigned name. The primitives so far have only interacted with the *vanilla* error system, with *specialised* errors being ungeneratable.

`unexpected` The `unexpected` combinator unconditionally fails, citing the given name as the unexpected *named* item. An example of `unexpected("number").parse("123")` can be seen in fig. 8.4a. However, the context for the `unexpected` may be wider than a single character, but there is nothing the library could itself divine about that; instead, an optional integer argument can specify the width of the caret manually. An example of this is given in fig. 8.4b, where the caret width is specified as 2 characters wide.

`fail` The `fail` combinator is the only means to generate specialised errors. It takes a variadic number of string messages and incorporates them in order into the error message. When two specialised messages merge, they will append their messages together in the order of generation: this means that `fail(msg1) <|> fail(msg2)` is the same as `fail(msg1, msg2)`. An example of an error generated by `fail` can be seen in fig. 8.2, which may have arisen from a single three-argument `fail`, or the merging of two or more `fail`s. Like `unexpected`, `fail` also supports an optional argument to control caret size: the implications of specifying caret width for both combinators is discussed in §8.1.2.

Interacting with Expected Items and Reasons

Up until this point, the combinators discussed (except for `fail`) only influence the unexpected component of a vanilla error message. This does not create the most informative error messages, and further primitives are needed to continue to refine the errors. The final two primitive combinators each work with one of the two untouched components, allowing for the construction of full and descriptive error messages.

`label` The `.label` combinator is the most used auxiliary primitive error combinator. Labelling an error means one of two things: if the parser inside fails, having not observably consumed input, one or more given labels are

```
(line 1, column 1):
  unexpected end of input
  expected any character
  >
   ^
```

(a) `item.label("any character")`

```
(line 1, column 1):
  unexpected "abc"
  expected "xyz"
  >abc
   ^^^)
```

(b) `string("xyz")`

```
(line 1, column 1):
  unexpected "a"
  expected end of input
  >abc
   ^
```

(c) `eof`

Fig. 8.5: Errors generated by labelling

added as a named expected items and all the items below it are suppressed; if the parser succeeds, `label` may rename any hints that occurred as part of the parser. An example of `label` can be found in fig. 8.5a.

By default, the library tags as many parsers as it can with labels, such as `digit`, `letter`, `char`, `string`, and so on. However, `label` only ever introduces named items, which means there is no way of generating "end of input" or raw items for the error messages. Even though the "look" of these items can be mimicked by using `.label("end of file")`, or `.label("\"" + str + "\"")`, say, these are not distinguishable for the sake of formatting later (§8.1.3). Instead, there is a benefit to making `eof`, `char`, and `string` primitives for the sake of introducing these labels. Examples of these are shown in figs. 8.5b and 8.5c.

`hide` The combinator `.hide` will suppress any errors that occur from the parser that it is called on; this means that errors generated from the parser will behave as if they were generated by `empty`, and any hints are discarded. Like `label`, this works based on the observably consumed input from the error. This has several uses, including the patterns identified in §8.3, and commonly to hide whitespace consumption: apart from very whitespace sensitive grammars, it is rarely useful to suggest adding whitespace as a problem fix.

`explain` The errors observed in fig. 8.5 are already more informative than those seen in fig. 8.3; however, even good labelling can sometimes struggle to capture the nuances of a particular part of a grammar. To this end, the `.explain` combinator can be used to attach descriptive messages to a particular portion of the grammar. Like `label`, `p.explain(reason)` will only apply the given reason if `p` fails having not observably consumed input. This is equally important for reasons as it was for labelling, where explanations posed at the wrong offset can be far more confusing than clarifying. Unlike `parsec`'s `fail` combinator, `explain` attaches to an existing error, while providing much the same benefits via `empty.explain(msg)`, which is analogous to `fail msg`.

As an example of how the `explain` combinator can help clarify errors, consider the following simple parser for string literals[2]:

```scala
import parsley.implicits.character.charLift // enables overloaded character literals (§7.1.2)
val escapeChar = choice('n' #> '\n', 't' #> '\t', '\"', '\\')
val strLetter = noneOf('\"', '\\').label("string char")
            <|> ('\\' *> escapeChar.label("escape char"))
val strLit = ('\"' *> stringOfMany(strLetter) <* '\"'.label("end of string")).label("string")
```

This parser allows for four different escape characters, which must be preceded by a backslash; if no backslash is present it allows for any non-backslash or non-quote character until the end of the string. The labelling performed on this parser does provide some effective error messages, see fig. 8.6a; however, the error generated for a bad

[2]Note that `#>` is the equivalent of `($>)`, since $ is a reserved character in Scala.

```
(line 1, column 2):
  unexpected end of input
  expected end of string, escape char, or string char
  >"
    ^
```

(a) `strLit.parse("\"")`

```
(line 1, column 3):
  unexpected "a"
  expected """, "\", "n", or "t"
  >"\a
     ^
```

(b) `strLit.parse("\"\\a")`

Fig. 8.6: Errors generated by `strLit`

escape character in fig. 8.6b is not particularly informative. A naïve attempt at improving this error might be to add `.label("\\n")`, and so on, to each of the possible escapes, but `expected \", \\, \n, or \t` is not accurate when the leading backslash has already been consumed. Instead, one might consider using `.label("end of escape sequence")`: this would be more informative but does not explain what that means to someone unfamiliar with the terminology. In fact, the `explain` combinator can complement this much more effectively:

```
val escapeChar = choice('n' #> '\n', 't' #> '\t', '\"', '\\')
                   .label("end of escape sequence")
                   .explain("valid escape sequences include \\n, \\t, \\\", or \\\\")
```

This parser would instead give the following error message, which makes the reason for the failure, and the possible fix, much clearer:

```
(line 1, column 3):
  unexpected "a"
  expected end of escape sequence
  valid escape sequences include \n, \t, \", or \\
  >"\a
     ^
```

In this case, the reader of the error is informed that the problem is that they have not ended the escape sequence in an appropriate way, and furthermore told what valid escape sequences look like: this is not misleading them into thinking that another backslash would be required, as the label clears that up. The reasons generated by `explain` cascade, so that an extra `.explain("invalid escape sequence")` could be added if the parser writer feels that would be useful: this would provide another line of error message.

8.1.2 Error Control

Section 8.1.1 introduced combinators that can generate and influence the content found within an error message produced by `parsley`. However, these combinators cannot influence the position that an error occurs at: this can be useful, either to build new filtering combinators with good error messages, or to help fine-tune where an error happens as to provide more informative errors.

The `Amend` Combinator

The first control combinator is `amend`; `amend(p)`, should `p` fail, will shift the error offset and position such that it appears that the error happened on entry to the `amend` combinator. Effectively, this adjusts the observed input consumption, so that an error appears to have occurred sooner than it actually did.

```
(line 1, column 2):
  unexpected "\"
  expected escape char or string char
  >"\a
    ^
```

(a) with `amend`

```
(line 1, column 2):
  unexpected illegal escape sequence
  expected escape char or string char
  >"\a
    ^^
```

(b) with `unexpected` and `amend`

Fig. 8.7: Effect of `amend` on `strLit`

When an amendment is performed on an error, the original point of error is overwritten; this means that errors that may have dominated when merging with errors from other branches may now either merge or themselves be dominated. Often this is a desirable property, as the errors appear to have occurred at the same positions, however, this may not be desirable if the error is more descriptive and backtracking is possible.

The `amend` combinator can be used to make parts of the parser atomic for the purposes of error messages in a straightforward way. Consider the escape character example from the previous section: in that example, the consumption of a backslash meant that it became trickier to formulate an informative error, which `explain` helped to mitigate. Instead, `amend` provides an alternative solution to this problem:

```
val strLetter = noneOf('\"', '\\').label("string char")
            <|> amend('\\' *> escapeChar).label("escape char")
```

By pulling the error generated after the parsing of the backslash back, it can now accept the "escape char" label, and the caret now points at the backslash itself, which can be seen in fig. 8.7a. This error message is not perfect, however, since it seems to indicate that the backslash is the problem, when really it is \a itself. However, this can be mitigated by adding `unexpected(2, "illegal escape sequence")` to the end of the `escapeChar` choices. This explicitly sets a caret width of 2, and this will be pulled back by the `amend` too, resulting in an improved error in fig. 8.7b.

While the decision about which error is better is subjective, both kinds of approach do have a benefit, and the `explain` introduced previously can still be effective here too: the use of `amend` here boils down to where the parser author intends the error to originate from. Notice, however, that the "string char" label appears in both amended errors in fig. 8.7: this is an artifact of the observed offset of the error changing, allowing it to merge with the error from `noneOf("\"", "\\")`, as opposed to dominating it.

Building Filtering Combinators with `Entrench` and `Dislodge`

Filter combinators as introduced in §2.2.4 are a way of introducing lightweight context sensitivity into a parser without relying on monads. The basic definition of `filter` makes use of `empty`, however, and this does not lead to good error messages. Instead, variants can be made that make use of `explain`, `unexpected`, or `fail` to provide error messages for the filtering.

For now, here are definitions for two combinators that generate fixed messages if the filter fails, as well as `filter` itself defined as *extension methods* (see appendix A.4.3):

```
def filterWith[A](p: Parsley[A], f: A => Boolean, err: =>Parsley[Nothing]) =
    select(p.map(x => if f(x) then Right(x) else Left(x)), err)
extension [A] (p: Parsley[A])
    def filterSpecialised(msg0: String, msgs: String*)(f: A => Boolean): Parsley[A] =
        filterWith(p, f, fail(msg0, msgs: _*))
    def filterVanilla(unex: String, reason: String)(f: A => Boolean): Parsley[A] =
        filterWith(p, f, unexpected(unex).explain(reason))
    def filter(f: A => Boolean): Parsley[A] = filterWith(p, f, empty)
```

Each of `filterSpecialised` and `filterVanilla` filter a parser but create a customised error message on failure. As an example, assume there is a parser for parsing arbitrary-precision decimal floating point numbers `floating`:

`Parsley[BigDecimal]`; the parser writer wants to turn this into a `Parsley[Double]` and wants to ensure that the value is representable as a `Double` before making the conversion. This is a perfect use for the filtering combinators:

```
def isDouble(n: BigDecimal): Boolean =
    val x = n.toDouble
    n == 0.0 || n == -0.0 || !x.isInfinity && x != 0.0 && x != -0.0
val msg = s"literal is not within the range ${Double.MinValue} to ${Double.MaxValue}"
val double = floating.filterSpecialised(msg)(isDouble).map(_.toDouble)
```

The above parser will first verify the range of a parsed `BigDecimal` to ensure it fits within the bounds of `Double` at least – even if not precisely represented. The given `msg` will be used should this validation fail, reporting an error as follows:

```
(line 1, column 6):
  literal is not within the range -1.7976931348623157E308 to 1.7976931348623157E308
  >1e400
        ^
```

The error here is pointing after the literal has already been parsed: this is not ideal, and it is preferable if the caret points at the start of the literal, at least. This can be easily done by adding an `amend` to the outside of the `filterWith` combinator as seen before. However, what about if there is a mistake inside the floating-point literal? Assume that valid literals must either contain a `.` or `e` for scientific notation, then the following is not a valid literal: `123`. Compare the errors in fig. 8.8: the error in fig. 8.8a is reasonable, pointing out that a decimal point or exponent is required; however, the error in fig. 8.8b has been mangled by the `amend` within the filter and suggests that these are expected at the front of the literal!

`entrench` While `amend` was useful here for pulling the error generated from the filtering back to the start of the filtered input, it also destroys any meaningful errors that arise from the argument to the filter. This is where the second combinator, `entrench`, comes in. At its core, `entrench(p)` does nothing more than protect the errors that arise from within `p` from being affected by the `amend` combinator: the etymology of `amend` and `entrench` come from US legal terminology, where entrenched clauses have constitutional protection against amendments. The way to fix the problem outlined above is to incorporate the `entrench` combinator into the filter to protect the errors within the argument `p`:

```
def filterWith[A](p: Parsley[A], f: A => Boolean, err: =>Parsley[Nothing]) =
    amend(select(entrench(p).map(x => if f(x) then Right(x) else Left(x)), err))
```

With this, the error messages will now behave as expected with respect to the filtering itself (i.e., both errors in fig. 8.8 would be the same). However, suppose that in some part of the wider parser an `amend` was used to correct this error: the `entrench` applied within the filter prevents this `amend` from functioning as well.

```
(line 1, column 4):
  unexpected end of input
  expected digit, ".", or "e"
  >123
     ^
```

(a) using `floating`

```
(line 1, column 1):
  unexpected "1"
  expected digit, ".", or "e"
  >123
   ^
```

(b) using `double`

Fig. 8.8: Difference in errors from `floating` and `double`

`dislodge` While `entrench` is a useful tool for protecting errors against wayward `amend` combinators, it can be overzealous in its role. To counterbalance this, its antonymous combinator, `dislodge`, can be used to undo entrenchment. Like `entrench`, the role of `dislodge` is to simply remove the entrenched property on an error message; however, care must again be taken to ensure that the same problem does not arise in reverse, where a `dislodge` undoes a legitimate `entrench` from within the argument to the filter itself. To make this system more refined, `dislodge` can either remove all entrenchments, or can remove a specific number of them: `entrench` increments a counter as opposed to setting a flag and an error is entrenched if this counter is non-zero. In this case, the final fix to the `filterWith` combinator is as follows:

```
def filterWith[A](p: Parsley[A], f: A => Boolean, err: =>Parsley[Nothing]) =
    dislodge(1)(amend(select(entrench(p).map(x => if f(x) then Right(x) else Left(x)), err)))
```

With the outer `dislodge(1)` in place to counter the `entrench`, this combinator now behaves as desired. The `amend` still exists between the two combinators, so gets to influence the `err` parser, but cannot influence anything within `p`, nor are any external interactions blocked by the `entrench`. While `amend` is by far the most useful of the three combinators, the pairing of `entrench` and `dislodge` can be used to create a fine-grained scope within which `amend` is allowed to operate. It is useful to create an `amendThenDislodge(n)` combinator for convenience.

Context-aware errors The filters seen so far in this section perform context-sensitive filtering, but with statically determined error messages. Some errors additionally benefit from being able to refer to the thing that was parsed within an error message. This can be accomplished using monadic `flatMap`:

```
def filterWith[A](p: Parsley[A], f: A => Boolean, err: A => Parsley[Nothing]) =
    amendThenDislodge(1) {
        entrench(p).flatMap { x => if f(x) then pure(x) else err(x) }
    }
```

This combinator is like the original `filterWith`, except it uses `flatMap`, which allows it to generate an error parser based on the result `x` of the parser `p`. This can be used to enrich the more abstract combinators with improved messages that refer to the parsed value, if desired. However, the use of `flatMap` is expensive, and indeed, when this system is ported to Haskell `parsley` (chapters 3, 4, and 6), the use of monadic combinators is specifically ruled out. Instead, this combinator can be made primitive into the library, with the various possible `err` implementations defunctionalised to allow high-level combinators to hook into it in a uniform way. Indeed, for performance it is valuable to do this for Scala `parsley` too, and the `err` can be translated to the `ErrorGen` ADT:

```
enum ErrorGen[-A] extends Parsley[A => Nothing]:
    case Specialised(msgGen: A => Seq[String])
    case Vanilla(unexGen: A => Option[String], reasonGen: A => Option[String])
```

This need not be exposed to the users but captures all the possible interactions between the error system and filtering combinators. Making it a subtype of `Parsley[A => Nothing]`, or correspondingly a constructor within the Combinator AST in Haskell, allows for the original `select`-based implementation to be used.

Partial Amendment

Section 8.1.2 introduced the `amend` combinator, which can be used to pull the error offset of an error back to an earlier position, and §8.1.2 used this to make a set of filtering combinators that report the error at the start of the token. Some uses of filtering can be considered recoverable, like ensuring identifiers are not keywords, but others are a clearer indicator of failure – the `double` example is one such case. The problem with the current formulation of `amend` is that it will make errors seemingly occur at earlier points in the input without respecting their origins at all: when two errors have matching error offsets they are allowed to merge, which may mean that irrelevant information is added into a filtering-based error. As an example, consider the following parser:

```scala
val double = floating.filterVanilla("bad double", msg)(isDouble)
                     .map(_.toDouble).label("double")
val integer = /*snip*/.label("integer")
val number = atomic(double) <|> integer
```

For the sake of example, assume that the definitions of `double` and `integer` have been carefully constructed so that `integer` will not succeed if followed by something resembling a floating value. Using the current description of `amend`'s behaviour and its use in `filterSpecialised`, parsing a value that does not fit into `Double` will produce the following error:

```
(line 1, column 6):
  unexpected bad double
  expected double or integer
  literal is not within the range -1.7976931348623157E308 to 1.7976931348623157E308
  >1e400
   ^
```

This error is not so bad: it points out that the double is to blame, the reason, and tells the user what they should have provided. However, one might argue that telling the user that an integer could be expected is meaningless when they clearly were trying to parse a double. The `filterSpecialised` combinator is one way of removing all this information since specialised errors will dominate vanilla ones. But a more satisfactory solution would be to allow `amend` to have a more fine-grained control over the offset: this means distinguishing between the *presentation offset* and the *underlying offset* of an error. The *presentation offset*, as discussed previously, is used to decide whether labels and reasons can be applied to an error – this is the visible information of the error. However, the *underlying offset* of an error is where it claims to originate from, and two errors are only allowed to merge if they occur at the same underlying offset and then subsequently the presentation offset. A full `amend`, which is what has been demonstrated so far, will uniformly adjust both presentation and underlying offset of an error, but a `partialAmend` will only adjust the presentation offset, leaving the underlying unchanged. Supposing that `withFilter` is changed to use `partialAmend`, the above error would instead become:

```
(line 1, column 6):
  unexpected bad double
  expected double
  literal is not within the range -1.7976931348623157E308 to 1.7976931348623157E308
  >1e400
   ^
```

The effect of the `partialAmend` here is to "focus" the error in on the failing branch: however, it will still apply the label "double" since the *presentation offset* of the error makes that sensible to apply. In terms of the high-level API, it is wise to support variants of the filters that do and do not have the partial amending applied to them.

Caret Widths

The filtering combinators in §8.1.2 currently produce carets of a fixed size of one. This is not so bad, but it would be better if the caret spanned the entire width of the parsed input. In §8.1.1, it was mentioned that both the `unexpected` and `fail` combinators can produce errors with a specified caret. This is reflected in table 8.1, which shows how unexpected items of various kinds can be generated with various lengths of caret. The zero-width caret for raw or end-of-file items does not make much sense, but conspicuously there is a gap in the table for wider "empty" carets. Recall that the `empty` combinator generates errors with no information, including a zero-width caret: but filters that rely on `empty` should still be able to benefit from a wider caret even if the unexpected message itself is not required.

It appears as if a class of combinator is missing, then, to complete this table properly (table 8.2). The obvious way of adjusting this would be to allow `empty` itself freedom to specify a caret-width: `empty` can then be defined as `empty(0)`. Laws that involve `empty` are referring to this `empty(0)` variant only: an `empty` with a wider caret is not informationless. This now allows for the regular `filter` combinator to have a stretched caret with an otherwise informationless error message.

Implementing this, however, again poses a challenge: the information required to formulate this caret for `p.filter(f)` would be the start, `s`, and end, `e`, offsets around the parser `p` which somehow need to make their way to the failing combinator in the form `e-s`. Like in §8.1.2, the `ErrorGen` type can be used to eliminate the `flatMap` required to generate context-aware messages but retain the selective API. Here, the values placed into the `Either` will differ between the success and failing branches:

```
enum ErrorGen[-A] extends Parsley[((A, Int)) => Nothing]:
    case Specialised(msgGen: A => Seq[String])
    case Vanilla(unexGen: A => Option[String], reasonGen: A => Option[String])
// offset: Parsley[Int]

def filterWith[A](p: Parsley[A], f: A => Boolean, err: ErrorGen[A]) = amendThenDislodge(1) {
    select(entrench((offset, p, offset).zipped {
        (s, x, e) => if f(x) then Right(x) else Left((x, e-s))
    }), err)
}
```

In this version, `ErrorGen` must incorporate the calculated caret width into the generated error message (again, this will correspond to one of `empty`, `unexpected`, or `fail`). Of course, a "pure" implementation can also be given

Caret	*empty*	*raw/eof*	*named*
0	`empty`	n/a	`unexpected(0, _)`
1	???	`satisfy`	`unexpected(1, _)`
n	???	`string`	`unexpected(n, _)`

Table 8.1: Partial possible configurations for unexpected items

Caret	*empty*	*raw*/*eof*	*named*
0	`empty(0)`	n/a	`unexpected(0, _)`
1	`empty(1)`	`satisfy`	`unexpected(1, _)`
n	`empty(n)`	`string`	`unexpected(n, _)`

Table 8.2: Possible configurations for unexpected items

in terms of `flatMap` and the aforementioned combinators manually, though it is less efficient. The effect of the new implementation can be seen in the following error message, whose caret is much more descriptive:

```
(line 1, column 1):
  literal is not within the range -1.7976931348623157E308 to 1.7976931348623157E308
  >1e400
   ^^^^^
```

***Rigid* and *flexible* carets** When the `unexpected` and `fail` combinators are used without specifying their caret, they have what is called a *flexible* caret; contrarily, specified carets are known as *rigid* carets. The distinction is that *rigid* carets may only be stretched to a wider width if they merge with another rigid caret; and *flexible* carets are allowed to merge with a wider variety of errors. Concretely, here is the system for merging errors:

- When two errors merge, the deeper of the two **always** dominates, comparing underlying offset first, then presentation offset.
- Otherwise, the two errors are at the same underlying and presentation offsets:
  - Specialised errors will not merge content with vanilla errors and will always dominate them.
  - Specialised errors will always just merge with each other, taking the longest caret.
  - Vanilla errors will merge, with a priority on the caret and unexpected item:
    - End-of-input will dominate all other items and has caret size 1.
    - Unexpected items will dominate raw items.
    - Matching item types will prefer the longer caret.
- When one of the two errors has a flexible caret, it may override the above rules:
  - A flexible caret is always dominated by a rigid caret, even if it is smaller.
  - A flexible caret may be stretched by an otherwise incompatible error: a specialised error with a flexible caret may accept a wider caret from a vanilla error, say.
  - A flexible caret on an unexpected item may stretch in the presence of a wider caret from a raw item.

Since errors generated by `empty` are at the bottom of the merging ordering anyway, there is no value to specifying the rigidity of the caret for this combinator. The distinction between flexible and rigid carets allows for a parser writers intention to be respected, while allowing freedom for the error system if not specified.

8.1.3 Error Formatting

The error messages presented so far in this chapter have been generated directly from `parsley` using its default error rendering. However, string-based errors messages are not ideal for some applications, like compiler writing, where errors may need to be structured differently or have consistent formatting with another part of the system. As such, `parsley`, supports a system by which the type of an error can be changed, with a rich system of sub-structures making use of associated types [Amin et al. 2016] – this is discussed in §8.1.3.

In §7.1.2, it was mentioned that a dedicated lexing pass with a parser combinator implementation is an anti-pattern: this affects how the API of `parsley` has been designed. This means that error messages often refer more to raw unexpected items than other kinds – except where explicitly generated by the `unexpected` combinator or its derivatives. One advantage of the lexing approach, however, is that it allows for more specific errors based on tokens and not raw input. Section 8.1.3 describes a system by which `parsley` can control how unexpected tokens are handled, allowing it to leverage the benefits of a lexing phase with respect to error messages, whilst not making use of a dedicated pass within the parser itself.

The `ErrorBuilder`

Figs. 8.1 and 8.2 demonstrated the high-level structure of the two kinds of error message within `parsley`'s error system. This structure is encoded into the error construction so that the user has control of the formatting for each specific sub-component, with the error system then stitching user provided values together in the right way: this is a use of the OOP *Builder* pattern [Gamma et al. 1995], where a complex object is constructed independent of its final representation. In fact, this system materialises as a form of typeclass (appendix A.4.1), bridging together the OOP and functional worlds.

There is not much insight to be gained from exploring the entire space of the `ErrorBuilder`, since it is mostly mechanical; instead, parts of the system will be outlined. The outermost part of the trait/typeclass is as follows:

```
trait ErrorBuilder[+Err]:
    type Position
    type Source
    type ErrorInfoLines
    def format(pos: Position, src: Source, lines: ErrorInfoLines): Err
    def pos(line: Int, col: Int): Position
```

Notice that the only type parameter is the type `Err`, which is the final type of the error that the user desires: this is the only type that is relevant for the overall use of any `ErrorBuilder` instance. To make this concrete, the `parse` method on `Parsley[A]` has the following signature:

```
def parse[Err: ErrorBuilder](input: String): Result[Err, A]
```

Here, the type `Err` has a so-called context bound on `ErrorBuilder` (appendix A.4.1), which can be thought of as the same as the Haskell `ErrorBuilder err =>`. The remaining types required by `ErrorBuilder` are associated types as they are only required for internal method calls. Unlike a regular *Builder*, the `ErrorBuilder`'s methods are only designed to be called from within `parsley` itself, and it never needs mutation nor can it control how the object is constructed: in fact, the associated types guarantee that the internal use of a given instance cannot hijack these user-directed components, nor can it put them in the wrong places, by parametricity.

An implementer of the `ErrorBuilder` typeclass will be forced to specify a concrete type for each of `Position`, `Source`, and `ErrorInfoLines` (along with any other associated types): this will allow them to implement the methods that consume and produce these types. In the above example, the `pos` method is a producer of `Position` and `format` is a consumer of it: this forces `parsley`'s hand internally, so that it will pass a call to `builder.pos(line, col)` to `builder.format`, and the user must maintain the consistency themselves. As an example:

```
given ErrorBuilder[String] with
  type Position = String
  ...
  def pos(line: Int, col: Int): String = s"($line, $col)"
```

Notice that the choice of `String` for `Position` now means that the `pos` method returns a string.

Token Extraction

Section 8.1.3 introduced the `ErrorBuilder` typeclass, which can be used to format errors in a user-directed way. The typeclass has an `unexpected` method to format an erroneous token. However, the origin of this token is not as straightforward as table 8.2 would suggest. In fact, the `ErrorBuilder` offers an opportunity to bypass the creation of raw tokens from the input in the form of the following method:

```
trait ErrorBuilder[+Err]:
    ...
    def unexpectedToken(cs: Iterable[Char], demandedInput: Int, lexicalError: Boolean): Token
```

Here, the `unexpectedToken` method takes the remaining input as an iterable stream of characters; the amount of input that the parser wanted to consume, which is the "original" caret width; and whether the parser generated this error while performing "lexing." In turn, it returns a `Token`:

```
enum Token(caretWidth: Int):
    case Raw(tok: String)                     extends Token(tok.codePointCount(0, tok.length))
    case Named(name: String, caretWidth: Int) extends Token(caretWidth)
```

The two cases within `Token` allow for a raw token extracted from the input, or a named token somehow synthesised from the remaining input. The `caretWith` field is be used to formulate the final caret width for the error.

Lexical errors As mentioned, the `lexicalError` parameter for `unexpectedToken` represents whether the error was generated during "lexing." Since there is no dedicated lexing parse within in `parsley`, what does this mean? In fact, this is a property of any error message that is generated underneath the `markAsToken` combinator:

```
def markAsToken[A](p: Parsley[A]): Parsley[A]
```

This combinator can be placed by the parser writer around things that should be considered as tokens within the grammar (i.e., this forms part of the *Tokenizing Combinators* pattern from §7.1.2). The idea is that if a failure occurs within a token, then this would have been an error from the lexer in a traditional two-pass system: as such, the unexpected item would still be in terms of the underlying input and not tokens as in the parser. Not all implementations of `unexpectedToken` will use this, but those that try to mimic pre-lexed input will.

Simple extractors Unlike the rest of the ErrorBuilder, token extraction is something that has a few distinct recipes and does not rely on user-directed typing. As such, parsley provides a few recipes for extracting tokens. One such extractor is MatchParserDemand:

```
trait MatchParserDemand { this: ErrorBuilder[_] =>
    override final
    def unexpectedToken(cs: Iterable[Char], demandedInput: Int, lexicalError: Boolean) =
        whitespaceOrNonPrintable(cs) match
        case Some(name) => Token.Named(name, demandedInput)
        case None       => Token.Raw(takeCodePoints(cs, demandedInput))
}
```

The this: ErrorBuilder[_] => on the first line means that the extractor must be mixed into an ErrorBuilder to be valid and is therefore allowed to override unexpectedToken explicitly. The implementation works by first considering the initial UTF-16 [Hoffman and Yergeau 2000] code-point of the input checking to make sure it is not a non-printable character or whitespace (which, to make the message clearer, should be handled as named items); otherwise returns a raw token as wide as the number of code-points the parser demanded: the function takeCodePoints is a version of take that is code-point aware. The whitespaceOrNonPrintable helper function has the following type signature:

```
def whitespaceOrNonPrintable(cs: Iterable[Char]): Option[String]
```

The full code is given in appendix F.1 for interest: it will return a name to represent the first code-point of the Iterable if it would not otherwise be visible to the reader (examples of use in appendix).

In addition to this extractor, there are two other pre-made extractors in parsley: SingleChar, which only reports the next code-point as the problematic token; and TillNextWhitespace, which will cut the token off at the next whitespace character or optionally matches parser demand, whichever comes first. The code for both can be found in appendix F.2. By default, parsley uses TillNextWhitespace trimmed to parser demand.

Using lexing parsers Section 7.2.2 mentions that it is a good idea to keep non-lexeme token definitions around because they can be used for token extraction. This behaviour is captured by the LexToken extractor:

```
trait LexToken { this: ErrorBuilder[_] =>
    def tokens: List[Parsley[String]]
    def selectToken(toks: NonEmptyList[(String, Int)]): (String, Int) = toks.maximumBy(_._2)
    def extractItem(cs: Iterable[Char], demandedInput: Int): Token

    override final
    def unexpectedToken(cs: Iterable[Char], demandedInput: Int, lexicalError: Boolean) =
        if lexicalError then extractItem(cs, demandedInput)
        else extractToken(cs)

    private lazy val extractParser: Parsley[Either[NonEmptyList[(String, Int)], String]] =
        val toks: Parsley[List[Option[(String, Int)]]] = traverse(tokens) { p =>
            option(lookAhead(atomic(p <~> offset)))
        }
        val toksNonEmpty = toks.mapFilter(ts => NonEmptyList.fromList(ts.flatten))
        val raw = stringOfSome(traverse_(tokens)(notFollowedBy(_)) *> item)
        toksNonEmpty <+> raw

    private final def extractToken(cs: Iterable[Char]): Token =
        assert(cs.nonEmpty, "`unexpectedToken` never provides empty input")
        val Success(rawOrToks) = extractParser.parse(cs)
        rawOrToks.fold(selectToken(_) andThen Token.Named(_, _).tupled, Token.Raw)
}
```

This extractor has three configurable parts: `tokens` is a list of tokens that the extractor should try and identify at the head of the input; `selectToken` specifies a strategy for picking a desired token from a non-empty list of parsed tokens (defaulting to picking the left-most longest); and `extractItem`, which specifies how a token should be extracted if the error was lexical – this will normally be one of the other three extractors.

The `unexpectedToken` implementation in this case would perform the user-directed `extractItem` if the error arose from within a token and otherwise uses the `extractToken` helper method. The `extractToken` method feeds the residual input to the parser `extractParser`, which is never allowed to fail, and folds the `Either` that is returned as a result: a `Left` here indicates that one or more tokens were successfully parsed, so `selectToken` is used to pick one, and the resulting pair is built into a `Token.Named` by passing it to the uncurried constructor (see appendix A.3); alternatively, a `Right` indicates that no tokens matched and raw input has been parsed up until the next valid token – this is then used to build a `Token.Raw`.

The parser `extractParser` is carefully constructed so that it may never fail and can also non-deterministically parse all possible tokens, even with `parsley`'s deterministic PEG-like choice operation. First, all the valid `tokens` are sequenced together, each having been made optional (using `option`, which returns an `Option`) and guaranteed to consume no input on either success or failure with `atomic` and `lookAhead`; each token is also paired with the offset of the parser after it has been successfully parsed, representing the token's length. Then the list of optional tokens is first flattened to remove any tokens that failed to parse and unwrap the others; then, if this results in a non-empty list, the parser `toksNonEmpty` succeeds, otherwise it fails. The parser `raw` will try and extract the longest sequence of characters that does not form part of a valid token: this is done by constructing a string of one or more characters with `stringOfSome`, where before each character is parsed none of the tokens may be valid[3]. Finally, the two sub-parsers `toksNonEmpty` and `raw` are joined together with `<+>`, pronounced "sum," an <|> that injects the left-hand side into a `Left` and the right-hand side into a `Right`.

The `LexToken` extractor is by far the most powerful of the built-in extractors: the user only needs to specify the set of tokens in their language, probably using the `Lexer` functionality they were using for their main parser, and they get error messages that appear to have been generated with a true lexer/parser distinction with none of the downsides. As an example, `parsley`'s `DefaultErrorBuilder` can be augmented with a `LexToken` extractor:

```
given ErrorBuilder[String] = new DefaultErrorBuilder with LexToken:
    def tokens = List(
        lexer.nonlexeme.integer.decimal.map(n => s"integer $n"),
        lexer.nonlexeme.identifier.map(v => s"identifier $v"),
        ...
    ) ++ keywords.map(k => lexer.nonlexeme.symbol(k) #> s"keyword $k")
```

[3]By repeating eq. (2.39), `notFollowedBy(choice(tokens.map(atomic(_))))` is the same as `traverse_(tokens)(notFollowedBy(_))`.

```
// each "simple" extractor aliases its behaviour as a function for convenience
def extractItem(cs: Iterable[Char], parserDemand: Int) = SingleChar.unexpectedToken(cs)
```

In the above example, `SingleChar` (appendix F.2) is used as the back-up extractor; and a list of each relevant token is given, appended onto a list of the keywords. The following examples of error messages are all generated by the WACC Reference Compiler, which uses `parsley` and in particular `LexToken` in a similar way to above:

```
(line 12, column 14):
  unexpected integer 0
  expected ";", operator, ...
  >  int x = 22 0
                ^
```

```
(line 12, column 12):
  unexpected identifier A4
  expected ";", operator, end of integer, ...
  >  int y = 1A4
              ^^
```

```
(line 12, column 23):
  unexpected "++"
  expected ";" or end of program
  >  int[] b = [1, 2, 3] ++ [4] ;
                         ^^
```

```
(line 13, column 16):
  unexpected "e"
  expected end of character literal
  >    char c = 'Hello'
                  ^
```

The first two errors both highlight instances where a token was successfully parsed and used to refine the error; the third highlights a non-token '`++`' which has been handled as a raw piece of input; and the last highlights an error within a token, which does not attempt to extract the identifier '`ello`' thanks to the `markAsToken` combinator used on character literals. In all, this is effective, with good specificity even in the absence of a lexer.

Summary

This section introduced the primitive combinators for error production including those that generate different message components and those that control the overall context of an error. These relatively few primitives allow for fine control over the error messages that `parsley` generates, while not exposing the underlying complexities of the system to the user: this is a strength, since the optimisations performed within §8.2 would make any such low-level interaction with the system very difficult to manage – or defeat the points of the optimisations.

The distinct separation between parsing offset and the error offsets allows for more intuitive interactions between the error combinators and the regular parsing combinators: no expressivity is lost, however, because the use of the `amend` combinator can be used to perform the same roles that `try` does in `megaparsec`'s formulation.

Finally, the `ErrorBuilder` abstraction allows for rich formatting of an error directly into a user-defined type; the addition of the `unexpectedToken` method allows for much finer control over the unexpected messages. Not only does `LexToken` allow for very prescriptive error messages, it also comes at no cost to the user during the main parsing: any associated cost for the extraction is only performed once after the parser has terminally failed.

8.2 System Implementation

Section 8.1 describes the underlying design choices that have contributed to the API for the `parsley` error system. This section briefly outlines some of the key implementation details for the system in isolation, without discussing how the combinators themselves operate within Scala `parsley`'s abstract machine [Willis and Wu 2018], as it is irrelevant.

Performance problems The new error system was first introduced in `parsley:2.6.0`: at the time, this was noted to result in an approximately 3× slowdown of the library. However, the implementation was fully strict, and performed lots of expensive work, much of which may be abandoned later in parsing. Work in `parsley:2.8.4` rectified this, reporting performance comparable to the original system.

Laziness The performance issues introduced by the rework boiled down to the increased redundant work performed throughout the parser. Introducing laziness into the system is one way of mitigating this, however there are two problems with that: the first is that Scala's laziness is not as ubiquitous as Haskell's; and the second is that overuse of laziness creates large space leaks.

Unforced thunks create space leaks [Hughes 1983]: this is where a piece of uncomputed data is retained (and is still accessible for purposes of garbage collection) but will never be used. If every operation in the error system were to be lazy, then this would build up a large tower of thunks needed to be evaluated right at the end of the parse: in the meantime, these thunks consume space, and many will not even form part of the end result due to how errors merge. This creates pressure on the garbage collector, which will affect performance and vastly increase the memory requirements for a parser to run. This must be avoided to ensure the system scales well.

A better solution Given the drawbacks of unconstrained laziness, a more nuanced approach was taken. Defunctionalisation [Reynolds 1972] is the process by which higher-order functions within a program are transformed into specialised first-order functions. However, they can also be represented as pure data [Danvy and Nielsen 2001]: in this formulation, a single function is responsible for performing the job of the defunctionalised operations by pattern matching on the data itself. In other words, one creates a deep-embedded DSL for the program they are working in.

This is the approach that was taken to fixing the performance issues with the original reworked error system: defunctionalise all the operations for both errors themselves and hinting and evaluate this language into an error message at the point of failure. However, this is like the unconstrained laziness approach outlined above: the more effective approach is to wrap each defunctionalised operation in a smart constructor that may perform some (but not all) of the work as to try and eliminate space-leaks.

Defunctionalised errors Error messages within `parsley`'s system are modelled by the `DefuncError` datatype (fig. 8.9). All the operation and base errors of the system are subtypes of this base class. Each error stores its presentation offset, but the underlying offset need not be stored as a value; and the set of operations required by the system are defined on this type. The properties of the error are stored bit-packed in `flags` where the last 28 bits are used for the entrenchment counter: though omitted, there are helper functions to extract these values.

8.2.1 Representing Operations

The `DefuncError` type is then split into two separate type hierarchies: one to represent vanilla errors, called `VanillaDefuncError`, and one to represent specialised errors, called `SpecalisedDefuncError`. The main respon-

```
sealed abstract class DefuncError:
    val flags: Int
    val presentationOffset: Int
    def underlyingOffset: Int

    def merge(err: DefuncError): DefuncError
    def withHints(hints: DefuncHints): DefuncError
    def withReason(reason: String, offset: Int): DefuncError
    def withLabels(label: Iterable[String], offset: Int): DefuncError
    def amend(partial: Boolean, presentationOffset: Int, line: Int, col: Int): DefuncError
    ...
```

Fig. 8.9: The `DefuncError` datatype

sibility of these classes is to implement the abstract operations on `DefuncError` uniformly for their subtypes. This is where the optimisations happen that help reduce the size of the unevaluated error messages.

```
sealed abstract class VanillaDefuncError extends DefuncError:
    def merge(that: DefuncError): DefuncError =
        if this.underlyingOffset > that.underlyingOffset          then this
        else if this.underlyingOffset < that.underlyingOffset     then that
        // underlying offsets are equal
        else if this.presentationOffset > that.presentationOffset then this
        else if this.presentationOffset < that.presentationOffset then that
        else that match
        case that: VanillaDefuncError                             => VanillaMerge(this, that)
        case that: SpecialisedDefuncError if that.isFlexibleCaret => AdjustCaret(that, this)
        case that: SpecialisedDefuncError                          => that
    ...
    def withLabels(labels: Iterable[String], offset: Int): VanillaDefuncError =
        if this.presentationOffset == offset then WithLabels(this, labels)
        else this
    def amend(partial: Boolean, presentationOffset: Int, line: Int, col: Int) =
        if !this.isEntrenched  then
            VanillaAmend(presentationOffset,
                         if partial then this.underlyingOffset else presentationOffset,
                         line, col, this)
        else this
    ...
```

Some parts of the implementation of `VanillaDefuncError` are given above. Each operation is implemented in a way that can avoid performing the allocation of the underlying concrete object. For instance, `VanillaMerge` will only be constructed if the errors will merge (following the rules outlined previously): if not then the other error will get garbage collected – trading some computation time for reduced space consumption. Similarly, `withLabels` checks pre-conditions of the data to ensure that the operation is only done when it will have an effect. The implementation of `amend` demonstrates the use of the `isEntrenched` flag, and that partial amending is implemented by whether the underlying offset or new presentation offset is passed to the underlying operation `VanillaAmend`. An example of the concrete operation classes is `VanillaMerge`:

```
case class VanillaMerge(err1: VanillaDefuncError, err2: VanillaDefuncError)
    extends VanillaDefuncError:
    override val flags              = err1.flags & err2.flags
    override val underlyingOffset   = err1.underlyingOffset
    override val presentationOffset = err1.presentationOffset
```

The `VanillaMerge` case is allowed to assume that, because the offsets must match before merge, the offsets of only one error can be used instead of taking the maximum of each error's offsets. It combines the flags by bitwise anding them. The type of the errors here guarantees their vanilla status. Assumptions are a running theme with the implementations of the concrete cases for each operation – assumptions are made[4] that the smart-constructors should have verified, and these allow for the data to be set up more effectively with less computation.

There is a similar implementation philosophy behind the `SpecialisedDefuncError` as well. The notable difference is that many of the operations do nothing as the behaviour is specific to vanilla errors. As examples:

```
sealed abstract class SpecialisedDefuncError extends DefuncError:
    ...
    def withHints(hints: DefuncHints): SpecialisedDefuncError = this
    def withLabels(labels: Iterable[String], offset: Int)     = this
    def withReason(reason: String, offset: Int)               = this
    ...
```

Here, the parts of the system that only interact with the vanilla errors return the error unchanged: this is cheap. The other operations are structured similarly, except with `this` and `that` having opposing roles in `merge`.

8.2.2 Representing Errors

Other than the operations, the leaves of the `DefuncError` hierarchy are the base error messages themselves. In the specialised case, there is only one error that can be generated, namely a `SpecialisedError` with messages. However, there are a couple of different errors that can be generated for vanilla errors (each representing one column of table 8.2):

```
case class RawError(presentationOffset: Int, line: Int, col: Int,
                    label: String, unexpectedWidth: Int) extends VanillaDefuncError:
    override val flags = 0x80000000 // just `isVanilla`
    override def underlyingOffset = presentationOffset
case class UnexpectedError(...) extends VanillaDefuncError { ... }
case class EmptyError(...) extends VanillaDefuncError { ... }
case class SpecialisedError(...) extends SpecialisedDefuncError { ... }
```

Each of these four base errors have a `line`, `col`, `presentationOffset`, and some form of caret width. Each sets the underlying offset to be the same as the presentation offset without additional space overhead. Flags are set depending on what the base error's properties are.

8.2.3 Collapsing the Structure

During parsing, `DefuncErrors` are constructed and forgotten. When the parser has terminally failed, exactly one `DefuncError` remains – containing sub-operations and base errors. This needs to be turned into an error, in

[4]In practice these are assertions elided for library publication, e.g., `assume(err1.presentationOffset == err2.presentationOffset)`.

particular a user-directed error.

One advantage of the defunctionalised approach is that it is an entirely immutable construction process: when two errors try merge and one is dropped, this has no effect on the overall system. However, at the end of the construction process, the error is final: all operations that are present within the tree will most certainly impact the result in some way. As such, the process of collapsing an error down can be performed mutably, with the state threaded through in an efficient, yet safe, way. Once the entire `DefuncError` has been traversed, the final state can be fed through a provided `ErrorBuilder` (§8.1.3) to generate the result.

The traversal over each kind of error is a recursive function given a value of either `VanillaState` or `SpecialisedState` depending on its kind. The outline for the `VanillaState` is as follows:

```
class VanillaState(presentationOffset: Int, lexicalError: Boolean):
    var line, col  = 0
    val expecteds  = mutable.Set.empty[ExpectItem]
    var unexpected = EmptyUnexpectItem
    var reasons    = mutable.Set.empty[String]
    private var underLabel = 0
    def withinLabel(action: =>Unit): unit = { underLabel += 1; action; underLabel -= 1 }
    def whenNotUnderLabel(action: =>Unit) = if (underLabel > 0) action
```

Here, the presentation offset and whether the error is lexical can be provided directly from the flags of the `DefuncError` about to be constructed. Then the other components will be modified as required during traversal. When entering a `WithLabels` operation any of the errors underneath it no longer contribute to the expected items of the final error; to support this, the builder keeps a count of the number of label combinators the traversal is underneath and exposes two operations `withinLabel` and `whenNotUnderLabel` that allow the traversal to control when expected items should be added. The `ExpectItem` and `EmptyUnexpectedItem` types are types that specify the different kinds of item: raw, named, and end-of-input. The definition for `SpecialisedState` is similar.

The traversal The traversal itself is a recursive pattern matching function that walks through the frozen operations and performs them in sequence, with a left-to-right traversal.

```
def asError[Err](builder: ErrorBuilder[Err], src: Option[String], err: VanillaDefuncError) =
    val state = VanillaState(err.presentationOffset, err.isLexicalError)
    def go(err: VanillaDefuncError): Unit = err match
        case RawError(_, line, col, label, width) =>
            state.line = line
            state.col = col
            state.whenNotUnderLabel:
                state.expecteds += RawItem(label)
            state.unexpected = state.unexpected.updateRaw(width)
        case VanillaMerge(err1, err2) => go(err1); go(err2)
        case WithLabels(labels, err) =>
            state.withinLabel:
                go(err)
            state.whenNotUnderLabel:
                state.expecteds ++= labels.map(NamedItem)
        case ...
    go(err)
```

```
builder.format(builder.pos(state.line, state.col), builder.source(src),
               builder.vanillaError(...))
```

The snippet of the function `asError` sketches this process. The base errors each set the position within the state and alters the relevant portions of the state; merging two errors just means processing each in turn; and the `WithLabels` operation demonstrates how the labels are suppressed underneath and how they are otherwise incorporated. Finally, the `ErrorBuilder` itself is used to construct the final error by extracting parts of the `state`.

Summary

This section has outlined the general ideas behind how `parsley`'s error system works behind the scenes: the instructions themselves will make use of the operations of the defunctionalised representation to construct objects that represent each operation, with some lightweight shortcutting. This design allows for important work to be delayed until it is needed, dropped if it is not, whilst still helping to keep memory usage from growing unnecessarily large. The final evaluation of a defunctionalised error can benefit from safe mutable state to help construct the result faster.

The hinting system also shares a similar implementation, keeping hold of defunctionalised errors and extract relevant parts from them if the hints are incorporated into a future error. Like the errors, if hints are invalidated or not needed, they will be discarded and garbage collected, and little unnecessary extra work is performed.

8.3 Error Patterns

The following section adapts the ideas found in section 5 of "Design Patterns for Parser Combinators (Functional Pearl)" [Willis and Wu 2021] in collaboration with Nicolas Wu, expanding on them in the context of `parsley` *and developing new combinators to facilitate their use.*

With `parsley`'s error system described, section builds on the work in CHAPTER 7: THE DESIGN PATTERNS OF PARSER COMBINATORS by identifying new patterns to help improve error messages in a targeted way. In addition, it will refine some of the existing patterns within the context of `parsley`, illustrating how the combinators introduced in §8.1 should integrate with the existing framework.

Willis and Wu [2021] presented some rudimentary versions of some of the patterns identified here: the generic focus of that paper meant that the patterns presented either do not necessarily port idiomatically to `parsley` or employ workarounds to handle the absence of the new combinators available here. Given the focus of this dissertation, however, it seems useful to discuss how these patterns fit into `parsley` specifically, where they originated.

Running Example

The discussion of error messages in this section requires an extension to the running example introduced in §7.1; fig. 8.10 extends the expression grammar outlined in fig. 7.1 with simple statements. Specifically, statements are not delimited by semi-colons, but rather the semi-colon is a right-associative binary operator; this allows for

$$
\begin{array}{lcl}
\langle \mathit{stmts} \rangle & ::= & \langle \mathit{stmt} \rangle \text{ `}\texttt{;}\text{' } \langle \mathit{stmts} \rangle \mid \langle \mathit{stmt} \rangle \\
\langle \mathit{stmt} \rangle & ::= & \langle \mathit{asgn} \rangle \mid \langle \mathit{if\text{-}stmt} \rangle \mid \text{`}\texttt{skip}\text{'} \\
\langle \mathit{asgn} \rangle & ::= & \langle \mathit{ident} \rangle \text{ `}\texttt{:=}\text{' } \langle \mathit{expr} \rangle \\
\langle \mathit{if\text{-}stmt} \rangle & ::= & \text{`}\texttt{if}\text{' } \langle \mathit{comp} \rangle \text{ `}\texttt{\{}\text{' } \langle \mathit{stmts} \rangle \text{ `}\texttt{\}}\text{' `}\texttt{else}\text{' `}\texttt{\{}\text{' } \langle \mathit{stmts} \rangle \text{ `}\texttt{\}}\text{'} \\
\langle \mathit{comp} \rangle & ::= & \langle \mathit{expr} \rangle \text{ `}\texttt{<}\text{' } \langle \mathit{expr} \rangle
\end{array}
$$

Fig. 8.10: The description of the statement language

some interesting error message discussion. The statements themselves can either be assignments to a variable, conditional if statements, or an empty statement ‘`skip`’. Specifically, the if statements must have both branches, and the condition will be exclusively formed of comparisons – these are not part of the expression grammar.

Assume that the datatype and relevant bridges (sections 7.1.3 and 7.2.3) have already been constructed, and the lexer has also been adapted to suit, for simplicity. The new parser can be defined as follows:

```
lazy val stmts: Parsley[Stmts] = infix.right1(stmt)(Seq <# ";")
lazy val stmt: Parsley[Stmt]   = asgn <|> ifStmt <|> (Skip <# "skip")
lazy val asgn: Parsley[Asgn]   = Asgn(ident, ":=" *> expr).label("assignment")
lazy val ifStmt: Parsley[If]   = If("if" *> comp, "{" *> stmts <* "}",
                                    "else" *> "{" *> stmts <* "}").label("if statement")
lazy val comp: Parsley[Comp]   = Comp(expr, "<" *> expr).label("comparison")
```

In the spirit of error message discussion, each of `asgn`, `ifStmt`, and `comp` have been labelled appropriately with the `label` combinator. One aspect of the syntax of this language is that the empty statement is explicitly ‘`skip`’, and an empty set of braces is not legal: this is illustrated by the error in fig. 8.11a. This looks like the sort of place where an `explain` may be useful when attached to ⟨*stmts*⟩, indeed this results in the error in fig. 8.11b. The new error message indeed is more informative and helps the user understand why what they have done is wrong. Similarly, semi-colons are not legal between the ‘`if`’ and ‘`else`’ portions of the statement. With these additional `explain` combinators, the parser is:

```
lazy val stmts: Parsley[Stmts] = infix.right1(stmt)(Seq <# ";")
lazy val stmt: Parsley[Stmt]   = asgn <|> ifStmt <|> (Skip <# "skip")
lazy val asgn: Parsley[Asgn]   = Asgn(ident, ":=" *> expr).label("assignment")
lazy val ifStmt: Parsley[If]   = If("if" *> comp, "{" *> stmts.explain("...") <* "}",
                                    "else".explain("semi-colons may not precede the else")
                                 *> "{" *> stmts.explain("...") <* "}").label("if statement")
lazy val comp: Parsley[Comp]   = Comp(expr, "<" *> expr).label("comparison")
```

However, the `explain` on the ‘`else`’ is poor: the reason given for why ‘`else`’ is expected in that position is

```
(line 2, column 12):
  unexpected closing brace
  expected assignment, if statement, or skip
  >x := 7 ;
  >if x < 4 { } else { skip }
              ^
```

(a) without any `explain`

```
(line 2, column 12):
  unexpected closing brace
  expected assignment, if statement, or skip
  empty braces are not permitted, try `skip`
  >x := 7 ;
  >if x < 4 { } else { skip }
              ^
```

(b) with an `explain`

Fig. 8.11: Attempting to write empty braces

```
(line 1, column 16):
  unexpected semi-colon
  expected else
  semi-colons may not precede the else
  >if x < 4 {skip}; skip
                  ^
```

(a) but dangling-if is misleading!

```
(line 1, column 16):
  unexpected identifier a
  expected else
  semi-colons may not precede the else
  >if x < 4 {skip}a else {skip}
                  ^
```

(b) but other garbage is misleading!

Fig. 8.12: Attempting to report bad semicolons before '`else`'

assuming something that is not guaranteed to be true, namely that a semi-colon did precede the '`else`' keyword – or indeed that there is an '`else`' there at all! The errors in fig. 8.12 both demonstrate how this can be misleading, even if it does work properly if "`; else`" is encountered in the input. For fig. 8.12a, the user of the language assumed that an "elseless if" is legal syntax, which it is not, but are told that they should not have written a semi-colon before the '`else`': there is no '`else`'! In fig. 8.12b, the user added an erroneous "`a`", but are told about a non-existent semi-colon being to blame.

As powerful a tool as `explain` can be, it is easy for a parser writer to accidentally use it without ensuring that the contextual obligations of the message are met. Section 8.3.1 explores how to report this problem correctly, using what is known as a *Verified Error*; this new error strategy does have its limitations however, which are addressed in part by *Preventative Errors* (§8.3.2). The addition of `label` to the parser is useful, but also introduces clutter, and §8.3.3 explores how the *Parser Bridge* pattern can nicely integrate with labels to keep this separate.

8.3.1 *Verified Errors*

The unrestricted use of `explain` as seen in fig. 8.12 results in confusing or misleading error messages, even if the idea behind them was well-intentioned. The problem is that the reasons given for an error are making assumptions about what has taken place in the surrounding input; there is nothing necessarily wrong with this, so long as these assumptions are *verified*.

Verifying assumptions requires the writing of additional segments of parser. These extra pieces of parser are called *error widgets*: they should be distinguishable from the main grammar by naming convention if nothing else – for instance, writing an underscore as the leading character of the widget's name. Combinators can be created that help facilitate the creation of these widgets, even if the burden is on the parser writer to use them correctly.

Verifying Different Errors

Consider the problem outlined at the start of this section: the parser writer wants to warn the user that semi-colons are not valid before an '`else`'. Instead of adding an `explain` to the '`else`' parser, try parsing a semi-colon if the '`else`' fails: if the semi-colon is present fail with a specific error, otherwise fail regularly. The parser should be modified to include `"else" <|> _noSemi`, where `_noSemi` is a widget with the following properties:

1. it should have type `Parsley[Nothing]`, so that a successful return is uninhabited.
2. it should produce errors that are unconditionally rooted at the start of the widget.
3. it should generate a caret as wide as the problematic input.

4. if the problematic parser succeeds having consumed input, it should consume input.

5. if the problematic parser fails, it should not consume input nor influence the error message of the surrounding parser in any way.

These properties can all be accomplished by various parts of the system that have been described in §8.1. Condition (2) requires use of `amend`, (3) and (4) can use the same ideas as the filtering combinators in §8.1.2, and (5) makes use of the error suppressing power of the `hide` combinator. In fact, for now the `filterWith` combinator can be used directly. The definition of `_noSemi` is as follows:

```
val _noSemi: Parsley[Nothing] = amend {
    withFilter(atomic(";" *> "else").hide,
               _ => false,
               ErrorGen.Vanilla(unexGen    = _ => None,
                                reasonGen = _ => Some("semi-colons may not precede the else"))
              ) *> empty
}
```

This definition does match all the desired characteristics for the widget: the `amend` on the outside guarantees the error is always rooted at the start; the `atomic` ensures that the semi-colon and ‘`else`’ never fails having consumed input; the `withFilter` using `_ => false` ensures that the filtering always fails, using `ErrorGen` to make an appropriate error; and the `empty` helps to fix the type to `Nothing` since the filter is not aware it always fails from its type alone. The error messages produced as a result of this widget are in fig. 8.13.

This makes a more descriptive error for the semi-colon and the else and nothing extra for either of the two problematic cases in fig. 8.12. The widget can be simplified by inlining the definition of `filterWith` and simplifying with the selective law eq. (2.17) knowing that `Left(())` is always produced for the branch:

```
val _noSemi: Parsley[Nothing] = amend {
        (offset, atomic(";" *> "else").hide, offset).zipped { (s, x, e) => (x, e-s) }
    <**> ErrorGen.Vanilla(unexGen    = _ => None,
                          reasonGen = _ => Some("semi-colons may not precede the else"))
}
```

This behaves identically to the original definition, but with simplified logic: the type is now correct by construction and the noise of the filtering has been eliminated. With the widget in this new form, the error message can be refined further: the current widget handles semi-colon before ‘`else`’, but what about “elseless if”s? In this case, the currently ignored parameter of the `ErrorGen` can be used to arbitrate between different messages if the parser within the `atomic` provides a rich type:

```
(line 1, column 16):
  unexpected ";"
  expected else
  semi-colons may not precede the else
  >if x < 4 {skip}; else {skip}
                  ^
```

(a) and it works!

```
(line 1, column 16):
  unexpected "a"
  expected else
  >if x < 4 {skip}a else {skip}
                  ^
```

(b) no misleading error

Fig. 8.13: Attempting to report bad semicolons before ‘`else`’

```
val _noSemi: Parsley[Nothing] = amend {
    val p = ";" *> ("else" #> true <|> pure(false))
         (offset, atomic(p).hide, offset).zipped { (s, x, e) => (x, e-s) }
    <**> ErrorGen.Vanilla(unexGen    = _ => None,
                          reasonGen = elsePresent => Some {
                            if elsePresent then "semi-colons may not precede the else"
                            else "elseless ifs are not permitted, try `else {skip}` instead"
                          })
}
```

By trying to optionally parse the ‘else’ and returning a boolean to denote whether that succeeds, the ErrorGen can use this value to decide which error message best fits the problem. This gives errors that are not only verified based on context, but that are fine-tuneable without additional required backtracking.

First-Class Combinator

The *Verified Error* pattern can vary in shape depending on the exact requirements of the errors: for instance, it might not be desirable to make the error caret exactly as long as the parsed bad input – perhaps some of it is only required for disambiguation with another alternative and is not part of the core issue. In that case, the parser writer is free to use the five properties of the pattern to write any definition that fits their exact requirements.

However, the structure of the _noSemi widget outlined previously is a common shape for this pattern. As such, it makes sense to introduce a new combinator to encapsulate this pattern and make it easier to apply. In a similar vein to filterWith, a general version can be written, and then more specialised extension methods can be built on top:

```
def verifiedErrorWith[A](p: Parsley[A], err: ErrorGen[A]): Parsley[Nothing] =
    amend((offset, atomic(p).hide, offset).zipped { (s, x, e) => (x, e-s) } <**> err)
extension [A] (p: Parsley[A])
    def verifiedExplain(reason: A => String): Parsley[Nothing] =
        verifiedErrorWith(p, ErrorGen.Vanilla(unexGen = _ => None,
                                              reasonGen = x => Some(reason(x))))
    ...
```

The verifiedErrorWith combinator generalises the structure of the _noSemi widget, and the verifiedExplain combinator builds on top of that with a specific version to generate reasons only. Using this new combinator, the _noSemi widget can be very concisely written as:

```
val _noSemi: Parsley[Nothing] = (";" *> ("else" #> true <|> pure(false))).verifiedExplain {
    elsePresent => if elsePresent then "semi-colons may not precede the else"
                   else "elseless ifs are not permitted, try `else {skip}` instead"
}
```

This final definition has all the same benefits as the original, however clearly captures the parser writer’s intention and removes any chance they forget one of the crucial amend, atomic, or hide combinators that would have otherwise violated the requirements of the pattern. Better still, this combinator allows for a rich error construction without resorting to monadic combinators, which still means it is suitable for use in Haskell parsley.

Willis and Wu [2021] also presented this pattern, however, in the absence of `parsley`'s `amend`, to support property (2) it must rely on `lookahead`, violating property (4). It also can only partially support property (5), as a `label`-based `hide` may still adjust the caret width.

8.3.2 *Preventative Errors*

The *Verified Errors* pattern is useful for manually identifying user problems when no other valid alternative is possible. As they are guaranteed to fail, they will always appear as part of a sequence of `<|>`s. However, there are cases where verified errors are not ergonomic, likely because the correct alternatives are not present nearby or are found in separate parts of the parser. In these cases, the *Preventative Errors* pattern has a slightly different formulation with different properties.

For a good example of where the *Verified Errors* pattern falls down, consider the error message in fig. 8.14a. Here, the user has tried to assign a comparison expression to a variable `x`. However, the grammar states that comparison may only appear inside ⟨*if-stmt*⟩. This might seem like it can be addressed with `verifiedExplain`, but the problem is where it might go. Focusing on `asgn`:

```
lazy val asgn: Parsley[Asgn] = Asgn(ident, ":=" *> expr).label("assignment")
```

In the previous examples, the verified combinator is attached alongside the next thing to parse: this, however, is nowhere to be found for assignment. After careful inspection of the grammar (or the error message), one can determine that there are three things that can follow a completed assignment: the end of the input, a semi-colon, or even a closing brace (twice). Indeed, by attaching a `"<".verifiedReason("comparisons may not appear in assignments")` widget in these three places, the error message does indeed get better – but just as in fig. 8.12, it also brings about misleading errors too, an example of which is seen in fig. 8.14b: in this error, a non-existent assignment is referenced after a '`skip`', and the valid continuation of "end of input" is missing. Simply put, the verified error verifies the presence of the '<', but not the presence of the assignment, and the eager failing means that other alternatives are not collected – any backtracking with `atomic` will "forget" the special error.

Preventing Issues

The *Preventative Errors* pattern differs from the *Verified Errors* pattern by seeking to rule out an issue at source and not as a last resort. In fact, *Preventative Errors* can handle everything *Verified Errors* can, however they are more proactive and so incur overhead on the "good" path as opposed to only when all other options have been exhausted. A *Preventative* widget should have the following desirable properties:

1. it must succeed if the issue is not present: likely returning `Unit`.

```
(line 1, column 8):
  unexpected "<"
  expected operator, ";", or end of input
  >x := 4 < y
          ^
```

(a) Attempting to assign a comparison to a variable.

```
(line 1, column 6):
  unexpected "<"
  expected ";"
  comparisons may not appear in assignments
  >skip <
        ^
```

(b) Misleading results from *Verified Error*.

Fig. 8.14: Restricted comparisons

```
(line 1, column 8):
  unexpected "<"
  comparisons may not appear in assignments
  >x := 4 < y
          ^
```

(a) no expected prompts because of the eager failing

```
(line 1, column 8):
  unexpected "<"
  expected end of assignment
  comparisons may not appear in assignments
  >x := 4 < y
          ^
```

(b) explicit labelling on the widget

Fig. 8.15: Attempting to report spurious comparisons

2. it should produce errors that are unconditionally rooted at the start of the widget.
3. it should generate a caret as wide as the problematic input.
4. if the problematic parser succeeds having consumed input, it should consume input.
5. if the problematic parser fails, it should not consume input nor influence the error message of the surrounding parser in any way.

Indeed, the only changed requirement is (1), which now demands success instead of unconditional failure. This means that this pattern is free to appear anywhere in the parser. A naïve implementation of the `_noComp` preventative error widget might be as follows:

```scala
lazy val asgn: Parsley[Asgn] = Asgn(ident, ":=" *> expr <* _noComp).label("assignment")

val _noComp = optional("<".verifiedExplain(_ => "comparisons may not appear in assignments"))
```

Here, the `asgn` parser has been augmented with a single `<* _noComp` instead of the previous four occurrences with the *Verified Error* attempt. The first property has been satisfied by making the widget optional: this works because of properties (4) and (5), which mean that `optional` only succeeds when the '<' itself fails to parse. The problem outlined in fig. 8.14b has been fixed, and the improved true-positive error can be seen in fig. 8.15a.

However, the error message is now missing any form of labelling to indicate the possible continuations: this is because a preventative error will fail eagerly, without offering a chance to parse any correct alternatives. This means that the `label` combinator becomes important to help manually address this deficiency: though recall it cannot produce either raw items or "end of input." Fig. 8.15b shows how the error looks when the widget is offered its own `label("end of assignment")`, which is at least slightly better.

First-Class Combinator

In general, using `verifiedX` to realise this pattern is not a good idea: property (4) only guarantees that a `verifiedX` widget will consume input on failure if the underlying parser does – when it does not, this approach simply does not work. Another candidate for an implementation, as outlined by Willis and Wu [2021] would be `notFollowedBy`: in this case, however, property (4) is certainly broken.

Instead, the selective implementation of `notFollowedBy` outlined in §2.3.2 can be adapted to match the requirements of the pattern precisely. As the original definition is written in Haskell, here is a translation to Scala to help keep the ideas clear (and in terms of `select` instead of `whenS`):

```
def notFollowedBy[A](p: Parsley[A]): Parsley[Unit] =
    val inner: Parsley[Either[A, Unit]] = atomic(lookahead(p)) <+> unit
    select(inner, empty)
```

In the above definition, the parser `inner` tries to parse `p`, never consuming input and returns a `Left` of the result if so; otherwise, a `Right(())` is returned (recall `<+>` wraps the result of each side of a conventional `<|>` in an `Either`). The `select` then fails with `empty` if a `Left` was returned or applies the identity function if a `Right` is returned. This structure can be exploited to formulate a generic preventative error combinator:

```
def preventWith[A](p: Parsley[A], err: ErrGen[A], labels: String*): Parsley[Unit] =
    val inner: Parsley[Either[(A, Int), Unit]] =
        atomic((offset, p.hide, offset).zipped((s, x, e) => (x, e-s))) <+> unit
    val labelledErr = labels match
        case l +: ls => err.label(l, ls: _*)
        case _       => err
    amend(select(inner, labelledErr))
```

The `empty` should be replaced by a more concrete error that can make use of the `A` value (i.e. `ErrorGen[A]`), the `lookahead` should be removed to ensure failure consumes input (as required by property (4)), an `amend` should be subsequently be added to patch property (2) once the `lookahead` is removed, and a `hide` should be added to suppress the errors of `p`. One final touch is that zero or more labels can be provided to the combinator: if there is at least one label, then the widget can apply the labels, otherwise it is left as-is. As before, concrete extension combinators can be built in terms of `preventWith`, such that the `_noComp` widget can be written as:

```
val _noComp = "<".preventativeExplain(_ => "comparisons may not appear in assignments",
                                          "end of assignment")
```

With the first-class support for the pattern as a combinator, a *Preventative Error* is no more verbose than *Verified Error*. However, they do incur a slight efficiency penalty due to their eager nature, and the continuations must be specified manually. Again, the `parsley` `amend` and `hide` combinators help to make this pattern adhere to all five desirable properties in a way that `notFollowedBy` alone cannot. Like the `verifiedErrorWith` combinator, `preventWith` is implemented non-monadically, which means it can be ported to selective Haskell `parsley`.

8.3.3 Errors and *Parser Bridges*

After the addition of the two error widgets, the parser ends up looking like this:

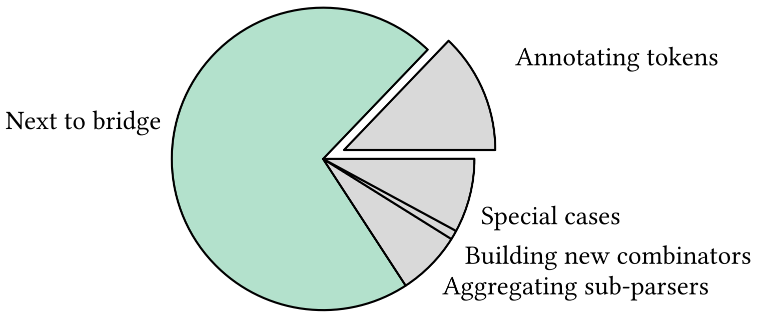


Fig. 8.16: Places in which `.label` was found amongst Scala WACC groups.

```
lazy val stmts: Parsley[Stmts]  = infix.right1(stmt)(Seq <# ";")
lazy val stmt: Parsley[Stmt]    = asgn <|> ifStmt <|> (Skip <# "skip")
lazy val asgn: Parsley[Asgn]    = Asgn(ident, ":=" *> expr <* _noComp).label("assignment")
lazy val branch: Parsley[Stmts] = "{" *> stmts.explain("empty braces are not [...]") <* "}"
lazy val ifStmt: Parsley[If]    = If("if" *> comp, branch, ("else" <|> _noSemi) *> branch)
                                   .label("if statement")
lazy val comp: Parsley[Comp]    = Comp(expr, "<" *> expr).label("comparison")

val _noComp = "<".preventativeExplain(_ => "comparisons may not appear in assignments",
                                       "end of assignment")
val _noSemi = (";" *> ("else" #> true <|> pure(false))).verifiedExplain {
    elsePresent => if elsePresent then "semi-colons may not precede the else"
                   else "elseless ifs are not permitted, try `else {skip}` instead"
}
```

One of the primary benefits of using the *Parser Bridge* pattern (§7.1.3) is that it allows for the decoupling of administrative work from the rest of the parser. However, by annotating the parser with additional descriptive labels and reasons, the parser has become more cluttered. While the error widgets are unavoidable, notice that all the labels applied in this parser are directly attached to the application of a bridge constructor.

In the WACC Reference Compiler, all except for one label are applied directly to either bridge constructors in the parser or are found within the lexer: this makes sense, since labels are usually describing a syntactic entity, which is usually going to be represented by its own AST node. This was often the case in student code too, with the exception that "aggregate" labels – for instance `statement` or `expression`, which combine many sub-non-terminals under one umbrella – are found on the outside of `<|>` chains. Fig. 8.16 shows the proportion of `label` uses across the student codebases for WACC in the academic year 22/23: 71% of uses are next to bridges, with 13% describing lexical tokens, and 7% labelling several parsers at once in aggregation. The remaining uses varied from individual to individual. This is true for many of the uses of the `explain` combinator too.

The existing *Parser Bridge* pattern can be extended with the facilities to abstract some of the more mundane error annotations away from the main body of the parser. Discounting those labels used within the lexer, incorporating labels into the *Parser Bridge* pattern would have been able to eliminate over 82% of the labels (and many of the longer-form `explain`s) from the main body of the student parsers, which would improve readability.

Incorporating `Label` and `Explain`

The current bridges used for this parser are the generic bridges described by §7.2.3, as follows:

```
object Seq extends generic.ParserBridge2[Stmt, Stmts, Stmts]
object Asgn extends generic.ParserBridge2[String, Expr, Asgn]
object If extends generic.ParserBridge3[Comp, Stmts, Stmts, If]
object Comp extends generic.ParserBridge2[Expr, Expr, Comp]
```

Ideally, the labelling properties of the parsers that correspond to these bridges can specify the label information cleanly into these bridges. One way to do this is to expose an overridable `def labels: Seq[String]`, which is set to `Seq.empty` by default, though there are other methods library designers may wish to explore. Labelling can also be performed for deferred application of a bridge, such as `Seq <# ";"`: this is usually in an aggregating way, marking multiple different operators with the same category. As such, it is best to make a new supertype of the `ParserSingletonBridge` trait:

```
trait ErrorBridge:
    def labels: Seq[String] = Seq.empty
    def reason: Option[String] = None
    final protected def error[A](p: Parsley[A]): Parsley[A] =
        reason.foldLeft(label.foldLeft(p)(_.label(_: _*)))(_.explain(_))
```

The new bridge type exposes `labels` and `reason` methods – defaultly set to empty values – and use these to create an `error` combinator that folds these optional values to apply the corresponding `label` or `explain` combinators if the values are present. To complete this, the `error` combinator is used in the definition of `<#` and each of the other bridges will need to use `error` in its template `apply` method:

```
trait ParserBridge2[-T1, -T2, +R] extends ParserSingletonBridge[(T1, T2) => R]:
    def apply(x1: T1, x2: T2): R
    def apply(p1: Parsley[T1], p2: Parsley[T2]): Parsley[R] =
        error(lift2(this.con, x1, x2))
    override final def con: (T1, T2) => R = this.apply(_, _)
```

This works fine; however, the implementer may wish to decouple a bit further, making an `unannotated` combinator to do the `lift2` portion and have `apply` call `error(unannotated(p1, p2))`: this would allow the user to override the `unannotated` behaviour without losing the built-in annotation. However, either way, the original bridges for the example can be modified to provide an override for the `labels` method:

```
object Seq extends generic.ParserBridge2[Stmt, Stmts, Stmts]
object Asgn extends generic.ParserBridge2[String, Expr, Asgn]:
    override def labels = Seq("assignment")
object If extends generic.ParserBridge3[Comp, Stmts, Stmts, If]:
    override def labels = Seq("if statement")
object Comp extends generic.ParserBridge2[Expr, Expr, Comp]:
    override def labels = Seq("comparison")
```

With these overrides in place, the labelling combinator can be removed from the parser, keeping the code more focused on the grammar itself.

Aggregation Bridges

One benefit to the bridge-based approach to labelling is that aggregate labels can also be cleanly factored into the bridges. Suppose instead that the parser writer simply wants to aggregate all the different binary operators within the expression portion of the language: namely '`+`', '`-`', and '`*`' so that they are all referred to as `binary operator`. In this case, this can be cleanly factored by writing an *aggregation bridge*:

```
trait BinaryOpBridge[-T1, -T2, R] extends ParserBridge2[T1, T2, R]:
    override def labels = Seq("binary operator")
```

Now, the parser writer need only extend `BinaryOpBridge` for each of the operators and the labelling with be applied consistently across them. This could also be done as a mixin trait, like how the lexical extractor recipes worked in §8.1.3: this way, the same label can be easily mixed into differently shaped bridges.

8.3.4 Adjustments to Lexers

The final consideration is whether the patterns developed for lexing in §7.1.2 need to be adjusted within the context of `parsley`'s error system. In particular, the `token` and `lexeme` combinators were defined as follows:

```
def token[A](p: Parsley[A]) = lexeme(atomic(p))
def lexeme[A](p: Parsley[A]) = p <* whitespace
```

It was mentioned in §7.1.2 that some tokens do not require the `atomic`, since they are non-ambiguous with respect to the other tokens, and there may be useful errors that can arise within the token – indeed, allowing this to occur was the role of the `markAsToken` combinator in §8.1.3. In the interest of improving the error messages that arise from lexing, tokens that can produce reasonable internal errors should only use `lexeme`, which means that this is a suitable place to keep `markAsToken`. Further to that, `token` now denotes truly atomic errors, so an `amend` should also be used to ensure the errors are rooted at the right place and labels can freely apply to them:

```
def token[A](p: Parsley[A]) = lexeme(atomic(amend(p)))
def lexeme[A](p: Parsley[A]) = markAsToken(p) <* whitespace
```

Labelled Symbols

Section 8.3.3 pointed out that among the uses of `label` found in codebases for Imperial's WACC project, 13% of them occur within the lexing module or uniquely attached to symbols within the parser. While it is easier to keep these labels out of the main parser by defining concrete aliases for the symbols in the lexer, it might be nice if the `symbol` combinator itself (and by extension, the implicit string literal lifting) can handle the labelling for all symbols uniformly and internally – like the bridges. For reference, the Scala version of the `symbol` combinator is:

```
def symbol(sym: String): Parsley[Unit] =
    if keys.contains(sym)     then keyword(sym)
    else if ops.contains(sym) then operator(sym)
    else                           token(string(sym).void)
```

The idea is to incorporate a `Map[String, String]` into the combinator that maps symbols to their labels: if a label is found in the map, this can be applied with `label`, otherwise the natural label is used instead. When defining the `symbol` combinator by-hand, it can close over this like it would `keys` or `ops`; in practice, `parsley` will incorporate this map into its `ErrorConfig` for the `Lexer` class, which allows control over various error messages that would otherwise be opaque to the user of the functionality. Regardless, the new combinator is as follows:

```
def symbol(sym: String): Parsley[Unit] =
    val symP = if keys.contains(sym)     then keyword(sym)
               else if ops.contains(sym) then operator(sym)
               else                           token(string(sym).void)
    labels.get(sym).foldLeft(symP)(_.label(_))
```

Here, `.get` fetches a value from a `Map` returning an `Option`, which is then folded in a similar way to the definition of `annotate` in §8.3.3. With this in place, the naming for all raw symbols within the parser can be changed without modifying the parser itself or creating aliases for them. In addition, the `labels` map can be used to help construct the naming for the `LexToken` token extractor (§8.1.3), reducing duplication further. As an example:

```
val labels = Map("(" -> "open parenthesis", ")" -> "closing parenthesis")
```

The above definition of `labels` will ensure that `"(" *> expr <* ")"` would have an expected label of `open parenthesis` if a '`(`' could not be parsed without sacrificing any readability in the parser itself.

Summary

This chapter has explored `parsley`'s error system, which has iterated on the ideas of both `parsec` and `megaparsec` to create a flexible system with a core set of simple primitive combinators that remains easy to continue to evolve without breaking users' code. The `amend` combinator allows for fine-grained control than the equivalent role of `atomic`/`try` in the previous systems: it has seen effective use across both §8.1 and §8.3 to help control the control of error messages. The token extraction system allows for very expressive token description at no "happy path" cost, and the novel `LexToken` functionality integrates well with other lexing patterns.

The implementation of the system makes use of an explicit defunctionalisation that allows for the exact point each part of the error is processed to be controlled precisely. Here, the use of shortcutting smart constructors helps to keep memory use lower without performing expensive calculations. This helps keep impact of the system on the "happy path" low.

Finally, `parsley`'s error system facilitates elegant new formulations for the *Verified Errors* and *Preventative Errors* patterns, which cannot achieve the same level of fine-tuning when implemented with "generic" combinators. These patterns have first-class support in `parsley`, which simplifies and encourages their use: in fact, the `Lexer`'s `ErrorConfig` even has support for some uses of these patterns within tokens – like illegal characters in strings, say. The integration of the error system into the *Parser Bridge* pattern further improves their utility and allows the noise of error annotation to be kept separate from the parsers themselves.

Chapter 9

Related Work

Parser Combinators

Aside from the parsec-style CPS approach, there are several other underlying models for parser combinators.

Scala Spiewak [2010] presents a simple non-deterministic parser combinator library in Scala adapting the GLL parsing algorithm [Scott and Johnstone 2010; Scott and Johnstone 2013]; this can handle left-recursive grammars without factoring in $O(n^3)$. For globally ambiguous grammars, obtaining all results may be exponential: in practice, this is uncommon. Izmaylova, Afroozeh, and Storm [2016] define the `meerkat` library in Scala, which also can automatically handle left-recursion without chain combinators. However, the implementation strategy used is very slow, and using *Disambiguator Bridges* (§7.1.3) and *Chain Combinators* (§7.1.1) make a parser non-ambiguous with `parsley` is much faster[1].

Parsing with Derivatives Brzozowski [1964] introduced the idea of parsing regular expressions by taking their derivative with respect to a specific character. Since then, Might, Darais, and Spiewak [2011] extended this idea to full context-free grammars: though with awful performance; Adams, Hollenbeck, and Might [2016] improved on this, however, introducing optimisations that make the technique much more viable in practice, including memoisation for left-recursion. This approach is similar to taking the CPS transformation over a parser, and Henriksen, Bilardi, and Pingali [2019] suggests that parsing with derivatives is foundational to the theory of parsing.

Error Recovery and Handling Swierstra and Duponcheel [1996] discussed how to make $LL(1)$ parser combinators that can make use of static information threaded through the parser to perform error recovery and predictive parsing. This is a very different approach and aims to correct faulty input as opposed to reporting errors. The memory characteristics are good, since no backtracking is possible. Maidl et al. [2016] introduce a system for improved error handling and reporting using labelled failures, where different handlers give fine-grained control over how failures are recovered from or propagated; they additionally show how their system can be used to implement a variety of different parser combinator approaches, including the atomic combinator.

Staged Parser Combinators

Light-weight modular staging (LMS) Jonnalagedda et al. [2014] uses LMS [Rompf and Odersky 2010] to pioneer staged parser combinators in Scala. By making use of a runtime technique, they can make use of monadic `flatMap` as opposed to this work's restriction to selective combinators. However, by doing compile-time staging, this work can perform much heavier analysis and optimisation that would otherwise be prohibitively expensive. Compared with the framework provided by Template Haskell, LMS allows for a natural inspection of generated code to perform domain-specific optimisation, however the equivalent is captured by the Defunc and Lam infrastructures in §6.1. Like `parsley`, Jonnalagedda et al. [2014] out-performs its non-staged contemporaries.

[1]`https://github.com/tom91136/scala-parser-benchmarks` demonstrated a 68x speedup from Scala `parsley` over `meerkat`.

Meta-OCaml Krishnaswami and Yallop [2019] discuss the implementation of typed staged parser combinators, but where the typing relates to the properties of the parser and not the results. They produce a system that rejects ambiguous grammars (aligning with the ideas of Swierstra and Duponcheel [1996]), handling what they refer to as μ-regular languages. This contrasts with my work, which opts instead for full PEG-like semantics for parsers, handling ambiguity in the grammar by ordering. Unlike my work, their work does not make use of a deep embedding, nor does it perform any optimisation or analysis. While they rule out ambiguities and left-recursion by rejecting the parser, it is possible to implement the algorithms of Baars and Swierstra [2004] to allow `parsley` to detect and auto-factor left-recursion via a generalised postfix operation – this is future work.

Domain-Specific Languages

`Parsley` is a domain-specific language [Fowler 2010] for parsing, specifically an embedded DSL [Hudak 1996]. As a deep embedding, `parsley` has the freedom to perform multiple traversals over data, with different semantic interpretations applied as traversals [Leijen and Meijer 1999; Gibbons and Wu 2014]. Wu, Schrijvers, and Hinze [2014] demonstrate parser combinators can be modelled with effect handlers, as a composition of non-determinism, state, and a teletype effect. Devriese and Piessens [2012] use deep-embeddings to model grammar languages and represents recursion explicitly, this work, of course, allows for direct-style recursion via the StableName analysis. They demonstrate multiple different interpretations of their grammars as traversals, including reachability (which is a bi-product of `parsley`'s reference analysis) and left-corner transforms [Moore 2000] to eliminate left-recursion. Kurš et al. [2016] models parser combinators as a non-embedded DSL, effectively a parser combinator compiler. It provides heuristics for handling runaway left-recursion, but not a general solution, and interestingly does not adhere to the whitespace convention from §7.1.2, potentially slowing the library down. In contrast to `parsley`'s StableName leveraging for sharing, Ljunglöf [2002] supports observable sharing in Haskell using IORef, though this will artificially inflate the binding counts, resulting in more spurious bindings than `parsley`'s system.

Set-Like Data Structures

CHAPTER 5 introduced the RangeSet data-structure, optimised for semi-contiguous data. There are many other similar sets used for a variety of different applications:

AVL trees AVL Trees [Adelson-Velskii and Landis 1962] are an implementation of a self-balancing binary tree. The key invariant is that given any node, where the heights of its two children are h_1 and h_2, it is the case that $|h_1 - h_2| \leqslant 1$. When an operation is performed that causes a disparity in the heights of two siblings, the tree must be rebalanced using the same balancing scheme used for RangeSet. The type of an AVL tree is:

```
data AVL a = NodeAVL Int a (AVL a) (AVL a) | TipAVL
```

This structure is like RangeSet but without the extra element, and RangeSet's implementation is adapted from the implementation of an AVL tree.

Weight-balanced (WB) trees In contrast, self-balancing size-weighted binary tree [Adams 1992; Nievergelt and Reingold 1972] for the basis for `Data.Set`. WB trees weight the children of a node according to their size, which is the number of elements contained within the child and is proportional the number of nodes within the child. A node is considered imbalanced when the weights of its children differ by a chosen factor, chosen to be 3 for Set. When an unbalanced node is identified, it is also balanced using rotations, and whether one or two rotations are requires depends on the ratio between the weights of the sub-children: if the weights are within the ratio, chosen to be 2 for Set, one rotation is performed, otherwise a double rotation is performed.

The reason that size is chosen as the weighting for these sets is to support size in $O(1)$, which necessitates storing the size in the nodes themselves. Since the size can serve as a weighting, it is more space-efficient to use this for the weighting. In practice, Adams [1992] claims that this only causes a performance difference of a few percent compared with an AVL-style height-based weighting.

Discrete interval encoding trees (DIET) Erwig [1998] introduces a similar structure to RangeSet, for Enum data. DIETs have a similar underlying structure, with a similar invariant that ranges within the set may not touch or overlap. However, unlike RangeSet, DIETs described by Erwig [1998] do not support union, intersection, difference, or complement. Indeed, he says that supporting these operations is possible with divide-and-conquer algorithms, but that it is difficult to implement these operations whilst preserving the invariances of the structure. This work has implemented these operations and shown their asymptotic complexity as a concrete contribution.

Interval and segment trees Computational geometry offers various structures that are like the RangeSet: including Interval Trees and Segment Trees [Berg et al. 2008, Chapter 5]. These structures sometimes do contain ranges in a similar formulation to DIETs or RangeSet, but the operation they support are different: usually asking whether points fall within certain ranges or finding the overlap between different intervals.

Parsers and Design Patterns

There is an abundance of parser combinator tutorials in the wild [Swierstra 2009; Fokker 1995; Hutton and Meijer 1996], but these tend to discuss how to build parser combinator libraries, and not how to use them effectively. While Kövesdán, Asztalos, and Lengyel [2014] discuss design patterns for parsing, this is an architectural discussion, centred around the tools and techniques for building parsers as opposed to what to do once an approach has been chosen. Along with Willis and Wu [2022] and Willis and Wu [2021], this dissertation continues to discuss specific ideas for writing parsers using combinator libraries: some of these ideas have appeared before in one form or another, but usually disjointed folklore.

Classic design patterns Many parsing libraries employ the OOP design patterns [Gamma et al. 1995] as part of either their API or as part of their language internals. Notably, ANTLR [Parr 2013] makes use of the *Visitor* and *Listener* patterns to allow the user to process the parse-tree generated by the tool or interact with the error system. This contrasts with the *Parser Bridge* pattern, which allows for deforested construction of an AST directly while still maintaining good decoupling. The *Verified* and *Preventative Error* patterns contrast with the use of *Listener*, allowing for simple parser driven errors with full contextual information available. Nguyen, Ricken, and Wong [2005] employs the classic OO design patterns to create $LL(1)$ parsers aiming to make them easily extensible and modular – this is, in part, like the aims of the *Parser Bridge* pattern, which aims to decouple aspects of parsing from each other. Schreiner and Heliotis [2008] also use design patterns for the internal design of a parser generator called `oops3`.

Left recursion Some approaches and libraries do avoid the general issue of left recursion, making the *Chain* patterns redundant. Various parsing algorithms, such as Earley's algorithm [Earley 1970], LR [Knuth 1965], and LALR [Deremer 1969b], all avoid left recursion trivially without any modification of the parser required. Many libraries also use some form of memoisation for dealing with left recursion [Frost, Hafiz, and Callaghan 2007; Izmaylova, Afroozeh, and Storm 2016; Johnson 1995], or make it illegal with analysis or typing systems [Danielsson 2010; Krishnaswami and Yallop 2019]. Some libraries can perform grammar transformations either because they are deeply embedded or are a standalone DSL: Devriese and Piessens [2011] employs a left-corner transform [Moore 2000], and Baars and Swierstra [2004] factors left recursion out of parser combinators by using a generalisation of the postfix combinator (§7.1.1). Precedence annotations are used by `happy` to allow for left-recursive grammars to be handled, and various forms of `parsec` support versions of the precedence combinator: though all these combinators are homogeneous variants, unlike the heterogeneous version presented in this dissertation.

Calculated Compilers

Program calculation [Gibbons 2002], involves the systematic derivation of a program from a specification of the program. Bahr and Hutton [2015] showed that compilers could be calculated for a small expression language, by leveraging the specification that eval ≡ exec.compile. They start by giving a semantics for the language, eval, and use induction to systematically derive the compiler and associated stack machine implementation for the language, going on to augment the language with exceptions and other features. Many of the instructions that fall out of these calculations bear striking resemblance to some of the instructions present within `parsley`: the `MARK` and `UNMARK` instructions map to Catch and Commit, for instance. Their presentation intermingles various elements together on a single stack, however. Pickard and Hutton [2021] expand on this technique, introducing dependent types into the formulation: this reinforces the similarities with the instructions of `parsley`, with similar static typing guarantees applied to both. While the discussion in this dissertation has also derived the compiler for parsley, it has done so from a different angle, focusing on the relationship between automata and parsers: the evaluation semantics fall out by parametricity later. The approach in this dissertation also handles the type indices more straightforwardly by using different stacks: this avoids the need for stack concatenation [Pickard and Hutton 2021], and multiple layers of stack unwinding during failure or returns by closing over the stack in continuations. Since `parsley` aspires to model the `parsec`-style CPS formulation of parser combinators, the equational reasoning required to perform a calculation more closely resembling their work would be much more involved; it is interesting, therefore, that the two approaches still do converge to very similar representations. Hughes and Swierstra [2003] provide a systematic description of regular expression parsers, resulting in another similar stack implementation, though they use pure continuations, complicating the types.

Bahr and Hutton [2022] also refine the technique of compiler calculation by allowing the calculation to reason about divergence. In this case, a recursive semantics leads to a calculation that is not inductive, and therefore produces an unbounded amount of stack instructions. This is similar to the problem of the finite versus infinite automata that plagued recursive parsers and the Call instruction. Bahr and Hutton address this issue by introducing a partiality monad and bisimilarity; this technique, however, has not been applied in this work. Bahr and Hutton [2020] change the underlying model of the calculated machinery to work with register machines as opposed to stack machines, this has a model of an unbounded set of registers, much like references found within `parsley`. However, `parsley` exposes registers directly to the user-API with lifetimes constrained by rank-2 types and not as a compilation target. `Parsley`'s references are also heterogeneous, making use of dependent maps, whereas registers used by Bahr and Hutton are purely homogeneous and stored in a list.

Arrows

Arrows [Hughes 2000] are a generalisation of Monad that sits between Applicative and Selective in expressive power [McBride and Paterson 2008; Mokhov et al. 2019]. They support an array of orthogonal combinators compared to the Functor to Monad hierarchy:

```
class Category (⇝) where
  id  :: a ⇝ b
  (·) :: (b ⇝ c) → (a ⇝ b) → (a ⇝ c)
class Category (⇝) ⇒ Arrow (⇝) where
  arr   :: (a → b) → (a ⇝ b)
  first :: (a ⇝ b) → ((a, x) ⇝ (b, x))
```

The high-level idea is to generalise computations from a to b to encode more properties than just a regular function type. The original motivation was to find an abstraction that naturally represents $LL(1)$ parser combinators à la Swierstra and Duponcheel [1996]. A monadic interface does not work because static information is required in the form of FIRST sets: as noted in §2.2.5, the (»=) operation introduces a dynamic dependency between two computations – this prohibits the threading of static information. Instead, Arrow allows for a composition of static components of the parser with the dynamic parsing function itself. However, Arrow came about before Applicative, and McBride and Paterson [2008] note that Applicative is equivalent to an Arrow that carries a static component: this is exactly what the original *arrowised* parsers were achieving, showing instead that Applicative parser combinator libraries suffice for $LL(1)$ parsing. The original formulation also adds two additional typeclasses:

```
class Arrow (⇝) ⇒ ArrowChoice (⇝) where
  left  :: (a ⇝ b) → (Either a x ⇝ Either b x)
class Arrow (⇝) ⇒ ArrowApply (⇝)   where
  app :: (a ⇝ b, a) ⇝ b
```

Mokhov et al. [2019] mention that ArrowChoice, which provides arbitration between two different branches, is equivalent in power to Selective; and Hughes [2000] notes that ArrowApply recovers full monadic power, with app capable of implementing left. However, the monadic interface has prevailed as the dominant base API for parsing; it does not appear as if Arrow offers any practical advantage over Applicative for parsing.

Chapter 10

Conclusion

This dissertation has presented the design and development of the staged selective parser combinator library `parsley`; described general design principles for writing parsers with parser combinators; and improved on the error systems of `parsec` and `megaparsec`. It is worth remembering, however, how it began:

Parser combinators are a viable means of writing ***efficient*** *and* ***maintainable*** *parsers with* ***good error messages****.*

Haskell `parsley` – efficient CHAPTER 3 demonstrated how the Cayley transformation on regular expressions gives rise to an NFA, and further expanded on this to support heterogeneous results, recursion, `parsec`-style backtracking semantics and selective combinators. By staging the evaluation semantics of the derived machine, the abstraction overheads of the combinators are entirely eliminated, leaving code of near hand-written quality at a slight cost to expressive power.

CHAPTER 4 rectified this lost expressive power by introducing general parsing state called references. These are threaded through the parser and clearly delimit their scope with rank-2 types. These allow for optimisations to so-called iterative combinators and allow for fully context-sensitive parsers: benchmarks showed that these iterative combinators and references can form efficient tail-recursive parsers, an optimisation not so readily accessible for non-staged libraries.

CHAPTER 5 provided the implementation of a DIET set with full support for composite operations union, intersection, and difference. It also provided an efficient implementation of complement, which is used in the static analysis of parsers. CHAPTER 6 used this to define defunctionalised fully-inspectable character predicates and showed how to compile these to optimal predicates by staging the static structure of the set. By exploiting the laws of parsers, optimisations and simplifications are performed. The nature of a deep-embedded staged library allows for the luxury of more comprehensive static analysis, allowing for input to be factored out and the backtracking properties to be statically resolving. In all, benchmarks demonstrated that `parsley` has competitive performance compared to its modern contemporaries.

Design patterns for parser combinators – maintainable CHAPTER 7 presented several guiding design principles for writing maintainable and extensible parsers using parser combinators. Left recursion is cleanly abstracted using efficient iterative chain combinators, and generalised forms allow for improved type-safety of the parser. It established clear conventions for how whitespace should be consumed efficiently, as well as how to make distinctions between lexical tokens and parsing non-terminals allowing for clear separation of concerns. By introducing *Parser Bridges*, parsing code can be effectively decoupled from the construction of the AST allowing for the design of the overall system to be iterated on without breaking other parts. Patterns are further promoted by giving guidance on library level support: `parsley` Scala directly supports all the presented patterns, with further support coming for `parsley` Haskell in future.

Scala `parsley` – good error messages CHAPTER 8 provided several new primitive combinators for error messages within the `parsley` error system. This iterates on its predecessors, aiming to reduce counter-intuitive behaviours and maximising the potential evolution of the system without breakage. The performance of the library is retained by careful use of defunctionalisation. Additional patterns were presented that allow for highly specific error messages of handwritten quality, and these enjoy first-class support too within `parsley`.

Reflection

CHAPTER 1 gave several strengths and weaknesses of the parser combinator paradigm. Before revisiting these points, the initial example is revisited to reflect the developments within this dissertation.

Returning to the Verbose Lambda Calculus

The motivating example of parser combinators in CHAPTER 1 was the "verbose Lambda Calculus," as described in fig. 1.1a. The final code for the example can be easily translated into `parsley` Haskell:

```
commaSep p = p <:> many (tok "," *> p)
prog  = ⟦Prog⟧  <$> many (func <* tok ";") <*> expr
func  = ⟦Func⟧  <$> (tok "def" *> ident) <*> (tok "(" *> commaSep arg <* tok ")") <*> (tok "=" *> expr)
expr  = ⟦Ident⟧ <$> ident <|> ⟦Call⟧ <$> (tok "call" *> ident) <*> (tok "(" *> commaSep expr <* tok ")")
ident = some (oneOf ['a'..'z'])
```

The adjustments required to render this parser in `parsley` Haskell simply boil down to adding quotations around the relevant user-defined functions. However, by leveraging the design patterns of parser combinators, and the automatic generation of *Lifted Constructors*, the parser can be improved:

```
symbol sym | sym ∈ ["def", "call"] = token $ string sym *> notFollowedBy (oneOf ['a'..'z'])
           | otherwise             = token $ void (string sym)
instance u~() ⇒ IsString (Parser u) where fromString = symbol
$(deriveLiftedConstructors "mk" ['Prog, 'Func, 'Ident, 'Call])
```

While some of the standard functionality has been omitted for brevity, the important symbol combinator and lifted constructor derivations are present, along with the IsString instance. These can be deployed as follows:

```
commaSep p = p <:> many ("," *> p)
prog  = mkProg  (many (func <* ";")) expr
func  = mkFunc  ("def" *> ident) ("(" *> commaSep arg <* ")") ("=" *> expr)
expr  = mkIdent ident <|> mkCall ("call" *> ident) ("(" *> commaSep expr <* ")")
ident = some (oneOf ['a'..'z'])
```

The effect of these patterns on the parser is evident: the introduced quotes as part of adapting to `parsley`'s staged API have been eliminated, and the noise from combining the results has also disappeared, leaving a much cleaner and more concise implementation. This could be improved further, however, with Scala `parsley`.

In Scala parsley Scala `parsley` has a richer and more fully complete API than Haskell `parsley`, with functionality for lexer generation – this automatically embodies and implements the relevant lexing patterns. The lexing boilerplate and bridge generation is as follows:

```
val lexer = Lexer(LexicalDesc.plain.copy(
    nameDesc   = NameDesc.plain.copy(identifierStart  = predicate.Basic(_.isLetter).
                                     identifierLetter = predicate.Basic(_.isLetter)),
    symbolDesc = SymbolDesc.plain.copy(hardKeywords = Set("def", "call")),
))

object Prog  extends generic.ParserBridge2[List[Func], Expr, Prog]
object Func  extends generic.ParserBridge3[String, List[String], Expr, Func]
object Ident extends generic.ParserBridge1[String, Ident]
object Call  extends generic.ParserBridge2[String, List[Expr], Call]
```

Here, the `Lexer` class allows for the description of various lexical elements within the grammar, providing specific configured parsers as a result. The definition of the bridges, using the built-in generic bridges, is more verbose than the Template Haskell generator, but this is to be expected.

```
import lexer.lexeme.symbol.implicits.given // for String conversion

lazy val prog = Prog(endBy(func, ";"), expr)
lazy val func = Func("def" ~> identifier, "(" ~> commaSep(arg) <~ ")", "=" ~> expr)
lazy val expr: Parsley[Expr]
             = Ident(identifier) | Call("call" ~> identifier, "(" ~> commaSep(expr) <~ ")")
```

The resulting parser is subjectively cleaner than the Haskell equivalent: here, the names of the bridges are not augmented with a `"mk"` prefix, and more clearly delimit their arguments. The `commaSep` combinator along with `identifier` are both exposed by the `lexer` (the imports have been omitted) and handle all the relevant requirements. Unlike in Haskell, the type signature for `expr` is required as it is recursive.

Revisiting the Strengths and Weaknesses

How well does the work presented in this dissertation still uphold the strengths and address the weaknesses?

Strengths Haskell `parsley` remains as an **integrated** library with no additional build steps to compile and execute a parser, however, due to implementation restrictions, parsers can only be generated in a separate file than its definition. It also remains **flexible**, with no real effect on its usability other than the required interaction with Template Haskell when constructing user-defined predicates and values: however, the `parsley-garnish` helps to ease that burden. The **extensibility** of the library, however, is not as good as other contemporary libraries, with no way of introducing new primitives or interacting with the underlying internals – though composite combinators can still be defined freely. The introduction of references allows `parsley` to remain **powerful**, though is it more difficult for the parser writer to construct some context-sensitive parsers with selectives and references than it would be with monads and references. Because of the optimisations and analysis performed on parsers, `parsley` is less **transparent**, though ultimately the underlying changes to a grammar are minor.

Weaknesses Parser combinators are often more **verbose** than parser generators; the introduction of the design patterns, however, help mitigate this. Though `parsley` Haskell introduces metaprogramming noise into the parser, the *Bridge Pattern* – particularly the derivation of it – can hide this from the parser and reduce the interaction the user needs to have with Template Haskell, though it still may not be suitable for beginners. The design patterns also help with the **undirected** and **unstandardised** nature of parser combinator libraries, by providing guiding principles, and advice to library authors; however, this does not mean that different libraries will converge on a shared standard – though the two `parsley` libraries endeavour to share an API for cross-language consistency. Haskell `parsley` has shown that it is possible to eliminate much of the abstraction overheads of parser combinators, making them **faster**; while there is still a price to pay, as `parsley` still generates generic code, what it generates is much closer to handwritten quality. The error system of Scala `parsley` facilitates techniques for making more bespoke error messages and can produce **informative** tailored error messages; the *Verified* and *Preventative Errors* patterns can be employed in many different libraries with varying effectiveness, and these allow for much more intuitive error generation than can be achieved with a parser generator. That said, handwritten parsers still offer the highest degree of control over error messages.

Future Work

Though this dissertation has presented a full, working implementation of `parsley`, there is still work left to do:

- The error system of Scala `parsley` will be ported to the Haskell version, and static information about the character predicates can be used to stage away their construction, limiting code size increases from generated errors.
- Direct access to STRefs for references can be eliminated by caching known values statically so that they can be acquired directly in the generated code. Aggressive application of this can result in the eliminate the STRefs entirely within a parser fragment, resulting in generated code more like a monadic implementation.
- When generated code does not use specific bindings or arguments, they may remain if GHC does not optimise them. To mitigate this and unlock a new collection of aggressive dead-code elimination, global analysis can be performed to determine unused values ahead of time, representing them with more granular type indices and allowing for *staged fusion* to be employed further. This will also allow for the elimination of dead references, or conversion from STRef to direct argument threading where appropriate.
- Per-binding analysis will be introduced to compute the input consumption characteristics of bindings, allowing the static handler optimisation to reason about input inequality even when they come from different sources. This combined with the previous point will also allow for the static resolution of a handler even across a dynamic boundary, for instance passing a yesSame handler to another non-terminal.
- The work of chapters 3 to 6 will be ported to Scala 3 as a new `meta-parsley` library. The rich API improvements of `parsley` Scala will also be brought across to `parsley` Haskell; and for further standardisation, `gigaparsec` will iterate on `megaparsec` to achieve API parity across the libraries without requiring the heavier staging machinery.

- The hint system in `megaparsec`/`parsley` makes `<|>` non-associative (`Issue #412`). The `DefuncError` type is a semi-group, which means the error system as presented is fine, but the hints provide a local approximation of the best error, not a global one. Merging all errors globally makes `<|>` associative, but incurs massive space leaks; instead, a global merge should be done such that errors are garbage collected as soon as it provable they cannot form part of the final error – there are subtleties to account for with `amend`, however.

Closing Comments

Parser combinators have long been a favourite tool for functional programmers writing parsers. Despite their elegance and popularity, they have not managed to find use in many real-world compilers and applications, instead often replaced by handwritten recursive descent, which is their analogue. What this work has shown, however, is that the abstraction helping make parsers easy to write does not have to come at a cost to performance, nor do errors need be vague and uninformative. It is my hope that, with the continued support of these techniques and ideas, parser combinators will be a serious consideration for even the most demanding of workloads.

Bibliography

Adams, Michael D., Celeste Hollenbeck, and Matthew Might (June 2016). "On the Complexity and Performance of Parsing with Derivatives". In: *SIGPLAN Not.* 51.6, pp. 224–236. ISSN: 0362-1340. DOI: 10.1145/2980983.2908128. URL: http://doi.acm.org/10.1145/2980983.2908128.

Adams, Stephen (1992). *Implementing Sets Efficiently in a Functional Language*. Tech. rep. CSTR 92-10. University of Southhampton.

— (1993). "Functional Pearls Efficient sets—a balancing act". In: *Journal of Functional Programming* 3.4, pp. 553–561. DOI: 10.1017/S0956796800000885.

Adelson-Velskii, G and Evgenii Mikhailovich Landis (1962). *An algorithm for the organization of information*. Tech. rep. Joint Publications Research Service Washington DC.

Aho, Alfred V. et al. (2006). *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc. ISBN: 0321486811. DOI: 10.5555/1177220. URL: https://dl.acm.org/doi/10.5555/1177220.

Alstrup, Stephen et al. (1999). "Dominators in Linear Time". In: *SIAM Journal on Computing* 28.6, pp. 2117–2132. DOI: 10.1137/S0097539797317263. URL: https://doi.org/10.1137/S0097539797317263.

Amin, Nada et al. (2016). "The Essence of Dependent Object Types". In: *A List of Successes That Can Change the World: Essays Dedicated to Philip Wadler on the Occasion of His 60th Birthday*. Ed. by Sam Lindley et al. Cham: Springer International Publishing, pp. 249–272. ISBN: 978-3-319-30936-1. DOI: 10.1007/978-3-319-30936-1_14. URL: https://doi.org/10.1007/978-3-319-30936-1_14.

Appel, Andrew W. (2004). *Modern Compiler Implementation in ML*. USA: Cambridge University Press. ISBN: 0521607647.

— (2007). *Compiling with Continuations*. USA: Cambridge University Press. ISBN: 052103311X.

Baars, Arthur I. and S. Doaitse Swierstra (2004). "Type-Safe, Self Inspecting Code". In: *Proceedings of the 2004 ACM SIGPLAN Workshop on Haskell*. Haskell '04. Snowbird, Utah, USA: Association for Computing Machinery, pp. 69–79. ISBN: 1581138504. DOI: 10.1145/1017472.1017485. URL: https://doi.org/10.1145/1017472.1017485.

Bahr, Patrick and Graham Hutton (2015). "Calculating correct compilers". In: *Journal of Functional Programming* 25, p. 47. DOI: 10.1017/S0956796815000180.

— (Aug. 2020). "Calculating correct compilers II: Return of the register machines". In: *Journal of Functional Programming* 30. DOI: 10.1017/S0956796820000209.

— (Aug. 2022). "Monadic Compiler Calculation (Functional Pearl)". In: *Proc. ACM Program. Lang.* 6.ICFP. DOI: 10.1145/3547624. URL: https://doi.org/10.1145/3547624.

Benton, Nick (Nov. 2005). "A Typed, Compositional Logic for a Stack-Based Abstract Machine". In: pp. 364–380. DOI: 10.1007/11575467_24.

Berg, Mark et al. (2008). *Computational Geometry: Algorithms and Applications*. Berlin, Heidelberg: Springer Berlin Heidelberg, p. 386. ISBN: 978-3-540-77974-2. DOI: 10.1007/978-3-540-77974-2. URL: https://doi.org/10.1007/978-3-540-77974-2.

Bird, Richard and Ross Paterson (Sept. 1999). “Generalised Folds for Nested Datatypes”. In: *Form. Asp. Comput.* 11.2, pp. 200–222. ISSN: 0934-5043. DOI: 10.1007/s001650050047. URL: https://doi.org/10.1007/s001650050047.

Blelloch, Guy E., Daniel Ferizovic, and Yihan Sun (2016). “Just Join for Parallel Ordered Sets”. In: *Proceedings of the 28th ACM Symposium on Parallelism in Algorithms and Architectures*. SPAA ’16. Pacific Grove, California, USA: Association for Computing Machinery, pp. 253–264. ISBN: 9781450342100. DOI: 10.1145/2935764.2935768. URL: https://doi.org/10.1145/2935764.2935768.

Böhm, Corrado and Alessandro Berarducci (1985). “Automatic synthesis of typed Λ-programs on term algebras”. In: *Theoretical Computer Science* 39. Third Conference on Foundations of Software Technology and Theoretical Computer Science, pp. 135–154. ISSN: 0304-3975. DOI: 10.1016/0304-3975(85)90135-5. URL: https://doi.org/10.1016/0304-3975(85)90135-5.

Bondorf, Anders (1992). “Improving Binding Times without Explicit CPS-Conversion”. In: *SIGPLAN Lisp Pointers* V.1, pp. 1–10. ISSN: 1045-3563. DOI: 10.1145/141478.141483. URL: https://doi.org/10.1145/141478.141483.

Brzozowski, Janusz A. (Oct. 1964). “Derivatives of Regular Expressions”. In: *J. ACM* 11.4, pp. 481–494. ISSN: 0004-5411. DOI: 10.1145/321239.321249. URL: http://doi.acm.org/10.1145/321239.321249.

Burge, William H. (1975). *Recursive Programming Techniques*. Addison-Wesley.

Cayley, A. (1854). “VII. On the theory of groups, as depending on the symbolic equation θn =1”. In: *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science* 7.42, pp. 40–47. URL: https://doi.org/10.1080/14786445408647421.

Chomsky, Noam (1956). “Three models for the description of language”. In: *IRE Transactions on Information Theory* 2.3, pp. 113–124. DOI: 10.1109/TIT.1956.1056813.

— (1962). “Context-free grammars and pushdown storage”. In: *MIT Res. Lab. Electron. Quart. Prog. Report.* 65, pp. 187–194.

Church, Alonzo (1936). “An Unsolvable Problem of Elementary Number Theory”. In: *American Journal of Mathematics* 58, p. 345.

— (1941). “The calculi of lambda-conversion”. In: *Annals of Mathematics studies* 6.4. DOI: 10.2307/2267126.

Consel, Charles and Olivier Danvy (1991). “For a better support of static data flow”. In: *Functional Programming Languages and Computer Architecture*. Ed. by John Hughes. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 496–519. ISBN: 978-3-540-47599-6.

Cormen, Thomas H. et al. (2009). *Introduction to Algorithms, Third Edition*. 3rd. The MIT Press. ISBN: 0262033844.

Danielsson, Nils Anders (Sept. 2010). “Total Parser Combinators”. In: *SIGPLAN Not.* 45.9, pp. 285–296. ISSN: 0362-1340. DOI: 10.1145/1932681.1863585. URL: https://doi.org/10.1145/1932681.1863585.

Danielsson, Nils Anders and Ulf Norell (2011). “Parsing Mixfix Operators”. In: *Implementation and Application of Functional Languages*. Ed. by Sven-Bodo Scholz and Olaf Chitil. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 80–99. ISBN: 978-3-642-24452-0. DOI: https://doi.org/10.1007/978-3-642-24452-0_5.

Danvy, Olivier, Karoline Malmkjær, and Jens Palsberg (Nov. 1996). “Eta-expansion Does The Trick”. In: *ACM Trans. Program. Lang. Syst.* 18.6, pp. 730–751. ISSN: 0164-0925. DOI: 10.1145/236114.236119. URL: http://doi.acm.org/10.1145/236114.236119.

Danvy, Olivier and Lasse R. Nielsen (2001). “Defunctionalization at Work”. In: *Proceedings of the 3rd ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming*. PPDP ’01. Florence, Italy: Association for Computing Machinery, pp. 162–174. ISBN: 158113388X. DOI: 10.1145/773184.773202. URL: https://doi.org/10.1145/773184.773202.

Danvy, Olivier and Ulrik Pagh Schultz (2002). “Lambda-Lifting in Quadratic Time”. In: *Proceedings of the 6th International Symposium on Functional and Logic Programming*. FLOPS ’02. Berlin, Heidelberg: Springer-Verlag, pp. 134–151. ISBN: 3540442332.

de Bruijn, N.G (1972). “Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem”. In: *Indagationes Mathematicae (Proceedings)* 75.5, pp. 381–392. ISSN: 1385-7258. DOI: 10.1016/1385-7258(72)90034-0. URL: https://doi.org/10.1016/1385-7258(72)90034-0.

Delbianco, Germán Andrés, Mauro Jaskelioff, and Alberto Pardo (2012). “Applicative Shortcut Fusion”. In: *Trends in Functional Programming*. Ed. by Ricardo Peña and Rex Page. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 179–194. ISBN: 978-3-642-32037-8.

Deremer, Franklin L. (1969a). “Generating Parsers for BNF Grammars”. In: *Proceedings of the May 14-16, 1969, Spring Joint Computer Conference*. AFIPS ’69 (Spring). Boston, Massachusetts: ACM, pp. 793–799. DOI: 10.1145/1476793.1476928. URL: http://doi.acm.org/10.1145/1476793.1476928.

— (1969b). *Practical Translators For LR(K) Languages*. Tech. rep. USA. DOI: 10.5555/888578.

Devriese, Dominique and Frank Piessens (2011). “Explicitly Recursive Grammar Combinators”. In: *Practical Aspects of Declarative Languages*. Ed. by Ricardo Rocha and John Launchbury. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 84–98. ISBN: 978-3-642-18378-2.

— (2012). “Finally tagless observable recursion for an abstract grammar model”. In: *Journal of Functional Programming* 22.6, pp. 757–796. DOI: 10.1017/S0956796812000226.

Earley, Jay (Feb. 1970). “An Efficient Context-Free Parsing Algorithm”. In: *Commun. ACM* 13.2, pp. 94–102. ISSN: 0001-0782. DOI: 10.1145/362007.362035. URL: https://doi.org/10.1145/362007.362035.

Erkök, Levent and John Launchbury (2002). “A Recursive Do for Haskell”. In: *Proceedings of the 2002 ACM SIGPLAN Workshop on Haskell*. Haskell ’02. Pittsburgh, Pennsylvania: ACM, pp. 29–37. ISBN: 1-58113-605-6. DOI: 10.1145/581690.581693. URL: http://doi.acm.org/10.1145/581690.581693.

Erwig, Martin (1998). “Diets for fat sets”. In: *Journal of Functional Programming* 8.6, pp. 627–632. DOI: 10.1017/S0956796898003116.

Fokker, Jeroen (1995). “Functional Parsers”. In: *Advanced Functional Programming, First International Spring School on Advanced Functional Programming Techniques-Tutorial Text*. Berlin, Heidelberg: Springer-Verlag, pp. 1–23. ISBN: 3-540-59451-5. URL: http://dl.acm.org/citation.cfm?id=647698.734153.

Fokkinga, Maarten M. (1989). “Tupling and Mutumorphisms”. In.

Ford, Bryan (Sept. 2002). “Packrat Parsing : a Practical Linear-Time Algorithm with Backtracking”. Master’s Thesis. Massachusetts Institute of Technology, p. 112.

— (Jan. 2004). “Parsing Expression Grammars: A Recognition-based Syntactic Foundation”. In: *SIGPLAN Not.* 39.1, pp. 111–122. ISSN: 0362-1340. DOI: 10.1145/982962.964011. URL: http://doi.acm.org/10.1145/982962.964011.

Fowler, Martin (2010). *Domain Specific Languages*. 1st. Addison-Wesley Professional. ISBN: 0321712943. DOI: 10.5555/1809745. URL: https://dl.acm.org/doi/10.5555/1809745.

Fredkin, Edward (Sept. 1960). "Trie Memory". In: *Commun. ACM* 3.9, pp. 490–499. ISSN: 0001-0782. DOI: 10.1145/367390.367400. URL: https://doi.org/10.1145/367390.367400.

Frost, Richard A., Rahmatullah Hafiz, and Paul C. Callaghan (2007). "Modular and Efficient Top-down Parsing for Ambiguous Left-Recursive Grammars". In: *Proceedings of the 10th International Conference on Parsing Technologies*. IWPT '07. Prague, Czech Republic: Association for Computational Linguistics, pp. 109–120. ISBN: 9781932432909. DOI: https://dl.acm.org/doi/10.5555/1621410.1621425.

Gamma, Erich et al. (1995). *Design Patterns: Elements of Reusable Object-Oriented Software*. USA: Addison-Wesley Longman Publishing Co., Inc. ISBN: 0201633612. DOI: https://dl.acm.org/doi/book/10.5555/186897.

Gibbons, Jeremy (2002). "Calculating Functional Programs". In: *Algebraic and Coalgebraic Methods in the Mathematics of Program Construction*. Ed. by Roland Backhouse, Roy Crole, and Jeremy Gibbons. Vol. 2297. Lecture Notes in Computer Science. Springer-Verlag, pp. 148–203.

Gibbons, Jeremy and Ralf Hinze (Sept. 2011). "Just Do It: Simple Monadic Equational Reasoning". In: *SIGPLAN Not.* 46.9, pp. 2–14. ISSN: 0362-1340. DOI: 10.1145/2034574.2034777. URL: https://doi.org/10.1145/2034574.2034777.

Gibbons, Jeremy and Nicolas Wu (2014). "Folding Domain-specific Languages: Deep and Shallow Embeddings (Functional Pearl)". In: *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming*. ICFP '14. Gothenburg, Sweden: ACM, pp. 339–347. ISBN: 978-1-4503-2873-9. DOI: 10.1145/2628136.2628138. URL: http://doi.acm.org/10.1145/2628136.2628138.

Gill, Andy (2009). "Type-Safe Observable Sharing in Haskell". In: *Proceedings of the 2nd ACM SIGPLAN Symposium on Haskell*. Haskell '09. Edinburgh, Scotland: Association for Computing Machinery, pp. 117–128. ISBN: 9781605585086. DOI: 10.1145/1596638.1596653. URL: https://doi.org/10.1145/1596638.1596653.

Gill, Andy, John Launchbury, and Simon L. Peyton Jones (1993). "A Short Cut to Deforestation". In: *Proceedings of the Conference on Functional Programming Languages and Computer Architecture*. FPCA '93. Copenhagen, Denmark: Association for Computing Machinery, pp. 223–232. ISBN: 089791595X. DOI: 10.1145/165180.165214. URL: https://doi.org/10.1145/165180.165214.

Gill, Andy and Simon Marlow (1995). *Happy: the parser generator for Haskell*.

Goguen, J. A. and J. W. Thatcher (1974). "Initial algebra semantics". In: *15th Annual Symposium on Switching and Automata Theory (swat 1974)*, pp. 63–77. DOI: 10.1109/SWAT.1974.13.

Hagino, Tatsuya (1987). "Category theoretic approach to data types". PhD thesis. PhD thesis, University of Edinburgh.

Henriksen, Ian, Gianfranco Bilardi, and Keshav Pingali (Oct. 2019). "Derivative Grammars: A Symbolic Approach to Parsing with Derivatives". In: *Proc. ACM Program. Lang.* 3.OOPSLA. DOI: 10.1145/3360553. URL: https://doi.org/10.1145/3360553.

Hill, Steve (Nov. 1994). *Continuation Passing Combinators for Parsing Precedence Grammars*. Technical report. University of Kent, Canterbury, UK: University of Kent, Computing Laboratory. URL: https://kar.kent.ac.uk/21168/.

Hill, Steve (1996). "Combinators for parsing expressions". In: *Journal of Functional Programming* 6.3, pp. 445–464. DOI: 10.1017/S0956796800001799.

Hinze, Ralf (2012). "Kan Extensions for Program Optimisation Or: Art and Dan Explain an Old Trick". In: *Mathematics of Program Construction*. Ed. by Jeremy Gibbons and Pablo Nogueira. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 324–362. ISBN: 978-3-642-31113-0.

Hinze, Ralf and Nicolas Wu (2013). "Histo- and Dynamorphisms Revisited". In: *Proceedings of the 9th ACM SIGPLAN Workshop on Generic Programming*. WGP '13. Boston, Massachusetts, USA: Association for Computing Machinery, pp. 1–12. ISBN: 9781450323895. DOI: 10.1145/2502488.2502496. URL: https://doi.org/10.1145/2502488.2502496.

Hinze, Ralf, Nicolas Wu, and Jeremy Gibbons (2013). "Unifying Structured Recursion Schemes". In: *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming*. ICFP '13. Boston, Massachusetts, USA: Association for Computing Machinery, pp. 209–220. ISBN: 9781450323260. DOI: 10.1145/2500365.2500578. URL: https://doi.org/10.1145/2500365.2500578.

Hoffman, Paul E. and François Yergeau (Feb. 2000). *UTF-16, an encoding of ISO 10646*. RFC 2781. DOI: 10.17487/RFC2781. URL: https://www.rfc-editor.org/info/rfc2781.

Hopcroft, John E, Rajeev Motwani, and Jeffrey D Ullman (2001). "Introduction to automata theory, languages, and computation". In: *Acm Sigact News* 32.1, pp. 60–65.

Hudak, Paul (Dec. 1996). "Building Domain-specific Embedded Languages". In: *ACM Comput. Surv.* 28.4es. ISSN: 0360-0300. DOI: 10.1145/242224.242477. URL: http://doi.acm.org/10.1145/242224.242477.

Hughes, John (July 1983). *The Design and Implementation of Programming Languages*. Tech. rep. PRG40. OUCL, p. 159.

— (1986). "A novel representation of lists and its application to the function "reverse"". In: *Information Processing Letters* 22.3, pp. 141–144. ISSN: 0020-0190. DOI: https://doi.org/10.1016/0020-0190(86)90059-1. URL: https://www.sciencedirect.com/science/article/pii/0020019086900591.

— (2000). "Generalising monads to arrows". In: *Science of Computer Programming* 37.1, pp. 67–111. ISSN: 0167-6423. DOI: https://doi.org/10.1016/S0167-6423(99)00023-4. URL: https://www.sciencedirect.com/science/article/pii/S0167642399000234.

Hughes, John. and S. Doaitse Swierstra (Aug. 2003). "Polish Parsers, Step by Step". In: *SIGPLAN Not.* 38.9, pp. 239–248. ISSN: 0362-1340. DOI: 10.1145/944746.944727. URL: https://doi.org/10.1145/944746.944727.

Hutton, Graham (1992). "Higher-order functions for parsing". In: *Journal of Functional Programming* 2.3, pp. 323–343. DOI: 10.1017/S0956796800000411.

— (1999). "A tutorial on the universality and expressiveness of fold". In: *Journal of Functional Programming* 9.4, pp. 355–372. DOI: 10.1017/S0956796899003500.

— (2016). *Programming in Haskell*. 2nd. USA: Cambridge University Press. ISBN: 1316626229.

Hutton, Graham and Erik Meijer (1996). *Monadic Parser Combinators*. Tech. rep. NOTTCS-TR-96-4. Department of Computer Science, University of Nottingham. DOI: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.54.1678.

Izmaylova, Anastasia, Ali Afroozeh, and Tijs van der Storm (2016). "Practical, General Parser Combinators". In: *Proceedings of the 2016 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation*. PEPM

’16. St. Petersburg, FL, USA: Association for Computing Machinery, pp. 1–12. ISBN: 9781450340977. DOI: 10.1145/2847538.2847539. URL: https://doi.org/10.1145/2847538.2847539.

Johnson, Mark (Sept. 1995). “Memoization in Top-down Parsing”. In: *Comput. Linguist.* 21.3, pp. 405–417. ISSN: 0891-2017. DOI: https://dl.acm.org/doi/10.5555/216261.216269.

Johnsson, Thomas (1985). “Lambda lifting: Transforming programs to recursive equations”. In: *Functional Programming Languages and Computer Architecture.* Ed. by Jean-Pierre Jouannaud. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 190–203. ISBN: 978-3-540-39677-2.

Jones, Mark P. (1995). “A system of constructor classes: overloading and implicit higher-order polymorphism”. In: *Journal of Functional Programming* 5.1, pp. 1–35. DOI: 10.1017/S0956796800001210.

Jones, N.D., C.K. Gomard, and P. Sestoft (1993). *Partial Evaluation and Automatic Program Generation.* Prentice-Hall international series in computer science. Prentice Hall. ISBN: 9780130202499. URL: https://books.google.co.uk/books?id=7rPPScYo8w8C.

Jonnalagedda, Manohar et al. (Oct. 2014). “Staged Parser Combinators for Efficient Data Processing”. In: *SIGPLAN Not.* 49.10, pp. 637–653. ISSN: 0362-1340. DOI: 10.1145/2714064.2660241. URL: http://doi.acm.org/10.1145/2714064.2660241.

Kennedy, Andrew (2007). “Compiling with Continuations, Continued”. In: *Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming.* ICFP ’07. Freiburg, Germany: Association for Computing Machinery, pp. 177–190. ISBN: 9781595938152. DOI: 10.1145/1291151.1291179. URL: https://doi.org/10.1145/1291151.1291179.

Kiss, Csongor, Tony Field, et al. (July 2019). “Higher-Order Type-Level Programming in Haskell”. In: *Proc. ACM Program. Lang.* 3.ICFP. DOI: 10.1145/3341706. URL: https://doi.org/10.1145/3341706.

Kiss, Csongor, Matthew Pickering, and Nicolas Wu (July 2018). “Generic Deriving of Generic Traversals”. In: *Proc. ACM Program. Lang.* 2.ICFP. DOI: 10.1145/3236780. URL: https://doi.org/10.1145/3236780.

Kleene, Stephen Cole (1952). *Introduction to Metamathematics.* Princeton, NJ, USA: North Holland. DOI: 10.2307/2268620. URL: https://doi.org/10.2307/2268620.

Kleene, Stephen Cole et al. (1956). “Representation of events in nerve nets and finite automata”. In: *Automata studies* 34, pp. 3–41.

Knuth, Donald E. (1965). “On the translation of languages from left to right”. In: *Information and Control* 8.6, pp. 607–639. ISSN: 0019-9958. DOI: 10.1016/S0019-9958(65)90426-2. URL: https://doi.org/10.1016/S0019-9958(65)90426-2.

Kövesdán, Gábor, Márk Asztalos, and László Lengyel (2014). “Architectural design patterns for language parsers”. In: *Acta Polytechnica Hungarica* 11.5, pp. 39–57. DOI: https://doi.org/10.12700/aph.11.05.2014.05.3.

Kozen, Dexter (May 1997). “Kleene Algebra with Tests”. In: *ACM Trans. Program. Lang. Syst.* 19.3, pp. 427–443. ISSN: 0164-0925. DOI: 10.1145/256167.256195. URL: https://doi.org/10.1145/256167.256195.

Krishnaswami, Neelakantan R. and Jeremy Yallop (2019). “A Typed, Algebraic Approach to Parsing”. In: *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation.* PLDI 2019. Phoenix, AZ, USA: ACM, pp. 379–393. ISBN: 978-1-4503-6712-7. DOI: 10.1145/3314221.3314625. URL: http://doi.acm.org/10.1145/3314221.3314625.

Kurš, Jan et al. (2016). "Optimizing Parser Combinators". In: *Proceedings of the 11th Edition of the International Workshop on Smalltalk Technologies.* IWST'16. Prague, Czech Republic: Association for Computing Machinery. ISBN: 9781450345248. DOI: 10.1145/2991041.2991042. URL: https://doi.org/10.1145/2991041.2991042.

Launchbury, John and Simon L. Peyton Jones (June 1994). "Lazy Functional State Threads". In: *SIGPLAN Not.* 29.6, pp. 24–35. ISSN: 0362-1340. DOI: 10.1145/773473.178246. URL: http://doi.acm.org/10.1145/773473.178246.

Leijen, Daan and Erik Meijer (Dec. 1999). "Domain Specific Embedded Compilers". In: *SIGPLAN Not.* 35.1, pp. 109–122. ISSN: 0362-1340. DOI: 10.1145/331963.331977. URL: http://doi.acm.org/10.1145/331963.331977.

— (2001). *Parsec: Direct Style Monadic Parser Combinators For The Real World.* Tech. rep. Microsoft. DOI: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.19.5187.

Lewis, P. M. and R. E. Stearns (July 1968). "Syntax-Directed Transduction". In: *J. ACM* 15.3, pp. 465–488. ISSN: 0004-5411. DOI: 10.1145/321466.321477. URL: https://doi.org/10.1145/321466.321477.

Lickman, P. (1995). "Parsing with Fixed Points". Master's Thesis. Oxford University, p. 63.

Lindley, Sam and Conor McBride (Sept. 2013). "Hasochism: The Pleasure and Pain of Dependently Typed Haskell Programming". In: *SIGPLAN Not.* 48.12, pp. 81–92. ISSN: 0362-1340. DOI: 10.1145/2578854.2503786. URL: https://doi.org/10.1145/2578854.2503786.

Ljunglöf, Peter (Mar. 2002). "Pure Functional Parsing". PhD thesis. Chalmers University of Technology and Göteborg University.

Magalhães, José Pedro and Andres Löh (2014). "Generic Generic Programming". In: *Proceedings of the 16th International Symposium on Practical Aspects of Declarative Languages - Volume 8324.* PADL 2014. San Diego, CA, USA: Springer-Verlag, pp. 216–231. ISBN: 9783319041315. DOI: 10.1007/978-3-319-04132-2_15. URL: https://doi.org/10.1007/978-3-319-04132-2_15.

Maidl, André Murbach et al. (2016). "Error reporting in Parsing Expression Grammars". In: *Science of Computer Programming* 132. Selected and extended papers from SBLP 2013, pp. 129–140. ISSN: 0167-6423. DOI: https://doi.org/10.1016/j.scico.2016.08.004. URL: https://www.sciencedirect.com/science/article/pii/S0167642316301046.

Marlow, Simon et al. (2010). "Haskell 2010 language report". In: URL: https://www.haskell.org/onlinereport/haskell2010/.

Mascarenhas, Fabio, Sérgio Medeiros, and Roberto Ierusalimschy (2014). "On the relation between context-free grammars and parsing expression grammars". In: *Science of Computer Programming* 89, pp. 235–250. ISSN: 0167-6423. DOI: https://doi.org/10.1016/j.scico.2014.01.012. URL: https://www.sciencedirect.com/science/article/pii/S0167642314000276.

Maurer, Luke et al. (2017). "Compiling without continuations". In: *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017.* Ed. by Albert Cohen and Martin T. Vechev. ACM, pp. 482–494. DOI: 10.1145/3062341.3062380. URL: https://doi.org/10.1145/3062341.3062380.

McBride, Conor (2011). "Functional pearl: Kleisli arrows of outrageous fortune". In: *Journal of Functional Programming (accepted for publication).*

McBride, Conor and Ross Paterson (2008). "Applicative programming with effects". In: *Journal of Functional Programming* 18.1, pp. 1–13. doi: 10.1017/S0956796807006326.

McCracken, Nancy (1984). "The Typechecking of Programs with Implicit Type Structure." In: *Proc. Of the International Symposium on Semantics of Data Types*. Sophia-Antipolis, France: Springer-Verlag New York, Inc., pp. 301–315. isbn: 3-540-13346-1. url: http://dl.acm.org/citation.cfm?id=1096.1107.

Mealy, George H. (1955). "A method for synthesizing sequential circuits". In: *The Bell System Technical Journal* 34.5, pp. 1045–1079. doi: 10.1002/j.1538-7305.1955.tb03788.x.

Meertens, Lambert (1988). "First steps towards the theory of rose trees". In: *CWI, Amsterdam*.

Might, Matthew, David Darais, and Daniel Spiewak (2011). "Parsing with Derivatives: A Functional Pearl". In: *Proceedings of the 16th ACM SIGPLAN International Conference on Functional Programming*. ICFP '11. Tokyo, Japan: Association for Computing Machinery, pp. 189–195. isbn: 9781450308656. doi: 10.1145/2034773.2034801. url: https://doi.org/10.1145/2034773.2034801.

Mokhov, Andrey et al. (July 2019). "Selective Applicative Functors". In: *Proc. ACM Program. Lang.* 3.ICFP, 90:1–90:29. issn: 2475-1421. doi: 10.1145/3341694. url: http://doi.acm.org/10.1145/3341694.

Moore, Edward F. (1956). "Gedanken-Experiments on Sequential Machines". In: *Automata Studies. (AM-34), Volume 34*. Ed. by C. E. Shannon and J. McCarthy. Princeton: Princeton University Press, pp. 129–154. isbn: 9781400882618. doi: doi:10.1515/9781400882618-006. url: https://doi.org/10.1515/9781400882618-006.

Moore, Robert C. (2000). "Removing Left Recursion from Context-Free Grammars". In: *Proceedings of the 1st North American Chapter of the Association for Computational Linguistics Conference*. NAACL 2000. Seattle, Washington: Association for Computational Linguistics, pp. 249–255. doi: https://dl.acm.org/doi/10.5555/974305.974338.

Morazán, Marco T. and Ulrik P. Schultz (2008). "Optimal Lambda Lifting in Quadratic Time". In: *Implementation and Application of Functional Languages*. Ed. by Olaf Chitil, Zoltán Horváth, and Viktória Zsók. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 37–56. isbn: 978-3-540-85373-2.

Morrisett, Greg et al. (Jan. 2002). "Stack-Based Typed Assembly Language". In: *J. Funct. Program.* 12.1, pp. 43–88. issn: 0956-7968. doi: 10.1017/S0956796801004178. url: https://doi.org/10.1017/S0956796801004178.

Morrison, Donald R. (Oct. 1968). "PATRICIA—Practical Algorithm To Retrieve Information Coded in Alphanumeric". In: *J. ACM* 15.4, pp. 514–534. issn: 0004-5411. doi: 10.1145/321479.321481. url: https://doi.org/10.1145/321479.321481.

Nguyen, Dung "Zung", Mathias Ricken, and Stephen Wong (Feb. 2005). "Design Patterns for Parsing". In: *SIGCSE Bull.* 37.1, pp. 477–481. issn: 0097-8418. doi: 10.1145/1047124.1047497. url: https://doi.org/10.1145/1047124.1047497.

Nielson, Flemming and Hanne Riis Nielson (1992). *Two-Level Functional Languages*. USA: Cambridge University Press. isbn: 0521403847.

Nievergelt, J. and E. M. Reingold (1972). "Binary Search Trees of Bounded Balance". In: *Proceedings of the Fourth Annual ACM Symposium on Theory of Computing*. STOC '72. Denver, Colorado, USA: Association for Computing Machinery, pp. 137–142. isbn: 9781450374576. doi: 10.1145/800152.804906. url: https://doi.org/10.1145/800152.804906.

Odersky, Martin et al. (2019). *The Scala language specification (2.13).* URL: https://www.scala-lang.org/files/archive/spec/2.13/.

Okasaki, Chris and Andy Gill (Mar. 2000). "Fast Mergeable Integer Maps". In: DOI: 10.1.1.37.5452.

Parr, Terence (2013). *The Definitive ANTLR 4 Reference.* 2nd. Pragmatic Bookshelf. ISBN: 1934356999. DOI: https://dl.acm.org/doi/10.5555/2501720.

Partridge, Andrew and David Wright (1996). "Predictive parser combinators need four values to report errors". In: *Journal of Functional Programming* 6.2, pp. 355–364. DOI: 10.1017/S0956796800001714.

Peyton Jones, Simon L. (1987). *The implementation of functional programming languages.* Prentice-Hall, Inc. ISBN: 0-13-453325-9.

Peyton Jones, Simon L., Mark Jones, and Erik Meijer (Jan. 1997). "Type classes: an exploration of the design space". In: *Haskell workshop.* Haskell workshop. URL: https://www.microsoft.com/en-us/research/publication/type-classes-an-exploration-of-the-design-space/.

Pfenning, F. and C. Elliott (June 1988). "Higher-Order Abstract Syntax". In: *SIGPLAN Not.* 23.7, pp. 199–208. ISSN: 0362-1340. DOI: 10.1145/960116.54010. URL: https://doi.org/10.1145/960116.54010.

Pickard, Mitchell and Graham Hutton (Aug. 2021). "Calculating Dependently-Typed Compilers (Functional Pearl)". In: *Proc. ACM Program. Lang.* 5.ICFP. DOI: 10.1145/3473587. URL: https://doi.org/10.1145/3473587.

Pickering, Matthew (2021). "Understanding the interaction between elaboration and quotation". PhD thesis. University of Bristol.

Plasmeijer, Marinus and Marko Eekelen (Jan. 1993). *Functional Programming and Parallel Graph Rewriting.* ISBN: 0-201-41663-8.

Rabin, M. O. and D. Scott (1959). "Finite Automata and Their Decision Problems". In: *IBM Journal of Research and Development* 3.2, pp. 114–125. DOI: 10.1147/rd.32.0114.

Reynolds, John C. (1972). "Definitional Interpreters for Higher-Order Programming Languages". In: *Proceedings of the ACM Annual Conference - Volume 2.* ACM '72. Boston, Massachusetts, USA: Association for Computing Machinery, pp. 717–740. ISBN: 9781450374927. DOI: 10.1145/800194.805852. URL: https://doi.org/10.1145/800194.805852.

— (Nov. 1993). "The Discoveries of Continuations". In: *Lisp Symb. Comput.* 6.3–4, pp. 233–248. ISSN: 0892-4635. DOI: 10.1007/BF01019459. URL: https://doi.org/10.1007/BF01019459.

Rivas, Exequiel and Mauro Jaskelioff (2014). "Notions of Computation as Monoids". In: *CoRR* abs/1406.4823. arXiv: 1406.4823. URL: http://arxiv.org/abs/1406.4823.

Rompf, Tiark and Martin Odersky (Oct. 2010). "Lightweight Modular Staging: A Pragmatic Approach to Runtime Code Generation and Compiled DSLs". In: *SIGPLAN Not.* 46.2, pp. 127–136. ISSN: 0362-1340. DOI: 10.1145/1942788.1868314. URL: http://doi.acm.org/10.1145/1942788.1868314.

Schreiner, Axel-Tobias and James Heliotis (2008). "Design Patterns in Parsing". Presented at Killer Examples, a workshop at OOPSLA '08. DOI: https://scholarworks.rit.edu/other/82/.

Schrijvers, Tom et al. (Sept. 2008). "Type Checking with Open Type Functions". In: *SIGPLAN Not.* 43.9, pp. 51–62. ISSN: 0362-1340. DOI: 10.1145/1411203.1411215. URL: https://doi.org/10.1145/1411203.1411215.

Scott, Elizabeth and Adrian Johnstone (2010). "GLL parsing". In: *Electronic Notes in Theoretical Computer Science* 253.7, pp. 177–189.

Scott, Elizabeth and Adrian Johnstone (2013). “GLL parse-tree generation”. In: *Science of Computer Programming* 78.10, pp. 1828–1844.

Sheard, Tim and Simon L. Peyton Jones (Dec. 2002). “Template Meta-programming for Haskell”. In: *SIGPLAN Not.* 37.12, pp. 60–75. ISSN: 0362-1340. DOI: 10.1145/636517.636528. URL: http://doi.acm.org/10.1145/636517.636528.

Sipser, Michael (2013). *Introduction to the Theory of Computation.* Third. Boston, MA: Course Technology. ISBN: 113318779X.

Spiewak, Daniel (2010). “Generalized Parser Combinators”. In.

Spivey, Mike (1990). “A functional theory of exceptions”. In: *Science of Computer Programming* 14.1, pp. 25–42. ISSN: 0167-6423. DOI: https://doi.org/10.1016/0167-6423(90)90056-J. URL: https://www.sciencedirect.com/science/article/pii/016764239090056J.

Steele, Guy L. (1978). *Rabbit: A Compiler for Scheme.* Tech. rep. USA.

Steele, Guy L. and Gerald J Sussman (1976). *Lambda: The Ultimate Imperative.* Tech. rep. USA.

Stucki, Nicolas, Aggelos Biboudis, and Martin Odersky (Apr. 2020). “A Practical Unification of Multi-Stage Programming and Macros”. In: *SIGPLAN Not.* 53.9, pp. 14–27. ISSN: 0362-1340. DOI: 10.1145/3393934.3278139. URL: https://doi.org/10.1145/3393934.3278139.

Sussman, Gerald J and Guy L. Steele (Dec. 1998). “Scheme: A Interpreter for Extended Lambda Calculus”. In: *Higher-Order and Symbolic Computation* 11, pp. 405–439. DOI: 10.1023/A:1010035624696.

Swierstra, S. Doaitse (2009). “Combinator Parsing: A Short Tutorial”. In: *Language Engineering and Rigorous Software Development: International LerNet ALFA Summer School 2008, Piriapolis, Uruguay, February 24 - March 1, 2008, Revised Tutorial Lectures.* Ed. by Ana Bove et al. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 252–300. ISBN: 978-3-642-03153-3. DOI: 10.1007/978-3-642-03153-3_6. URL: https://doi.org/10.1007/978-3-642-03153-3_6.

Swierstra, S. Doaitse and Luc Duponcheel (1996). “Deterministic, Error-Correcting Combinator Parsers”. In: *Advanced Functional Programming, Second International School-Tutorial Text.* London, UK: Springer-Verlag, pp. 184–207. ISBN: 3-540-61628-4. URL: http://dl.acm.org/citation.cfm?id=647699.734159.

Taha, Walid (2004). “A Gentle Introduction to Multi-stage Programming”. In: *Domain-Specific Program Generation: International Seminar, Dagstuhl Castle, Germany, March 23-28, 2003. Revised Papers.* Ed. by Christian Lengauer et al. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 30–50. ISBN: 978-3-540-25935-0. DOI: 10.1007/978-3-540-25935-0_3. URL: https://doi.org/10.1007/978-3-540-25935-0_3.

Taha, Walid and Tim Sheard (Dec. 1997). “Multi-stage Programming with Explicit Annotations”. In: *SIGPLAN Not.* 32.12, pp. 203–217. ISSN: 0362-1340. DOI: 10.1145/258994.259019. URL: http://doi.acm.org/10.1145/258994.259019.

Tarjan, Robert Endre (June 1976). “Edge-disjoint spanning trees and depth-first search”. English (US). In: *Acta Informatica* 6.2, pp. 171–185. ISSN: 0001-5903. DOI: 10.1007/BF00268499.

Thompson, Ken (1968). “Programming techniques: Regular expression search algorithm”. In: *Communications of the ACM* 11.6, pp. 419–422.

Uustalu, Tarmo and Varmo Vene (1999). “Primitive (Co)Recursion and Course-of-Value (Co)Iteration, Categorically”. In: *Informatica* 10, pp. 5–26. DOI: 10.3233/INF-1999-10102. URL: https://doi.org/10.3233/INF-1999-10102.

Voigtländer, Janis (2008). “Asymptotic Improvement of Computations over Free Monads”. In: *Mathematics of Program Construction*. Ed. by Philippe Audebaud and Christine Paulin-Mohring. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 388–403. ISBN: 978-3-540-70594-9.

Wadler, Philip (1985). “How to replace failure by a list of successes a method for exception handling, backtracking, and pattern matching in lazy functional languages”. In: *Functional Programming Languages and Computer Architecture*. Ed. by Jean-Pierre Jouannaud. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 113–128. ISBN: 978-3-540-39677-2. DOI: https://doi.org/10.1007/3-540-15975-4_33.

— (Jan. 1988). “Deforestation: Transforming Programs to Eliminate Trees”. In: *Theor. Comput. Sci.* 73.2, pp. 231–248. ISSN: 0304-3975. DOI: 10.1016/0304-3975(90)90147-A. URL: https://doi.org/10.1016/0304-3975(90)90147-A.

— (1992). “Comprehending Monads”. In: *Mathematical Structures in Computer Science*, pp. 61–78.

— (1995). “Monads for Functional Programming”. In: *Advanced Functional Programming, First International Spring School on Advanced Functional Programming Techniques-Tutorial Text*. Berlin, Heidelberg: Springer-Verlag, pp. 24–52. ISBN: 3540594515.

Willis, Jamie (2018). “Parsley: The Fastest Parser Combinator Library in the West”. Master’s Thesis. University of Bristol, p. 62.

Willis, Jamie and Nicolas Wu (2018). “Garnishing Parsec with Parsley”. In: *Proceedings of the 9th ACM SIGPLAN International Symposium on Scala*. Scala ’18. St. Louis, MO, USA: ACM, pp. 24–34. ISBN: 978-1-4503-5836-1. DOI: 10.1145/3241653.3241656. URL: http://doi.acm.org/10.1145/3241653.3241656.

— (2021). “Design Patterns for Parser Combinators (Functional Pearl)”. In: *Proceedings of the 14th ACM SIGPLAN International Symposium on Haskell*. Haskell 2021. Virtual, Republic of Korea: Association for Computing Machinery, pp. 71–84. ISBN: 9781450386159. DOI: 10.1145/3471874.3472984. URL: https://doi.org/10.1145/3471874.3472984.

— (2022). “Design Patterns for Parser Combinators in Scala”. In: *Proceedings of the Scala Symposium*. Scala ’22. Berlin, Germany: Association for Computing Machinery, pp. 9–21. ISBN: 9781450394635. DOI: 10.1145/3550198.3550427. URL: https://doi.org/10.1145/3550198.3550427.

Willis, Jamie, Nicolas Wu, and Matthew Pickering (Aug. 2020). “Staged Selective Parser Combinators”. In: *Proc. ACM Program. Lang.* 4.ICFP. DOI: 10.1145/3409002. URL: https://doi.org/10.1145/3409002.

Willis, Jamie, Nicolas Wu, and Tom Schrijvers (2022). “Oregano: Staging Regular Expressions with Moore Cayley Fusion”. In: *Proceedings of the 15th ACM SIGPLAN International Haskell Symposium*. Haskell 2022. Ljubljana, Slovenia: Association for Computing Machinery, pp. 66–80. ISBN: 9781450394383. DOI: 10.1145/3546189.3549916. URL: https://doi.org/10.1145/3546189.3549916.

Wu, Nicolas, Tom Schrijvers, and Ralf Hinze (2014). “Effect Handlers in Scope”. In: *Proceedings of the 2014 ACM SIGPLAN Symposium on Haskell*. Haskell ’14. Gothenburg, Sweden: Association for Computing Machinery, pp. 1–12. ISBN: 9781450330411. DOI: 10.1145/2633357.2633358. URL: https://doi.org/10.1145/2633357.2633358.

Yallop, Jeremy (Aug. 2017). “Staged Generic Programming”. In: *Proc. ACM Program. Lang.* 1.ICFP, 29:1–29:29. ISSN: 2475-1421. DOI: 10.1145/3110273. URL: http://doi.acm.org/10.1145/3110273.

Yallop, Jeremy, Tamara von Glehn, and Ohad Kammar (July 2018). “Partially-Static Data as Free Extension of Algebras”. In: *Proc. ACM Program. Lang.* 2.ICFP. DOI: 10.1145/3236795. URL: https://doi.org/10.1145/3236795.

Yallop, Jeremy and Oleg Kiselyov (2019). “Generating Mutually Recursive Definitions”. In: *Proceedings of the 2019 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation.* PEPM 2019. Cascais, Portugal: ACM, pp. 75–81. ISBN: 978-1-4503-6226-9. DOI: 10.1145/3294032.3294078. URL: http://doi.acm.org/10.1145/3294032.3294078.

Yang, Zhixuan and Nicolas Wu (2022). “Fantastic Morphisms and Where to Find Them”. In: *Mathematics of Program Construction.* Ed. by Ekaterina Komendantskaya. Cham: Springer International Publishing, pp. 222–267. ISBN: 978-3-031-16912-0.

Appendix A

Scala for Haskell Programmers

While not much Scala knowledge is needed to be able to read this dissertation, some concepts may be unfamiliar to readers. The version used within the dissertation is **Scala 3.3.0**, and indentation sensitive syntax is used often to help de-clutter. As of writing, no official specification for Scala 3 exists, however, much of the Scala 2.13 specification [Odersky et al. 2019] still applies.

A.1 Classes, Traits, Objects, and Enums

Scala is an object-oriented and functional language. This means that whilst all values in the language are objects (at least at the language level), and as functions are values, they too are objects. As such, OO features such as classes make a frequent appearance. Classes in Scala are fairly standard, and are a collection of attributes and methods associated to a type. Classes can extend up to one other class and zero or more traits. Classes may be marked as `abstract`, which means they are not designed to be instantiated themselves and are designed to serve as a superclass of another concrete type.

Traits in Scala are like Java's interfaces: they are a collection of methods that can be extended by another type with `extends`. This is used to form generic uniform method APIs to a collection of similar types. In a Haskell versus Scala sense, traits and inheritance can be thought of as an alternative to typeclasses, which provide ad-hoc polymorphism instead of subtype polymorphism. In appendix A.4.1, we learn that traits can be used to implement ad-hoc polymorphism as well, using Scala's implicits.

`sealed` and `final` Classes can be marked with `final` to denote that they cannot be extended by another class. Both classes and traits can be marked with `sealed` to denote that all subtypes must be found within the same file: this allows for the construction of closed ADTs and exhaustivity checking in pattern matches.

Case classes In Scala, a class can be marked with a `case` prefix to have the compiler synthesise some functionality [Odersky et al. 2019, §5.3.2]. This means automatically deriving the equivalent of both of Haskell's `Eq` and `Show`, as well as enabling pattern matching on values of the class and creating a factory `apply` method on the *companion object*: this allows for `Foo(x, y)` for object construction instead of `new Foo(x, y)`.

A.1.1 Objects and Companion Objects

Scala has standalone singleton objects with the `object` keyword. These can extend other types, like classes or traits, but may not have generic parameters and cannot be instantiated multiple times. There is a special case where an object and class in the same file share the same name, making the object a *companion object* [Odersky et al. 2019, §5.5]: in a Java sense, this means that it holds all the static members of the class. I refer to methods on companion object as "functions" and not "methods", since morally they do not have a receiver that varies.

A.1.2 Enums

While it is possible to construct complex ADTs in Scala with case classes, objects, and sealed abstract classes/traits, Scala 3 offers an alternative more concise and convenient syntax for defining them, called enums. These are not as expressive as the former[1], but they cover most use-cases well. As an example, here are two ways to define the `Option` type, the equivalent of Haskell's `Maybe`:

```
// Scala 2 style ADT
sealed abstract class Option[+A]
case class Some[A](x: A) extends Option[A]
case object None extends Option[Nothing]
```

```
// Scala 3 style ADT with enum
enum Option[+A]:
    case Some(x: A)
    case None
```

In this case, both definitions have the same expressive power: the `Some` and `None` "constructors" are types in their own right and are both subtypes of `Option`. The covariance of `Option` (denoted by the `+`) means that `None` has type `Option[Nothing]`, with `Nothing` a subtype of every other type. The main difference is that one must write `Option.Some(6)` in the enum version and `Some(6)` in the flat version; this can be avoided by importing the constructors with `import Option.*`, however.

Enums can serve as GADTs as well, like in Haskell:

```
enum Foo[A]:
    case I(x: Int) extends Foo[Int]
    case B(x: String) extends Foo[String]
    case G[A, B](x: B, y: B) extends Foo[A]
```

```
data Foo a where
    I :: Int -> Foo Int
    B :: String -> Foo String
    G :: b -> b -> Foo a
```

A.2 Block Syntax and By-Name Parameters

Scala is an expression-oriented language with very few true statements. In fact, a pair of braces `{}` can itself serve as an expression where the last line of the body is treated as the return value [Odersky et al. 2019, §6.11].

```
foo =
  let x = let y = 2 in y * y
  in x + x
```

```
def foo = {
  val x = { val y = 2; y * y }
  x + x
}
```

As a result, the above two snippets are equivalent to each other. Often, Scala 3 allows us to drop the braces and rely on indentation instead: this would not work for the inner-most set of braces, but there is a work-around for this we will see shortly. These braces are known as "blocks".

Blocks as arguments Sometimes, one may wish to pass a block into a function as an argument. In this case, the syntax may look a little clumsy:

```
square({
    val y = 2
    y + 3
})
```

This applies the `square` function to, effectively, `5`. However the presence of both braces and parentheses is

[1]In particular, one enum may not be a subtype of another, which prevents more flexible hierarchies.

annoying. Instead, Scala allows us to drop the parentheses when a block is provided as the argument [Odersky et al. 2019, §6.6]:

```
square {
    val y = 2
    y + 3
}
```

```
square:
    val y = 2
    y + 3
```

In the above examples, the parentheses have been removed. As an additional sugar, Scala 3 allows for the braces to be substituted in such instances with significant indentation and a colon. The braceless syntax is seldomly employed in the dissertation, however, the block-argument syntax is used liberally to aid readability. It is also possible to write `(x => { ... })` more concisely as `{ x => ... }`.

By-name parameters Closely linked with the block syntax is the by-name parameter [Odersky et al. 2019, §4.6.2], which is one form of laziness within Scala. An argument can be declared by-name by writing `=>A` instead of `A`. Technically, this is describing a function with no arguments or application parentheses that returns `A`: naturally, every time it is used it will be re-evaluated. This is the only way to make an argument to a function itself lazy (`lazy val` can only be used within the body of the function) and is employed through-out `parsley` to facilitate the lazy AST. Often, block application and by-name parameters are used together to create "new syntax", for instance a `when` expression can be built and used as follows:

```
def when(cond: Boolean)(body: =>Unit): Unit =
    if cond then body else ()
```

```
when (x < 7) {
    println("x is less than 7!")
}
```

Here, a by-name parameter of type `Unit` is taken in a curried position, meaning that the application takes two sets of arguments. It will not be evaluated until it is used within the function, namely only if the given condition is `true`. The example on the right shows how this may be used, with the second set of parentheses dropped in favour of the block-argument syntax: this allows an effective means of creating native looking "syntactic" constructions.

A.3 Functions

Appendix A.2 showed an example of currying in the definition of the `when` function. In Scala, functions are naturally uncurried, and a two-argument function would have type `(A, B) => C` and not `A => B => C`. Unlike Haskell, however, `(A, B) => C` is distinct from a function that accepts a pair: `((A, B)) => C`, which requires an extra set of parentheses. Strictly speaking, `(A, B) => C` is syntactic sugar for `Function2[A, B, C]`:

```
trait Function2[-A, -B, +C] {
    def apply(x: A, y: B): C
    def curried: A => B => C  = (x: A) => (y: B) => this.apply(x, y)
    def tupled: ((A, B)) => C = (xy: (A, B)) => this.apply(x._1, x._2)
}
```

A function in Scala is an object of one of the `FunctionN` traits, which has the `apply` method; in practice, `f.apply(x, y)` sugars into `f(x, y)`. A function can be curried or tupled by the corresponding methods, which adapts it to either be a function returning a function, or indeed a function accepting a tuple. Here, the `-` and `+` are variance annotations that describe the covariance and contravariance of the type parameters. Several `uncurried` functions also exist in the `Function` companion object to go in the other direction.

A.3.1 Composition and Infix Method Style

The `Function1[A, B]` trait, representing `A => B`, has some extra methods compared with `Function2[A, B, C]`:

```
trait Function1[-A, +B] {
    def apply(x: A): B
    def compose[T](g: T => A): T => B = x => this.apply(g(x))
    def andThen[C](g: B => C): A => C = x => g(this.apply(x))
}
```

The `compose` and `andThen` methods are analogous to Haskell's `(.)` and `flip (.)` respectively. In Scala, method application can be turned into an infix syntax [Odersky et al. 2019, §6.12.3], so `f andThen g` is more commonly seen than `f.andThen(g)`: this is usually done on a case-by-case basis, with some methods almost always written infix style and some never written like that.

A.3.2 Making Functions out of Methods

Methods are not objects, but they can be converted into functions, which are objects. The compiler can perform this conversion automatically, or it can be done explicitly via eta-expansion: a two-argument function `f` can be eta-expanded to `(x, y) => f(x, y)`. This can be written in a shorter form with *placeholder syntax* [Odersky et al. 2019, §6.23.2]: `f(_, _)`. Placeholder syntax can be used for partial application as well:

```
def power(n: Int, x: Int): Int = ...
val square: Int => Int = power(2, _)
```

In the above example, the power function has been partially applied to `2` to make a residual `square` function.

A.3.3 Partial Functions

In addition to `Function`s, Scala also has a `PartialFunction` trait, which is a subtype of single argument functions. In addition to the `apply` method, it has an `isDefinedAt` method that denotes whether a given input is defined for the function. A `PartialFunction[A, B]` can be converted to an `A => Option[B]` function and vice-versa. Their benefit is in having their own literal syntax [Odersky et al. 2019, §8.5]:

```
// on List[A]:
//   def collect[B](f: PartialFunction[A, B]): List[B]
xs.collect {
    case x if isEven(x) => x / 2
}
```

The `collect` method can be thought of as an analogue to Haskell's `mapMaybe`, which simultaneously maps and filters a collection. In Scala, this is done with a partial function: if the input is defined for the function `f`, it passes through the filter and is mapped by applying the function `f`, otherwise it is dropped. This can be conveniently written as something that resembles a floating `case` expression: the example above will filter the list to remove all odd values, and map all of them to be themselves divided by 2. Some combinators within `parsley`, like `collect`, do accept partial functions, and partial function literals can serve as a convenient way to do pattern matching (almost like Haskell's `LambdaCase` extension).

A.4 Implicits

The implicit system [Odersky et al. 2019, Chapter 7] is one of Scala's killer features. Scala does not have first-class support for typeclasses, for instance, but the implicits can be used to implement them with more flexiblity than Haskell offers – though with the absence of coherency, which sometimes is an advantage!

Scala 3 reworked the implicits to provide a bit more syntactic support and improve how they work. There are broadly three classes of implicit: implicit values, which form the basis behind typeclasses; implicit conversions, via the `Conversion` typeclass; and extension methods.

A.4.1 Implicit Values and Typeclasses

As mentioned above, Scala has no support for typeclasses as a first-class construct in the language. Instead, they can be implemented using implicit values and arguments. In Scala 3 these two concepts have been each given their own keyword to distinguish them: `given` makes a value implicit, and `using` makes an argument implicit.

One example of this is the `ExecutionContext`, which is used to control threading when using Scala's `Future`s. Constructing a `Future`, or using its methods, requires access to an `ExecutionContext`, which will describe what threads are available and how the work distribution should be performed. This would be very noisy if the programmer had to specify this everywhere, so this is all passed implicitly – we will see an example of this soon. Scala defines a *canonical* `ExecutionContext` called `global`, which is found in the `ExecutionContext` companion object: this will always be available and in scope as it is present in the companion object – think an instance defined alongside a typeclass in Haskell:

```scala
trait ExecutionContext {
    def execute(runnable: Runnable): Unit
    def reportFailure(cause: Throwable): Unit
}
object ExecutionContext {
    given global: ExecutionContext = new ExecutionContext {
        def execute(runnable: Runnable) = ...
        def reportFailure(cause: Throwable) = ...
    }
}
```

Here, the `given` keyword tells us that `global` is available implicitly, though it also behaves like `val` so that `global` is also a regular binding within the object. When the right-hand side of this assignment is a *single abstract method* (SAM) type, it can be implemented as a lambda or function – this is like the Java *functional interfaces* sugar. To align better with typeclasses, however, there is an alternative syntax for a `given` object construction:

```scala
object ExecutionContext {
    given global: ExecutionContext with
```

```
        def execute(runnable: Runnable) = ...
        def reportFailure(cause: Throwable) = ...
}
```

Here, there is no assignment or `new` keyword and instead the implementation of this `ExecutionContext` comes after the `with` keyword. This looks more akin to Haskell's `instance ExecutionContext where` syntax. If the name is not required, it can be dropped by writing `given ExecutionContext = ...` or `given ExecutionContext with ....`

On the other side of the coin, methods of `Future` require a specific `ExecutionContext` to be in scope (if one is not specified, `global` will be used as it is the canonical value):

```
trait Future[+A] extends Awaitable[A] {
    ...
    def map[B](f: A => B)(using executor: ExecutionContext): Future[B] = ...
    ...
}
```

Here, the `map` method of `Future` uses `using` to specify it requires an `ExecutionContext` to be in implicit scope (or otherwise passed explicitly into the argument):

```
val fut1: Future[Int] = ...
val fut2: Future[Boolean] = fut1.map(_ > 0)
val fut3: Future[Boolean] = fut1.map(_ > 0)(using ExecutionContext.global)
```

The two values `fut2` and `fut3` are constructed in the same way: the compiler will elaborate the definition of `fut2` to the same expression as the definition of `fut3`. Providing another implicit value would change the elaboration.

In the definition of `map`, the name could be dropped if it is not needed by writing `using ExecutionContext`. By specifying a `given ExecutionContext` in a more specific scope, the chosen value passed to `Future`'s methods can vary – this is unlike Haskell, which requires global uniqueness.

Typeclasses In some sense, `ExecutionContext` can already be thought of like a Haskell typeclass, with the `given global: ExecutionContext` being a named typeclass instance (in Scala, ambiguities can be resolved by specifying the name of the instance). As an example of a parametric typeclass though, consider how `Monoid` can be formulated in Scala:

```
trait Monoid[M] {
    def empty: M
    def combine(x: M, y: M): M
    def concat(xs: List[M]): M = xs.foldLeft(empty)(combine)
}
object Monoid {
    given [A] Monoid[List[A]] with
        def empty = Nil
        def combine(xs: List[A], ys: List[A]) = xs ++ ys

    given additiveInt: Monoid[Int] with
        def empty = 0
        def combine(x: Int, y: Int) = x + y
    given [A, B](using monB: Monoid[B]): Monoid[A => B]
        def empty = _ => monB.empty
        def combine(f: A => B, g: A => B) = x => monB.combine(f(x), g(x))
}
```

Here we can see the trait `Monoid[M]`, which represents the Haskell class `Monoid m`, and three provided instances. The first defines a monoid for any `List[A]`, implementing the methods how you would expect; the second defines a named monoid for `Int` which is additive – a multiplicative monoid could also be defined, and disambiguated by name; and the third defines an instance for `A => B`, so long as there is an implicit instance for `Monoid[B]` available. The third instance also demonstrates how the instance can be used by invoking the methods on the implicitly provided instance, which is just an object. There are more convenient ways of calling the methods without naming the instance, like using `summon[Monoid[B]].empty`, which is an identity function with an implicit parameter, or defining an `apply` on `Monoid` that behaves similarly to `summon`, allowing `Monoid[B].empty`.

Context Bounds The `using` syntax can be a bit cumbersome however, so there is a sugar to allow for the typeclass constraint to be specified a way that is more similar to Haskell's constraints [Odersky et al. 2019, §7.4]. This works for typeclasses with a single type parameter, and takes the form `A: C` instead of `using C[A]`:

```
object Monoid {
    // for convenience
    def apply[A: Monoid]: Monoid[A] = summon[Monoid[A]]

    ...

    given [A, B: Monoid]: Monoid[A => B]
        def empty = _ => Monoid[B].empty
        def combine(f: A => B, g: A => B) = x => Monoid[B].combine(f(x), g(x))
}
```

Using this syntax, the instance for functions that return monoids can be simplified into a more concise form. Both the instance and the `apply` method use the context bound syntax, which desugars back into a `using` clause.

A.4.2 Implicit Conversions

Implicit conversions from `A` to `B` are simply an instance of the `Conversion[A, B]` typeclass, which extends `A => B`. What this means is that the version can be applied implicitly as well as explicitly:

```
trait Conversion[-A, +B] extends (A => B) {
    def apply(x: A): B
}

given charLift: Conversion[Char, Parsley[Char]] with
  def apply(c: Char) = char(c)

val p: Parsley[Char] = charLift('a') // sugared charLift.apply('a')
val q: Parsley[Char] = 'a'
```

In the above example, the charLift implicit conversion has been applied explicitly to construct the parser p, and is elaborated implicitly by the compiler into the same shape as p's definition in q's case. The process for implicit search will occur whenever a type occurs, looking for an in-scope implicit function to resolve the type-error. As one might imagine, this is considered to be a very abusable feature, and usually requires a compiler flag or feature import to use without warnings. They are useful for DSLs, however, and charLift helps reduce noise in parsley parsers. As Conversion is a SAM, the instance can be implemented concisely as the char(_) function:

```
given charLift: Conversion[Char, Parsley[Char]] = char(_)
```

A.4.3 Extension Methods

Extension methods are a way of adding additional methods onto a class, which are implemented via static dispatch of a function instead. They take the form extension typeparams receiver { methods } where the braces can be optionally dropped in favour of indentation. As an example, the zipped notation that parsley allows for applying an arbitrary arity map can be defined as follows:

```
extension [A, B] (pq: (Parsley[A], Parsley[B]))
    def zipped[C](f: (A, B) => C): Parsley[C] = lift2(f, pq._1, pq._2)
```

Here, zipped is an extension method defined on a pair of parsers, and takes a function as an argument. The body of the combinator just calls lift2 with the function and then first and second projections of the pair, respectively. With this in scope:

```
val decl = (ident, '=' ~> expr).zipped(Decl(_, _))
// is equivalent to
val decl = zipped((ident, '=' ~> expr))(Decl(_, _))
```

The method is desugared into a curried function call with static dispatch. When a method is called on a type and that method does not exist, Scala will look for any extension methods in scope that can be substituted in instead. Extension methods can be defined generically by typeclass instances, so that the Applicative typeclass in Scala might look like:

```
trait Functor[F[_]] {
    extension [A] (u: F[A]) def map[B](f: A => B): F[B]
}

trait Applicative[F[_]] extends Functor[F] {
    def pure[A](x: A): F[A]
    extension [A, B] (u: F[A => B]) def <*>(v: F[A]): F[B]
    extension [A] (u: F[A])
        override def map[B](f: A => B): F[B] = pure(f) <*> u
}
```

Here, the Functor trait requires instances to define a map extension method on F[A], and the Applicative typeclass requires an extension method on F[A => B] specifically. In this case, the Applicative shows how this might be implemented, by providing an overridden map extension method in terms of <*> and pure. As an aside, in practice, this actually means that a downstream implementer of Applicative only needs to specify pure and

`<*>` and gets `Functor` for free too, unlike in Haskell, which requires them to implement both explicitly – this is not without its downsides, however, but that is out of scope for this discussion.

Thanks to `@SystemFW` and Guillaume Martres (`@smarter3`) from the Typelevel/Scala Discord as well as David Davies for checking this over!

Appendix B

Full Implementation of `LetRec`

The presentation of the `letRec` function in §3.2.4 was simplified for discussion's sake. There are two problems with it: the first is that the type alias `CodeT` cannot be partially applied and used as an argument to `DMap`, and second that type aliases cannot be provided for the parameters `f1` or `f2`. These issues are both due to the requirement that type synonyms (and type families) need to be fully saturated [Kiss, Field, et al. 2019]. In fact, Kiss, Field, et al. [2019] introduce an extension `UnsaturatedTypeFamilies`, which would allow for this to be side-stepped and the original presented code would work, but this has sadly not yet been merged into GHC as of writing.

Instead, `CodeT` must be rendered as a newtype, and these need to be wrapped and unwrapped at the use-sites:

```haskell
{-# LANGUAGE ScopedTypeVariables #-}
import Data.Dependent.Map as DMap ((!), foldrWithKey, map, traverseWithKey, DMap)
import Data.GADT.Compare   (GCompare)
import Data.Functor.Const  (Const(Const))
import Language.Haskell.TH (unTypeCode, unsafeCodeCoerce, Q, Code, Quote(newName), Name,
                            Exp(LetE, VarE), Clause(Clause), Dec(FunD), Body(NormalB))

newtype CodeT f x = CodeT { unCodeT :: Code Q (f x) }

letRec :: forall k f1 f2 r. GCompare k => DMap k f1 -> (forall x. k x -> String)
       -> (forall x. k x -> f1 x -> DMap k (CodeT f2) -> CodeT f2 x)
       -> (DMap k (CodeT f2) -> Code Q r) ->  Code Q r
letRec bindings nameOf genBinding expr = unsafeCodeCoerce $
  do  names <- DMap.traverseWithKey (\key _ -> makeName key) bindings
      let typedNames = DMap.map makeTypedName names
      let makeDecl :: forall x. k x -> f1 x -> Q Dec
          makeDecl key body =
           do let Const name = names DMap.! key
              binding <- unTypeCode (unCodeT (genBinding key body typedNames))
              let decl = FunD name [Clause [] (NormalB binding) []]
              return decl
      decls <- forallKeyVals makeDecl bindings
      exp   <- unTypeCode (expr typedNames)
      return (LetE decls exp)
  where makeName :: k x -> Q (Const Name x)
        makeName key = fmap Const (newName (nameOf key))
        makeTypedName :: Const Name x -> CodeT f2 x
        makeTypedName (Const name) = CodeT (unsafeCodeCoerce (return (VarE name)))

forallKeyVals :: Applicative m => (forall x. k x -> f x -> m b)-> DMap k f -> m [b]
forallKeyVals f = DMap.foldrWithKey (\key val -> liftA2 (:) (f key val)) (pure [])
```

The above code does fully type-check and will work as intended. However, the fix has not addressed the caller-side issue with the types `f1` or `f2`. Simply, neither of these type constructors can be type-aliases or families, since they would be unsaturated; instead, they must too be rendered as newtypes. This is probably fine for `f1`, since it is a static type, but `f2` is found underneath `Code Q`. This means that it is also necessary to perform the newtype wrapping and unwrapping in the generated code – this will be eliminated by GHC, as all newtypes are, but it is still intrusive in the rest of the staged interpreter.

Alternatively, this combinator can simply be specialised to the concrete types `k`, `f1` and `f2` and will work just fine without newtypes. This is the approach taken in the real `parsley` implementation.

Appendix C

Reference Lifting Graph Algorithms

For simplicity, this appendix shows a cubic algorithm for the three algorithms omitted from §4.4.3 improved with an ordered worklist as described below – in practice, the more sophisticated algorithm should be implemented.

Worklist algorithms Iterative algorithms that compute a least-fixed point of a set of equations can benefit from a *worklist* [Appel 2004]. This means that, instead of recomputing every equation every iteration, only compute those for which the result may have changed. This is accomplished by tracking the dependents as well as dependencies and adding all dependents to a queue whenever a result changes. The algorithm stops when the worklist is empty.

Topological ordering Iterative algorithms can also be improved by recomputing the equations such that dependencies are computed before their dependents. For example, many data-flow algorithms for classical compilers traverse a source program backwards [Appel 2004]. For algorithms based on graphs, proceeding in a reverse topological ordering [Tarjan 1976; Cormen et al. 2009] – i.e. deepest first in a depth-first traversal – is usually effective.

Ordered worklist Both the worklist and topological ordering optimisations can be combined by ensuring that the worklist is a priority queue that prioritises elements that are topologically later: this makes it more likely that all the dependencies of a work item have been recomputed before it is processed, avoiding needless reinsertion into the worklist.

While these strategies do not improve the asymptotic complexity of an iterative algorithm (suppose each work item has dependencies on all others, this degenerates to the pathological case in either algorithm), they do have a benefit in practice, especially for more sparse graphs.

C.1 Constructing Graphs

The analyseNTs function returns a map from Word64 (the underlying type within NT) to a set of other Word64s. However, this is not the most efficient structure to perform queries over, given that the domain of the map is contiguous. Instead, it is more efficient to represent the call graph as an Array Word64 [Word64], which is an adjacency list representation [Cormen et al. 2009] and provides $O(1)$ query time. This can be done easily for the call graph:

```
type Graph = Array Word64 [Word64]
buildCallGraph :: Map Word64 (Set Word64) → Graph
buildCallGraph calls = Array.listArray (0, fst (Map.findMax calls)) (map Set.toList (Map.elems calls))
```

This function constructs an Array given a range of values and list of index-element pairs. The range of values is from 0 to the largest non-terminal found in the calls map, and the elements of the array are sourced from the calls map's elements.

In addition to this, an inverse graph is also required, which denotes the callers of any given non-terminal (excluding the top-level). This can be done with the following iterative algorithm:

```
invert :: Graph → Graph
invert g = runSTArray $ do g′ ← newArray (Array.bounds g) [ ]
                           forM_ (Array.indices g) $ λn →
                             forM_ (g ! n) $ λsucc → do
                               preds ← readArray g′ succ
                               writeArray g′ succ (n : preds)
```

This function uses the ST monad to construct a mutable array g′ with the same bounds as the graph g, initialised with empty lists, and then iterates over g and adds each node n into the predecessors of its successors, which is given by g′. This is a simplistic $O(n^2)$ algorithm, which does not dominate the complexity of the overall analysis.

C.2 Topological Ordering

As noted above, a topological ordering can improve the performance (though not the complexity) of iterative algorithms by ensuring that results are computed that are more likely to not require re-computation first. This means that a topological ordering needs to be established for the call graph. This is achieved by associating each node with a DFNum, which is just a number, that indicates at which step it was encountered in a depth-first traversal of the graph. This is done by the topoOrdering function:

```
type DFNum = Word64
topoOrdering :: Set Word64 → Graph → Array Word64 DFNum
topoOrdering roots g = runSTArray $ do
  dfnums ← newArray (Array.bounds g) 0
  nextNum ← newSTRef 1
  let dfs v = whenS (fmap (≡ 0) (readArray dfnums v)) $ do dfnum ← readSTRef nextNum
                                                           writeSTRef nextNum (dfnum + 1)
                                                           writeArray dfnums v dfnum
                                                           forM_ (g ! v) dfs
  forM_ roots dfs
  return dfnums
```

The function first initialises an array of the same bounds as the graph g, and fills it with 0, which indicates "not visited". Then it initialises a counter called nextNum used to track the DFNum during the traversal. Finally, it traverses all the roots in turn using dfs: this is a simple depth-first traversal that only traverses if the node in question has not yet been visited. When a node is visited, however, it is assigned the next DFNum from nextNum, and this is written into the dfnums array. The topoOrdering function runs in $O(n)$.

C.3 Iterative Propagation

With all the required components defined, it is now possible to give the algorithm for propagating the free references transitively through the non-terminals, propagate.

```
propagate :: Array Word64 DFNum
          → Map Word64 (Set (Some SigmaVar))
          → Map Word64 (Set (Some SigmaVar))
          → Graph → Graph
          → Map Word64 (Set (Some SigmaVar))
propagate dfnums uses defs callees callers = (Map.fromList ∘ Array.assocs ∘ runSTArray) $ do
  let vs = Array.indices dfnums
  freeRefs ← newArrayList (Array.bounds dfnums) (map (λv → (uses ! v) \\ (defs ! v)) vs)
  let worklist = addWork Map.empty vs
  maybe (return ()) (unfoldM_ (uncurry (prop freeRefs))) (Map.maxView worklist)
  return freeRefs
  where addWork = foldl' (flip (λv → Map.insert (dfnums ! v) v))
        prop freeRefs v work = do
          frees       ← readArray freeRefs v
          freesCallees ← fmap Set.unions (traverse (readArray freeRefs) (callees ! v))
          let frees′   = frees ‘union‘ (freesCallees \\ (defs ! v))
          if size frees ≢ size frees′ then do
            writeArray freeRefs v frees′
            return (Map.maxView (addWork work (callers ! v)))
          else return (Map.maxView work)
```

The propagate function first allocates an array mapping non-terminals to their set of immediate free references, computed by taking the difference between the local uses and local definitions. Then it constructs a worklist with all the non-terminals (which will return the non-terminal with the largest DFNum with maxView) and iteratively performs prop until there is no more work to be done: this uses the function unfoldM_ with the following signature:

```
unfoldM_ :: Monad m ⇒ (s → m (Maybe s)) → s → m ()
```

The function unfoldM_ f s repeatedly applies the function f to the seed s, performing an effect in the process, so long as f s results in a Just value. In this case, it captures the essence of repeatedly performing iterations while there is still work available on the worklist.

The function prop performs a single step of the algorithm: first collect the current free references for the non-terminal v, frees, and compute the union of the free references of each of the callees of v too. Let frees′ be the union of the original free references and those required by the callees that are not defined by v; if frees and frees′ differ in size, then more free references were added this iteration so update this set in the array and add all the callers of the non-terminal v to the worklist (since they may need updating). If the size matches then no update is required, and no more items are added to the worklist.

Once the worklist is exhausted, the freeRefs array is converted into a Map, which forms the result of the analysis. The overall runtime of the algorithm is $O(n^3)$, assuming that set union can be done in $O(n)$. In practice, it is likely to perform faster except for parsers with many nested mutually recursive parsers.

Appendix D

Code after Chapters 3 and 4

Section 3.1 introduced the basis for the `compile` function as well as the `Combinator` tree. Section 3.2 introduced the evaluation semantics, which were subsequently staged: this made some minor modifications to the `Combinator` tree and then additional changes to `compile` to support join-points. Section 3.3 then further modified the `compile` function to break away from requiring the full set of bindings at all times.

Section 4.3 then introduces more constructors to the `Combinator` and `Instr` types, and additional rules to the `compile` function. The staged evaluation is also altered to support the threading of references in §4.4.

To help make the development on top of these two chapters, this appendix summarises the final forms of both the `Combinator` and `Instr` types and the `compile` and `eval` functions. This can be used as a base reference for the changes that occur during CHAPTER 6: ANALYSIS AND OPTIMISATION.

D.1 Syntactic Functors

The `Parsley` type is a fix-point of the `Combinator` syntactic functor, which now contains capacities for references and incorporates `Code` instead of raw values:

```
type Parser a = Fix (Combinator :+: ScopeRef) a
type Parsley a = Fix Combinator a

data Combinator k a where
  Pure      :: Code a -> Combinator k a
  Empty     :: Combinator k a
  Satisfy   :: Code (Char -> Bool) -> Combinator k Char
  (:<*>:)   :: k (a -> b) -> k a -> Combinator k b
  (:*>:)    :: k a -> k b -> Combinator k b
  (:<*:)    :: k a -> k b -> Combinator k a
  (:<|>:) :: k a -> k a -> Combinator k a
  Branch    :: k (Either x y) -> k (x -> a) -> k (y -> a) -> Combinator k a
  Atomic    :: k a -> Combinator k a
  Look      :: k a -> Combinator k a
  NegLook   :: k a -> Combinator k ()
  Let       :: NT a -> Combinator k a
  MakeRef   :: SigmaVar a -> k a -> k b -> Combinator k b
  ReadRef   :: SigmaVar a -> Combinator k a
  WriteRef :: SigmaVar a -> k a -> Combinator k ()

data ScopeRef k a where
  ScopeRef :: k a -> (forall r. Ref r a -> k b) -> ScopeRef k b
```

The `Machine` type is the fix-point of the `Instr` syntactic functor, it specifically demands one handler, an empty stack and the return goal much match the final result of the parser. The `Instr` functor is now augmented with the machinery for join-points and references. Code is also explicitly used within the functor:

```haskell
type Machine a = Fix Instr '[] 'One a
data Instr k xs n a where
  Sat    :: Code (Char -> Bool) -> k (Char ': xs) ('Succ n) a -> Instr k xs ('Succ n) a
  Push   :: Code x -> k (x : xs) n a -> Instr k xs n a
  Pop    :: k xs n a -> Instr k (x ': xs) n a
  Swap   :: k (x ': y ': xs) n a -> Instr k (y ': x ': xs) n a
  Red    :: Code (x -> y -> z) -> k (z ': xs) n a -> Instr k (y ': x ': xs) n a
  Case   :: k (x ': xs) n a -> k (y ': xs) n a -> Instr k (Either x y ': xs) n a
  Tell   :: k (String ': xs) n a -> Instr k xs n a
  Seek   :: k xs n a -> Instr k (String ': xs) n a
  Call   :: NT x -> k (x ': xs) ('Succ n) a -> Instr k xs ('Succ n) a
  Ret    :: Instr k '[a] n a
  Catch  :: k xs ('Succ n) a -> k (String ': xs) n a -> Instr k xs n a
  Commit :: k xs n a -> Instr k xs ('Succ n) a
  Raise  :: Instr k xs ('Succ n) a
  MkJoin :: Phi x -> k (x ': xs) n a -> k xs n a -> Instr k xs n a
  Join   :: Phi x -> Instr k (x ': xs) n a
  Make   :: SigmaVar x -> k x s n a -> Instr k (x ': xs) n a
  Read   :: SigmaVar x -> k (x ': xs) n a -> Instr k xs n a
  Write  :: SigmaVar x -> k xs n a -> Instr k (x ': xs) n a
```

D.2 Compilation Function

Since its original formulation at the end of §3.1, the `compile` function has had `Code` incorporated into it; the introduction of join-points in §3.2.4 forces the algebra to be embedded into the `State` monad so that `(<=<)`, `liftM`, `return`, and `liftM2` must be used throughout to make the types align properly; and it supports references.

```haskell
type CodeGen x =
  forall xs n a. Fix Instr (x ': xs) ('Succ n) a -> State Word64 (Fix Instr xs ('Succ n) a)
compile :: Parsley a -> Machine a
compile p = evalState (cata comp p ret) 0 where
  comp :: Combinator CodeGen x -> CodeGen x
  comp (Pure x)         = return . push x
  comp (pf :<*>: px)    = pf <=< px . red [||($)||]
  comp (p :*>: q)       = p . pop <=< q
  comp (p :<*: q)       = p <=< q . pop
  comp (Satisfy f)      = return . sat f
  comp Empty            = \k -> return raise
  comp (p :<|>: q)      = \k -> do phi <- freshPhi @x
                                   liftM2 (mkJoin phi k . catch)
                                          (p (commit (join phi)))
                                          (liftM (tell . same) (q (join phi)))
    where same k        = red  [|| \new old -> if new == old then Right () else Left () ||]
                               (kase raise (pop . k))
  comp (Atomic p)       = \k -> liftM2 catch (p (commit k)) (return (seek raise))
  comp (Branch b l r) = \k -> do phi <- freshPhi @x
                                   let k' = red [||flip ($)||] (join phi)
                                   liftM (mkJoin phi k) (b =<< liftM2 kase (l k') (r k'))
  comp (Look p)         = liftM tell . p . swap . seek
  comp (NegLook p)    = \k -> liftM2 catch
                                   (liftM tell (p (pop (seek (commit raise)))))
                                   (return (seek (push [||()||] k)))
  comp (Let nt)       = return . call nt
  comp (NewRef s p b) = p . make s <=< b
  comp (ReadRef s)    = return . read s
  comp (WriteRef s p) = p . write s . push [||()||]
```

D.3 Staged Evaluation

Evaluation carrier and state The Eval type, along with Gamma and Ctx are as follows:

```
data Gamma r xs n a = Gamma { input    :: Code String
                            , moore    :: QList xs
                            , handlers :: Vec n (Code (String -> r))
                            , retCont  :: Code (a -> String -> r)
                            }

type BaseFunc r s a = String -> (String -> ST s r) -> (a -> String -> ST s r) -> ST s r
type family Func xs r s a where
  Func '[]       r s a = BaseFunc r s a
  Func (x ': xs) r s a = STRef s x -> Func xs r s a

data NTBound r s a  = forall rs. NTBound (Refs rs) (Code (Func rs r s a))
type PhiBound r s a = Code (String -> (a -> String -> ST s r) -> ST s r)
type QSTRef s a     = Code (STRef s a)

data Ctx r s = Ctx { nts  :: DMap NT (NTBound r s)
                   , phis :: DMap Phi (PhiBound r s)
                   , regs :: DMap SigmaVar (QSTRef s)
                   }

type Eval r s xs n a = Ctx r s -> Gamma r s xs n a -> Code (ST s r)
```

Top-level `eval` The eval function is modified to support multiple bindings, and staging the code. In addition, the free references of each binding have been threaded through, and ST is used throughout.

```
eval :: Machine a -> DMap NT Machine -> Map Word64 (Some Refs)
     -> Code String -> Code (Maybe a)
eval m ms freeregs inp =
  letRec ms show (\(NT w) m nts -> makeNTBound (freeregs ! w) $ \regs -> [|| \inp h ret ->
                  $$(evalMachine m nts regs [||inp||] [||h||] [||ret||])||])
               (\nts -> [|| runST $$(evalMachine m nts DMap.empty inp fail accept) ||])
  where fail   = [|| \_ -> return Nothing ||]
        accept = [|| \x _ -> return (Just x) ||]

        evalMachine :: Machine x -> DMap NT (NTBound r s) -> DMap SigmaVar (QSTRef s)
                    -> Code String -> Code (String -> ST s r)
                    -> Code (x -> String -> ST s r) -> Code (ST s r)
        evalMachine m mts regs inp h ret =
          cata alg m (Ctx { nts, phis = DMap.empty, regs = regs })
                     (Gamma { input = inp, moore = QNil,
                              handlers = Cons h Nil, retCont = ret })

        alg :: Instr (Eval r) xs n a -> Eval r xs n a
        alg Ret ctx gamma = evalRet ctx gamma
        alg ...
```

Evaluation functions The evaluation functions for each instruction account for all of the additional parts of the system that may not have been made explicit in §3.2 and CHAPTER 4: each instruction has a type parameter `s` for the underlying `ST s`, and they all take a `ctx` argument.

```
evalPush :: Code x -> Eval r s (x ': xs) n a -> Eval r s xs n a
evalPush x k ctx gamma{..} = k ctx (gamma { moore = QCons x moore })

evalPop  :: Eval r s xs n a -> Eval r s (x ': xs) n a
evalPop k ctx gamma{..} = let QCons x xs = moore in k ctx (gamma { moore = xs })

evalSwap :: Eval r s (x ': y ': xs) n a -> Eval r s (y ': x ': xs) n a
evalSwap k ctx gamma{..} = let QCons x (QCons y xs) = moore
                           in k ctx (gamma { moore = QCons y (QCons x xs) })

evalRed :: Code (x -> y -> z) -> Eval r s (z ': xs) n a -> Eval r s (y ': x ': xs) n a
evalRed f k ctx gamma{..} = let QCons y (QCons x xs) = moore
                            in k ctx (gamma { moore = QCons [||$$f $$x $$y||] xs })

evalCase :: Eval r s (x ': xs) n a -> Eval r s (y ': xs) n a
         -> Eval r s (Either x y ': xs) n a
evalCase left right ctx gamma{..} = let QCons exy xs = moore in
  [|| case $$exy of
        Left x  -> $$(left ctx (gamma { moore = QCons [||x||] xs}))
        Right y -> $$(right ctx (gamma { moore = QCons [||y||] xs}))
  ||]

evalRaise :: Eval r s xs ('Succ n) a
evalRaise ctx gamma{..} = let Cons h _ = handlers in [|| $$h $$input ||]

evalCommit :: Eval r s xs n a -> Eval r s xs ('Succ n) a
evalCommit k ctx gamma{..} = let Cons _ hs = handlers in k ctx (gamma { handlers = hs })

evalCatch :: Eval r s xs ('Succ n) a -> Eval r s (String ': xs) n a -> Eval r s xs n a
evalCatch k h ctx gamma{..} =
  [|| let handler input' =
            $$(h ctx (gamma { input = [||input'||], moore = QCons input moore}))
      in $$(k ctx (gamma { handlers = [||handler||] : handlers }))
  ||]

evalSat :: Code (Char -> Bool) -> Eval r s (Char ': xs) ('Succ n) a -> Eval r s xs ('Succ n) a
evalSat f k ctx gamma{..} = [|| case $$input of
    c : cs | $$f c -> $$(k (gamma { input = [||cs||], moore = QCons [||c||] moore }))
  ||]

evalTell :: Eval r s (String ': xs) n a -> Eval r s xs n a
evalTell k ctx gamma{..} = k ctx (gamma { moore = QCons input xs })

evalSeek :: Eval r s xs n a -> Eval r s (String ': xs) n a
evalSeek k ctx gamma{..} = let QCons input' xs = moore
                           in k ctx (gamma { input = input', moore = xs })

evalRet :: Eval r s '[a] n a
evalRet ctx gamma{..} = let QCons x QNil = moore in [|| $$retCont $$x $$input ||]

evalCall :: NT x -> Eval r s (x ': xs) ('Succ n) a -> Eval r s xs ('Succ n) a
evalCall nt k ctx{..} gamma{..} =
  [|| $$(applyNTBound (nts DMap.! nt) regs) $$input $$h $ $$ret ||]
  where Cons h _ = handlers
        ret      = [|| \x input' -> $$(k ctx (gamma { input = [||input'||]
                                                     , moore = QCons [||x||] moore })) ||]

evalJoin :: Phi x -> Eval r s (x ': xs) n a
evalJoin phi ctx{..} gamma{..} = let QCons x _ = moore
                                 in [|| $$(phis DMap.! phi) $$input $$x ||]

evalMkJoin :: Phi x -> Eval r s (x ': xs) n a -> Eval r xs n a -> Eval r xs n a
evalMkJoin phi body k ctx{..} gamma{..} =
  [|| let join x input' = $$(body ctx (gamma { input = [||input'||]
                                             , moore = QCons [||x||] moore }))
      in $$(k (ctx { phis = DMap.insert phi [||join||] phis}) gamma) ||]

evalRead :: SigmaVar x -> Eval r s (x ': xs) n a -> Eval r s xs n a
evalRead s k ctx{..} phi{..} =
  [|| do x <- readSTRef $$(regs DMap.! s)
         $$(k ctx (gamma { moore = QCons [||x||] moore})) ||]

evalWrite :: SigmaVar x -> Eval r s xs n a -> Eval r s (x ': xs) n a
evalWrite s k ctx{..} phi{..} =
  let QCons x xs = moore
  in [|| do writeSTRef $$(regs DMap.! s) $$x
            $$(k ctx (gamma { moore = xs})) ||]

evalMake :: SigmaVar x -> Eval r s xs n a -> Eval r s (x ': xs) n a
evalMake s k ctx{..} phi{..} =
  let QCons x xs = moore
  in [|| do reg <- newSTRef $$x
            $$(k (ctx {regs = DMap.insert s [||reg||] regs}) (gamma { moore = xs})) ||]
```

Appendix E

Full Code for Templating Bridges

```
posAp :: Bool -> Q Exp -> Q Exp
posAp True  p = [e| pos <**> $p |]
posAp False p = p

deriveLiftedConstructors :: String -> [Name] -> Q [Dec]
deriveLiftedConstructors prefix = fmap concat . traverse deriveCon
  where
    deriveCon :: Name -> Q [Dec]
    deriveCon con = do
      (con', ty, func, posFound, n) <- extractMeta True prefix (funcType . map parserOf) con
      args <- replicateM n (newName "x")
      sequence [ sigD con' ty
               , funD con' [clause (map varP args)
                   (normalB (posAp posFound (applyArgs [e|pure $func|] args))) []]
               ]

    applyArgs :: Q Exp -> [Name] -> Q Exp
    applyArgs = foldl' (\rest arg -> [e|$rest <*> $(varE arg)|])

deriveDeferredConstructors :: String -> [Name] -> Q [Dec]
deriveDeferredConstructors prefix = fmap concat . traverse deriveCon
  where
    deriveCon :: Name -> Q [Dec]
    deriveCon con = do
      (con', ty, func, posFound, _) <- extractMeta False prefix (parserOf . funcType) con
      sequence [ sigD con' ty
               , funD con' [clause [] (normalB (posAp posFound [e|pure $func|])) []]
               ]

funcType :: [Q Type] -> Q Type
funcType = foldr1 (\ty rest -> [t| $ty -> $rest |])

parserOf :: Q Type -> Q Type
parserOf ty = [t| Parser $ty |]

extractMeta :: Bool -> String -> ([Q Type] -> Q Type) -> Name
          -> Q (Name, Q Type, Q Exp, Bool, Int)
extractMeta posLast prefix buildType con = do
  DataConI _ ty _ <- reify con
  (forall, tys) <- splitFun ty
  posIdx <- findPosIdx con tys
  let tys' = maybeApply deleteAt posIdx tys
  let nargs = length tys' - 1
  let con' = mkName (prefix ++ pretty con)
  let func = buildLiftedLambda posLast con nargs posIdx
  return (con', forall (buildType (map return tys')), func, isJust posIdx, nargs)
```

```haskell
splitFun :: Type -> Q (Q Type -> Q Type, [Type])
splitFun (ForallT bndrs ctx ty) = do
  kindSigs <- isExtEnabled KindSignatures
  let bndrs' = if kindSigs then bndrs else map sanitiseStarT bndrs
  return (forallT bndrs' (pure ctx), splitFun' ty)
splitFun ty                     = return (id, splitFun' ty)

splitFun' :: Type -> [Type]
splitFun' (AppT (AppT ArrowT a) b)             = a : splitFun' b -- regular function type
splitFun' (AppT (AppT (AppT MulArrowT _) a) b) = a : splitFun' b -- linear function type
splitFun' ty                                   = [ty]

-- When KindSignatures is off, the default (a :: *) that TH generates is broken!
sanitiseStarT (KindedTV ty flag StarT) = PlainTV ty flag
sanitiseStarT ty = ty

findPosIdx :: Name -> [Type] -> Q (Maybe Int)
findPosIdx con tys = case elemIndices (ConT ''Pos) tys of
     []    -> return Nothing
     [idx] -> return (Just idx)
     _     -> fail $ unwords -- more than 1 index, which is ambiguous
        ["constructor", pretty con, "has multiple occurrences of Parsley.Patterns.Pos"]

buildLiftedLambda :: Bool -> Name -> Int -> Maybe Int -> Q Exp
buildLiftedLambda posLast con nargs posIdx = do
  args <- replicateM nargs (newName "x")
  posArg <- newName "pos"
  let pargs = if | isNothing posIdx -> map varP args
                 | posLast          -> map varP args ++ [varP posArg]
                 | otherwise        -> varP posArg : map varP args
  let eargs = maybeApply (flip insertAt (varE posArg)) posIdx (map varE args)
  lamE pargs (foldl' (\acc arg -> [e|$acc $arg|]) (conE con) eargs)

maybeApply :: (a -> b -> b) -> Maybe a -> b -> b
maybeApply = maybe id

deleteAt :: Int -> [a] -> [a]
deleteAt 0 (_:xs)  = xs
deleteAt n (x:xs)  = x : deleteAt (n-1) xs
deleteAt _ []      = []

elemIndices :: Eq a => a -> [a] -> [Int]
elemIndices y = go 0
  where  go _ [] = []
         go i (x:xs)
           | x == y     = i : go (i + 1) xs
           | otherwise  = go (i + 1) xs

pretty :: Name -> String
pretty = reverse . takeWhile (/= '.') . reverse . show
```

Appendix F

Extra Error System Code

F.1 Full Code for `WhitespaceOrNonPrintable`

```
def whitespaceOrNonPrintable(cs: Iterable[Char]): Option[String] =
    whitespaceOrNonPrintable(cs.take(2).mkString)
def whitespaceOrNonPrintable(s: String): Option[String] =
    whitespaceOrNonPrintable(s.codePointAt(0))
def whitespaceOrNonPrintable(cp: Int): Option[String] =
  // if the JVM considers this character whitespace (which include a few categories)
  // then give it a dedicated name
  if Character.isWhitespace(cp) then cp match
  case 0x000a => Some("newline")
  case 0x000d => Some("carriage return")
  case 0x0009 => Some("tab")
  case 0x0020 => Some("space")
  case _      => Some("whitespace character")
  // otherwise, check the Unicode category of the code-point
  else Character.getType(cp) match
  // these are all categories under "Control", and are all not printable
  case Character.FORMAT
     | Character.SURROGATE
     | Character.PRIVATE_USE
     | Character.UNASSIGNED
     | Character.CONTROL =>
       Character.toChars(cp) match
       // the code-point is made of a surrogate pair `h` and `l`, decompose it
       // and format as two 4-digit hex strings, plus its 6-digit hex code value
       case Array(h, l) =>
         Some(f"non-printable codepoint (\\u${h.toInt}%04x\\u${l.toInt}%04x, or 0x$cp%06x)")
       // it is a plain single character
       case Array(c) => Some(f"non-printable character (\\u${c.toInt}%04x)")
  // if it is none of the above, then it can be rendered as-is
  case _ => None
```

Examples of use include:

```
scala> whitespaceOrNonPrintable("a")
None
scala> whitespaceOrNonPrintable(" ")
Some(space)
scala> whitespaceOrNonPrintable("\u0000") // NULL
Some(non-printable character (\u0000))
scala> whitespaceOrNonPrintable("🐔")
Some(non-printable codepoint (\ud83d\udc14, or 0x01f414))
```

F.2 Other Built-In Extractors

Compared with MatchParserDemand, which returns wide tokens, the SingleChar extractor can be used to consider a single codepoint instead:

```
trait SingleChar { this: ErrorBuilder[_] =>
    override final
    def unexpectedToken(cs: Iterable[Char], demandedInput: Int, lexicalError: Boolean) =
        val s: String = cs.take(2).mkString
        val cp: Int = s.codePointAt(0)
        whitespaceOrNonPrintable(cp) match
        case Some(name)                                  => Token.Named(name, 1)
        case None if Character.isSupplementaryCodePoint(cp) => Token.Raw(s)
        case None                                        => Token.Raw(s"${cp.toChar}")
}
```

The implementation of SingleChar works by considering the first UTF-16 [Hoffman and Yergeau 2000] codepoint (which may be up to two characters wide): first it checks to make sure it is not a non-printable character or whitespace, otherwise if it represented by two 16-bit characters print the sliced string or the codepoint converted to a plain character.

Dealing with whitespace The MatchParserDemand extractor works well, but may result in carets and unexpected items stretching across whitespace: while this might be desirable, often whitespace delimits tokens in a language, so this may not look as clean. Instead, the TillNextWhitespace token extractor can be used to use whitespace as a token delimiter:

```
trait TillNextWhitespace { this: ErrorBuilder[_] =>
    def trimToParserDemand: Boolean
    def nonWhitespace(c: Char): Boolean = !c.isWhitespace

    override final
    def unexpectedToken(cs: Iterable[Char], demandedInput: Int, lexicalError: Boolean) =
        whitespaceOrNonPrintable(cs) match
        case Some(name) => Token.Named(name, if trimToParserDemand then demandedInput else 1)
        case None       =>
            val tok = cs.takeWhile(nonWhitespace(_))
            if trimToParserDemand then Token.Raw(takeCodePoints(tok, demandedInput))
            else                       Token.Raw(tok.mkString)
}
```

The abstract method trimToParserDemand controls whether or not this extractor should also behave like MatchParserDemand by cutting tokens that are too wide down to the size the parser actually tried to read. In either case, as many non-whitespace characters as possible are taken off the input stream cs and either returned or trimmed. This extractor, with trimToParserDemand = true, is the default extractor for parsley.