Garnishing Parsec with Parsley

Jamie Willis
Department of Computer Science
University of Bristol
United Kingdom
jw14896@my.bristol.ac.uk

Nicolas Wu
Department of Computer Science
University of Bristol
United Kingdom
nicolas.wu@bristol.ac.uk

Abstract

Parser combinators are a clean and powerful abstraction which can provide reasonably efficient means of parsing a grammar into a form which the programmer desires. They remain close to the domain of grammars whilst at the same time offering enormous flexibility. In Haskell, the Parsec library is a prime example of such a library. However, a direct translation to Scala proves to be unbearably slow. This paper describes the semantics and design of a new library, called Parsley, which retains a close resemblance to Parsec style whilst providing very competitive performance.

CCS Concepts • **Software and its engineering → General programming languages**; • **Social and professional topics** → *History of programming languages*;

Keywords Parsers, Combinators, Domain-Specific Languages

ACM Reference Format:
Jamie Willis and Nicolas Wu. 2018. Garnishing Parsec with Parsley. In *Proceedings of the 9th ACM SIGPLAN International Scala Symposium (Scala '18), September 28, 2018, St. Louis, MO, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3241653.3241656

1 Introduction

Parsers are an important tool for computer scientists; they are needed for many practical tasks: compiler writing, JSON parsing, XML parsing, and more. Traditionally, parsers have either been written by hand or by using a parser generator library. The handwritten approach is error-prone but does provide precise error messages and excellent performance. On the other hand, parser generators are closer to the original grammar but often are more rigid in what they produce.

The middle ground between the two is a parser combinator library [Hutton and Meijer 1996; Swierstra and Duponcheel 1996; Wadler 1985]. Unlike a parser generator, parser combinator libraries are written directly in a host language where the result of parsing will be used; this allows them to remain very flexible and very powerful. Despite this, they still remain much closer to the original grammar, reducing the possibility for bugs and keeping the parser readable. They are also theoretically very well grounded; by creating parser combinator libraries that adhere to the laws of applicative functors, PEG (Parser Expression Grammar) parsers which are well-behaved can be produced [Adams and Ağacan 2014; Ford 2004]. Adhering to the monad laws permits the same for context-sensitive grammars[1].

Many modern examples of parser combinator libaries in Haskell are in the Parsec family of libraries [Kmett 2011; Leijen and Meijer 2001; Martini and Leijen 2015; O'Sullivan 2012]. Their implementations are continuation style parsers that adhere to the monad laws. The problem, however, is that they do not translate well to Scala. The library *Parsec for Scala*[2] was a direct rewrite of Parsec, in Scala. The problem with this implementation is that the continuation passing style creates a large overhead from function construction. In addition, since Scala has limitations on tail call optimisation, they are liable to stack overflow unless expensive workarounds, such as trampolines [Baker 1995], are used.

Contributions Our contribution is to provide a formal semantics for Parsley[3], a new parser combinator library with a familiar feel to the Parsec family, but with drastically improved performance for Scala. The idea of Parsley is to create a deep embedding for a parsing Domain-Specific Language (DSL) which compiles into an instruction set, for a stack-based parsing machine, which executes very efficiently. In addition, the compiler is capable of performing a whole host of optimisations relying on the aforementioned laws. This paper provides a small-step structural semantics for the parsing machine which helps to prove the properties of parsers and provide a concrete meaning for the combinators.

Outline The rest of the paper is structured as follows:

- Section 2 provides a motivating example of how our applicative parsers operate as well as providing a description of the language of parsing itself. It will discuss

Scala '18, September 28, 2018, St. Louis, MO, USA

ACM ISBN 978-1-4503-5836-1/18/09…$15.00
https://doi.org/10.1145/3241653.3241656

[1] As a monadic parser combinator library, our library, Parsley, is capable of realising PEGs and context free grammars with backtracking capabilities.
[2] https://bitbucket.org/J_mie6/parsec-for-scala/wiki/Home
[3] https://github.com/J-mie6/Parsley

how users can choose to remain close to the grammar or deviate slightly for the same of performance and idiomatic representation.
- Section 3 discusses how the compiler is structured, what its duties are, what optimisations it performs and how it can be kept fast and robust.
- Section 4 presents the semantics of the virtual machine and describes how combinators can be decomposed into these instructions. The method by which the laws are proven is also outlined.
- Section 5 provides anecdotal evidence for why our library is a good contribution and show that it outperforms similar libraries such as FastParse [Haoyi 2014] and can even compete with hand-written parsers.
- Section 6 outlines the future work.
- Section 7 summarises the work and its consequences.

2 The Language

The translation of a grammar into Parsley parser is straightforward and maintains a close resemblence to the original grammar. Consider this simple expression language:

```
ident  ::= "x" | "y" | "z"
num    ::= digit+
expr   ::= factor "+" expr
factor ::= atom "*" factor
atom   ::= var | num | "(" expr ")"
```

It is instructive to see how this grammar can be translated into the Parsley parser combinator library. Firstly, a datatype which will encapsulate the result of our parsing must be produced. This is not limited to producing ASTs, however; instead imagine that parsing an expression might result in a function of type `(Int, Int, Int) => Int`, where values for each of the variables `x`, `y`, and `z` are fed in and produce the result of evaluating the expression in that context.

```
trait Exp
case class Ident(name: String) extends Exp
case class Num(x: Int)         extends Exp
case class Add(x: Exp)(y: Exp) extends Exp
case class Mul(x: Exp)(y: Exp) extends Exp
```

The parser which corresponds to the original grammar generating an instance of our `Exp` data-type can be produced:

```
val ident = "x" <|> "y" <|> "z"
val num: Parsley[Int]
  = some(digit).map(_.mkString.toInt)
lazy val expr: Parsley[Exp]
  = Add <#> factor <* '+' <*> expr
lazy val factor: Parsley[Exp]
  = Mul <#> atom <* '*' <*> factor
lazy val atom: Parsley[Exp]
    = (Ident <#> ident
  <|> (Num <#> num)
  <|> ('(' *> expr <* ')'))
```

$$\texttt{p.map(identity) = p} \tag{1}$$
$$\texttt{p.map(f).map(g) = p.map(f andThen g)} \tag{2}$$
$$\texttt{pure(identity) <*> p = p} \tag{3}$$
$$\texttt{pure(f) <*> pure(x) = f(x)} \tag{4}$$
$$\texttt{u <*> pure(x) = pure(f => f(x)) <*> u} \tag{5}$$
$$\texttt{pure(\_ compose \_) <*> u <*> v <*> w = u <*> (v <*> w)} \tag{6}$$
$$\texttt{p flatMap (pure(\_)) = p} \tag{7}$$
$$\texttt{pure(x) flatMap f = f(x)} \tag{8}$$
$$\texttt{p.flatMap(f).flatMap(g) = p flatMap \{x => f(x) flatMap g\}} \tag{9}$$
$$\texttt{p <|> (q <|> r) = (p <|> q) <|> r} \tag{10}$$
$$\texttt{empty <|> p = p = p <|> empty} \tag{11}$$

Figure 1. Functor (1, 2), Applicative (3, 4, 5, 6), Monad (7, 8, 9), and Alternative laws (10, 11).

The parser itself is very close to the original representation that was provided. The differences are mainly the addition of the `<#>` and `<*>` operators which help construct the tree and the `<*` and `*>` operators which guide the results. Crucially, this parser works because it has been left-factored to remove left-recursion [Aho et al. 1986]. An alternative approach to solving left-recursion is by using so-called *chain* combinators, which act like folds at the parser level [Fokker 1995]. This deviates from the original form of the grammar, but can be more performant. The `f <#> p` operator is a synonym for `map`, which applies function `f` to the result of parser `p`. The operator `pf <*> px` takes a result of `pf`, a function, and applies it to the result of `px` [Mcbride and Paterson 2008]. Both `p <* q` and `p *> q` execute `p` first then `q` but return the result of the parser pointed to by the arrow. These are by far the most common operations found in Parsley, but `flatMap` (or `>>=`) and a more idiomatic `lift2` operation, which acts on uncurried functions are available, as well as other composite operations. To retain ease-of-use for users of Scala's combinator libraries, Parsley also supports the more familiar `<~>` and the following relation between the two holds: `p <*> q = (p <~> q) map {case (f, x) => f(x)}` [Mcbride and Paterson 2008].

Importantly, our library observes a series of laws (found in Figure 1) that govern the use of the `<*>`, `flatMap`, and `<|>` operations as well as any derived operations.

These laws have been proven true for Parsley (found in Section 4 and the appendices). They also form the basis for many of the Parsley compiler's optimisations. Importantly, Laws (5) and (6) permit an optimisation known as applicative fusion [Delbianco et al. 2012]: any pure computations can be moved to the beginning and performed once.

The semantics of `p <|> q` is deterministic; if `p` succeeds, `q` is not attempted. More specifically, Parsley follows Parsec semantics closely: `p` *must not* consume input in order for `q` to

be attempted. This helps produce better error messages. This behaviour can be avoided using the `attempt` combinator.

3 Deep Embedding

Creating a compiler for an EDSL like Parsley presents a few challenges. This section will first outline what is meant by a deep embedding for parsers and then describes the responsibilities of the compiler and briefly outlines the issues and how they were addressed. Then the compiler optimisations will be described and some of the more interesting optimisations capable of being performed will be presented.

3.1 How is Parsley Represented?

Parsley is a *deep embedding*: the language is represented by objects and behaviours are realised as methods on an abstract trait. In Parsley's case, the deep embedding provides methods for code generation and optimisation and classes for each core combinator. The advantage of a deep embedding over a shallow embedding is that it significantly easier to optimise using pattern matching on constructors instead of bytecode. In addition, global state is not required, instead using local state transferred by the function calls. It does, however, introduce additional overhead from the construction of objects; this is mitigated by re-using objects (by means of mutable state) when possible during optimisation of ASTs.

Composites Each combinator in the language maps to one or more case classes. Only core combinators are given specific classes. Composite operators correspond to their definitions in terms of the core combinators. This is known as a *smart-constructor* pattern. The advantage compared to producing a new case class for each combinator is that it means that optimisations have a single form, which reduces the amount of code required considerably, also improving performance during pattern matching. As an example, consider law (2), using the definition of `p.map(f)` as `pure(f) <*> p`: the law now has four different forms depending on whether `<*>` or `map` was used. This obviously increases the patterns by four-fold and this is not ideal.

Core The case classes for the core combinators follow a similar structure. For instance, the implementation of the deep embedding constructor for the `<*>` combinator is:

```
case class <*>[A, +B](_pf: =>Parsley[A=>B],
                      _px: =>Parsley[A])
  extends Parsley[B] {
  lazy val pf = _pf
  lazy val px = _px
  def preprocess
    (implicit seen: Set[Parsley[_]])
    = ...
  def optimise = (pf, px) match {
    case (Pure(f), Pure(x)) => Pure(f(x))
    ... }
  def codeGen(instrs: Buffer[Instr],
              labels: Counter): Unit = {
    pf.codeGen(instrs, labels)
    px.codeGen(instrs, labels)
    instrs += instructions.Apply }}
```

This shows the three phases of the compiler whose implementation is delegated to the AST nodes. The node itself takes in both arguments of the combinator as by-name parameters and assigns these to lazy values. The `preprocess` stage is where recursion analysis happens. The `optimise` method performs pattern matching on the already optimised subtrees and applies optimisations based on the laws. The code generation phase is a recursive descent process which first compiles the subtrees interleaved with instructions used for this combinator (in this case just an `Apply` is inserted). Optimisations are performed here but are omitted for brevity.

3.2 Responsibilities

The first responsibility of the compiler is a very important one. Recursive parsers pose a problem when constructing an AST: as the tree is built there is a risk of entering an infinite loop at the recursion points. In order to prevent this, it is necessary to first construct a completely lazy tree and then perform analysis to identify recursive parts of the parser and ensure they are not re-evaluated (in essence, finding fixpoints of the tree). In order to create a completely lazy tree, the combinators must also be lazy on both sides. This is accomplished by using an implicit class with a by-name parameter (a parser) which in turn provides every combinator to the user.

The next responsibility of the compiler is to propagate error messages through the tree before performing so-called *lawful* optimisations on the AST. These optimisations are ones which rely on the applicative and monad laws.

Finally, the tree is traversed once more in order to produce the bytecode for the parser. During this phase, a second round of optimisation is performed which focus on peephole style optimisations that are more specific to our machine. Then it must process the bytecode in order to remove labels ready for execution.

3.3 Issues

There are five main issues with the approach taken, which will be described and addressed in the following paragraphs.

Laziness Overhead The first issue is the lazy ASTs themselves: whilst it is necessary to utilise laziness in order to prevent infinite recursion as the AST is constructed, it does add overhead to the remainder of the compilation process. Once recursion analysis is complete, however, laziness is no longer required but optimisations still have to perform a lazy value access every time a parser is inspected. To fix this, the compiler makes use of mutable state along with by-name parameters to provide a one-time lazy barrier instead.

```
case class <*>[A, B](_pf: =>Parsley[A=>B],
                     _px: =>Parsley[A])
  extends Parsley[B] {
  var pf: Parsley[A=>B] = _
  var px: Parsley[B]    = _
  def preprocess
   (implicit seen: Set[Parsley[_]]) = {
    pf = _pf.fix.preprocess.optimise
    px = _px.fix.preprocess.optimise
    this
  }
  def optimise = (pf, px) match ...
  def codeGen(...) = /*unchanged*/
}
```

The code above illustrates the changes that are made. The lazy values originally used have been replaced by mutable variables initially left empty. These variables are assigned during preprocessing after having been analysed for potential recursion with `fix`. The `optimise` method is unchanged, but notice that the pattern matching no longer inspects lazy values, instead using mutable variables directly.

Stack Overflows The second issue is that the preprocessing stage of the compiler will have to perform potentially very deep tree traversals. This runs the risk of stack overflow for big parsers. In order to fix this, the compilation process is converted to utilise continuation-passing style, then a trampoline is used to carefully execute the traversal. Both preprocessing and code generation are CPS transformed, however the code in `optimise` is not, since, in practice, recursion does not get too deep: it is bounded as a result of optimisation normal forms (see Section 3.4.1). The result of the transform is:

```
case class <*>[A, B](_pf: =>Parsley[A=>B],
                     _px: =>Parsley[A])
  extends Parsley[B] {
  var pf: Parsley[A=>B] = _
  var px: Parsley[B]    = _
  def preprocess(cont: Parsley[B]
                    => Bounce[Parsley[_]])
       (implicit seen: Set[Parsley[_]]) =
  {
    _pf.fix.preprocess(pf => Thunk {
      _px.fix.preprocess(px => Thunk {
        this.pf = pf.optimise
        this.px = px.optimise
        cont(this)
  })})}
  def optimise = /*unchanged*/
  def codeGen(...)
             (cont: =>Bounce[Unit]) = {
    Thunk(pf.codeGen(instrs, labels){
      px.codeGen(instrs, labels){
        instrs += instructions.Apply
        cont
}})}}
```

The `preprocess` method has been dramatically changed: each subtree is processed and continues by processing the next, finishing by calling the provided continuation `cont` which defers back to parent trees. Code generation is similarly adjusted: continuations guide the control flow through generating code for the subtrees, in order, as well as inserting the relevant instructions.

Over-Thunking The next issue is that the thunks introduced by the trampoline also adds runtime overhead, and this can be alleviated somewhat by only ensuring a thunk is generated every n recursion levels.

Aggressive Inlining Finally, there is an issue with Parsley's approach to optimisation. Parsley employs a policy of extreme inlining: everything is inlined if it can be. Whilst this does allow for more opportunities to optimise, even across parsers, it can cause an explosion in the size of the tree itself. Consider the `some` combinator defined like so: `def some[A](p: =>Parsley[A]) = lift2(_::_, p, many(p))`. Since `p` is used twice, the effective size of the AST has doubled. This can cause serious performance issues at compile-time, and minor performance issues at run-time. In order to prevent this, a unary prefix operator `+` is introduced which denotes a parser should be compiled as a subroutine instead of inlined. To put this into perspective, an expression parser for the C programming language is 300 instructions using subroutines and 100,000 instructions otherwise.

3.4 Optimisation

Two types of optimisations have been alluded to: lawful and peephole. Each will now be described in a little more detail.

3.4.1 Lawful Optimisations

As discussed, the first set of optimisations the compiler performs is a bottom-up pass for lawful optimisation. These optimisations have two roles: re-association, and application of the laws outlined in Figure 1.

Re-association It is useful to create normal forms for sequences of the same combinator. For instance, `<|>` is always rearranged to be right-associative, as is `<*`, but, in constrast, `*>` is always left-associative. Another important reassociation is `(q *> pf) <*> px` $\Rightarrow$ `q *> (pf <*> px)`, which allows us to push chains of `<*>`s together to perform fusion optimisations. There is an corresponding, similar, version of this re-association for `<*` and `<*>`. These re-associations arise from the applicative laws and, as a result, are legal.

A reader familiar with Parsec might wonder why right-associativity for the normal forms of `<|>` and `<*` is chosen when, traditionally, these operators are left-associative. Firstly, the ability to perform these re-associations follows

from the applicative laws ((4), (5), and (6) with definition `p <* q = lift2(x => _ => x, p, q)`) and alternative law (10), which means it does not really matter which is chosen from the users perspective. The reason this is actually useful is to facilitate some optimisations. For `<|>`, an important optimisation which creates jump tables requires a linear iteration over a sequence of `<|>`s and this is easier to perform in right-associative normal form. For `<*`, the result is obtained from the leftmost parser in a chain; it is, therefore, easier to find the result of a chain in a right-associative form ($O(1)$ lookup versus $O(N)$). This is also the reason `*>` is chosen to be left-associative, with the reasoning reversed.

Law Application Applying the applicative, functor, alternative and monad laws directly to the tree can, in many cases, reduce the number of operations performed. For instance, Law (2) tells us that a parser like `p.map(f).map(g)` can be written as `p.map(f.andThen(g))`; this is better since the single `map` can be performed with one instruction instead of two. The compiler performs this automatically for us. There are two things that make lawful optimisations difficult, however: `flatMap` and side-effectful parsers. Firstly, side-effectful parsers prevent optimisation occurring as the laws rely on pure computation. Parsers can be marked as unsafe in order to disable optimisation and obey the correct semantics, however. Secondly, `flatMap`, is a problem because the provided function must be evaluated in order to perform optimisations on that part of the tree, but clearly, this is not possible until runtime. Functions cannot be tested for equality, so this cannot be used as an avenue to perform optimisations involving Law (7). Whilst there are a few limited optimisation performed on monadic parsers (Laws (8) and (9)), they are not as useful as those for applicatives. Because of this, several intrinsic combinators were introduced to perform commonly used monadic tasks without `flatMap`.

3.4.2 Peephole Optimisations

The second class ofoptimisations, peephole optimisations, are performed from the top-down during code generation. They are less tailored to the abstraction of parsing and more to the underlying machine which is targetted. They are split into roughly three categories themselves.

Strength-Reduction The first is strength-reduction: the aim is to convert core combinators like `<*>` into combinators such as `map` whenever possible, for instance. The deep embedding offered by the compiler only has AST nodes for non-composite operations as this allows for a cleaner interface for optimisation (no repeated patterns to account for separate forms). The optimiser is responsible for selecting the correct instruction to correspond to composite structures.

Instruction Selection The second category is optimised instruction selection. The underlying stack machine supports several optimised instructions, such as `CharTokFastPerform`, which reads a character and performs some function on it (at compile-time). The compiler selects these instructions where possible, using pattern matching. There are even specialised instructions for common patterns like `attempt(p) <|> q`.

Complex Optimisations The third category are more complex optimisations like jump table generation. Jump tables provide a means of reducing a chain of `<|>` operations from $O(N)$ runtime in worst case to $O(1)$ by precomputing a jump table from the leading characters of the branches. It proves to be a very effective optimisation. The application of the jump table optimisation is completely tail recursive and runs much faster than the linear approach.

4 Semantics

In this section, the architecture of the machine we have designed, as well as some of the considerations taken to ensure its performance on the JVM and in Scala, in particular, will be described. Then, a formal semantics of the instruction set is presented, followed by a description of how the combinators relate to these instructions.

4.1 The Machine

Our machine is a stack machine with five different stacks. Let Γ be the context of the machine, which is parameterised by $(i, o, s, X, H, \Sigma, C, F)$. For the purposes of brevity, error message tracking states are omitted. The lowercase symbols represent elements other than the stacks: i is the program counter, o is the input offset, and s is the status of the machine (Good, Recover or Failed). The stacks are as follows:

- **Operational** X is the more traditional of the five; it is used to store the intermediate results of computations performed on parsers.
- **Handler** H is used to store information about the current branches and control flow of parsers in *handlers*. A handler is a similar concept to an exception handler: it dictates control flow in case of failure. Formally, it is a tuple containing the size of the operational stack sz, jump label h, and recursion depth d.
- **State** Σ keeps a record of the previous state of the machine at different branch points so that it can be rolled back if necessary. For the semantics, the error position state and the stateful *registers* is omitted for brevity leaving a record of the offset o.
- **Check** C is used to store past input offset history in order to realise the Parsec semantics of `<|>` (only take second branch if first failed.)
- **Call** F is used to track recursive calls to parsers. Each frame consists of a tuple of return address ret and instruction buffer I'.

Note that not all the stacks are in use at any one time, some instruction pairs use the check stack and others use the state stack. It is for this reason that the handler, state, and check stacks are not merged into a single one: it would be

wasteful to have to manipulate all the information in every instruction using all three stacks. The stacks themselves are not `Lists` as might be expected. The operational stack is a custom resizing array stack which proves to be the fastest implementation from the microbenchmarks: the operational stack is a `ArrayStack[Any]` and `Array` is heavily tied in with the `Any` type. The remainder of the stacks are implemented using a similar class to `List`, with a mutable head for fast top-of-stack replacement operations and a smaller, more specialised interface.

4.2 Notation

Before the instruction set and its semantics are introduced, notation that is used to present it will be outlined.

During execution, the machine will be parameterised by a pair $\langle I[i], \Gamma \rangle$ where $I[i]$ is a buffer of instructions indexed at position i. Where a value from Γ must be extracted, we write $v \leftarrow \Gamma$ with v providing a syntactic indication of the element in Γ. This syntax is extended by allowing for multiple extractions where required.

In order to perform an update to the context Γ for some named field Y with new value Y' we adopt the following notation $\Gamma(Y \mapsto Y')$ where Y' is an expression, for instance $y :: Y$ represents a prepending operation onto the stack Y. Any parts of the context that are not referenced by the update must remain unchanged.

Finally, pattern matching may be performed on any extracted values. These pattern matches act as semantic constraints and must hold true. In addition, there is an implicit pattern match on $I[i]$ for the purposes of identifying which semantics is being defined.

4.3 Applicative Instructions

Now, applicative operations are described[4], starting with the simplest applicative instruction, `Push`, corresponding directly to `pure`, this rule is called *Push*:

$$\frac{(i, X) \leftarrow \Gamma}{\langle \texttt{Push}(x), \Gamma \rangle \longrightarrow \langle I[i+1], \Gamma(i \mapsto i+1, X \mapsto x :: X) \rangle}$$

From the instruction `Pure(`x`)`, the next state of the machine is derived from adding one to the program counter and putting x on the operational stack with no further requirements from the state.

A slightly more complex instruction is `Apply`, which relates to the `<*>` combinator, given by rule *Apply*:

$$\frac{(i, x :: f :: X') \leftarrow \Gamma \mid x : \alpha,\ f : \alpha \to \beta}{\langle \texttt{Apply}, \Gamma \rangle \longrightarrow \langle I[i+1], \Gamma(i \mapsto i+1, X \mapsto f(x) :: X') \rangle}$$

From the instruction `Apply`, if it is the case that an operational stack with at least two elements $x : \alpha$, $f : \alpha \to \beta$ and remainder X can be extracted, then the machine can be resolved to a new state with the program counter incremented and the result of placing $f(x)$ onto the remainder of stack X.

Now the semantics of `Pop`, used for `<*` and `*>`, are given by rule *Pop*:

$$\frac{(i, x :: X) \leftarrow \Gamma \mid x : \alpha}{\langle \texttt{Pop}, \Gamma \rangle \longrightarrow \langle I[i+1], \Gamma(i \mapsto i+1, X \mapsto X) \rangle}$$

From the instruction `Pop`, a single element of the stack X is discarded, and the program counter is incremented.

4.4 Handler Instructions

Figure 2 presents the remainder of the semantics. The rules for InputCheck and PushHandler describe the key instructions for setting up control flow in parsers. `InputCheck` is an instruction which is paired with the `JumpGood` instruction to realise the `<|>` combinator. Rule $\mathit{JumpGood}_2$ defines how failure can be recovered from when no input was consumed between it and the corresponding `InputCheck` instruction. In contrast, $\mathit{JumpGood}_3$ resolves to the same semantics as `Empty`, representing true failure. If there are no handlers present (Empty_1), then the machine results in total failure (defined by Step_3) otherwise the next handler on the stack is used to fallback to the next recovery point. In a similar vein, $\mathit{Attempt}_2$ resolves to the *Empty* rules. The `Attempt` instruction is paired with the `PushHandler` instruction. The use of handlers allows the machine to perform chaining of `<|>` operations or allow combinators to be visited during failure like `attempt`.

4.5 Call Stack Interactions

The rule *DynSub* gives the semantics of the `flatMap` operation. The instructions placed into the buffer by the instruction are removed as illustrated by the rule Step_2, which pops a frame off the call stack in order to restore it (see *Return*). This also pairs with the behaviour for the `Call` instruction. In the *Call* rule, the semantics demand a copy of the instructions to be performed: this allows for the use of special state containing instructions to be used which must be copied to provide a fresh state for a recursive call (a semantics of these instructions is not provided as they are not core instructions). In Empty_3 the semantics makes reference to a *drop* function which allows us to remove n elements of a stack. This ensures that multiple returns are performed if a handler is further back in the call stack, similar to an exception handler in a traditional sense. The `GoSub` and `Return` instructions are paired; the *GoSub* rule shows that the call stack is needed to manage sub-routine calls, but the instruction buffer is not changed: a jump elsewhere in the same buffer is performed.

4.6 Input Interactions

Interactions with the input are defined by the rules *Char*, *Sat* and *Look*. Whilst `String` is considered as a core instruction its semantics are not explicitly defined, instead it is given

[4] `pf <*> px` compiles to `Pf; Px; Apply`. `pure(x)` compiles to `Push(x)`. Then `p *> q` compiles to `P; Pop; Q` with `p <* q` compiling to `P; Q; Pop`. The relationships for other combinators can be found in Appendix E.

$$InputCheck\ \frac{(i, o, X, C, H, F) \leftarrow \Gamma}{\langle \texttt{InputCheck}(h), \Gamma\rangle \longrightarrow \langle I[i+1], \Gamma(i \mapsto i+1, C \mapsto o :: C, H \mapsto (|X|, h, |F|) :: H)\rangle}$$

$$PushHandler\ \frac{(i, o, X, H, \Sigma, F) \leftarrow \Gamma}{\langle \texttt{PushHandler}(h), \Gamma\rangle \longrightarrow \langle I[i+1], \Gamma(i \mapsto i+1, H \mapsto (|X|, h, |F|) :: H, \Sigma \mapsto o :: \Sigma)\rangle}$$

$$JumpGood_1\ \frac{(s, \_ :: H', \_ :: C') \leftarrow \Gamma \mid s = \text{Good}}{\langle \texttt{JumpGood}(l), \Gamma\rangle \longrightarrow \langle I[l], \Gamma(i \mapsto l, H \mapsto H', C \mapsto C')\rangle}$$

$$JumpGood_2\ \frac{(i, o, s, o' :: C') \leftarrow \Gamma \mid s = \text{Recover}, o = o'}{\langle \texttt{JumpGood}(l), \Gamma\rangle \longrightarrow \langle I[i+1], \Gamma(i \mapsto i+1, s \mapsto \text{Good}, C \mapsto C')\rangle}$$

$$JumpGood_3\ \frac{(o, s, o' :: C') \leftarrow \Gamma \mid s = \text{Recover}, o \neq o'}{\langle \texttt{JumpGood}(l), \Gamma\rangle \longrightarrow \langle \texttt{Empty}, \Gamma(C \mapsto C')\rangle}$$

$$Empty_1\ \frac{H \leftarrow \Gamma \mid H = \epsilon}{\langle \texttt{Empty}, \Gamma\rangle \longrightarrow \langle \epsilon, \Gamma(s \mapsto \text{Failed})\rangle}$$

$$Empty_2\ \frac{((sz, h, d) :: H', X, F) \leftarrow \Gamma \mid d = |F|}{\langle \texttt{Empty}, \Gamma\rangle \longrightarrow \langle I[h], \Gamma(i \mapsto h, s \mapsto \text{Recover}, H \mapsto H', X \mapsto drop(|X| - sz, X))\rangle}$$

$$Empty_3\ \frac{((sz, h, d) :: H', X, F) \leftarrow \Gamma \mid d < |F| \longrightarrow (I', \_) :: F' \leftarrow drop(|F| - d, F)}{\langle \texttt{Empty}, \Gamma\rangle \longrightarrow \langle I'[h], \Gamma(i \mapsto h, s \mapsto \text{Recover}, H \mapsto H', X \mapsto drop(|X| - sz, X), F \mapsto F')\rangle}$$

$$Attempt_1\ \frac{(i, s, \_ :: H', \_ :: \Sigma') \leftarrow \Gamma \mid s = \text{Good}}{\langle \texttt{Attempt}, \Gamma\rangle \longrightarrow \langle I[i+1], \Gamma(i \mapsto i+1, H \mapsto H', \Sigma \mapsto \Sigma')\rangle}$$

$$Attempt_2\ \frac{(s, o' :: \Sigma') \leftarrow \Gamma \mid s = \text{Recover}}{\langle \texttt{Attempt}, \Gamma\rangle \longrightarrow \langle \texttt{Empty}, \Gamma(o \mapsto o', \Sigma \mapsto \Sigma')\rangle}$$

$$DynSub\ \frac{(i, x :: X', F) \leftarrow \Gamma \mid x : \alpha,\ f : \alpha \to I}{\langle \texttt{DynSub}(f), \Gamma\rangle \longrightarrow \langle f(x)[0], \Gamma(i \mapsto 0, X \mapsto X', F \mapsto (I, i+1) :: F)\rangle}$$

$$Char_1\ \frac{(i, o, X) \leftarrow \Gamma \mid In[o] = c}{\langle \texttt{Char}(c), \Gamma\rangle \longrightarrow \langle I[i+1], \Gamma(i \mapsto i+1, o \mapsto o+1, X \mapsto c :: X)\rangle} \qquad Char_2\ \frac{(i, o, X) \leftarrow \Gamma \mid o \geq |In| \vee In[o] \neq c}{\langle \texttt{Char}(c), \Gamma\rangle \longrightarrow \langle \texttt{Empty}, \Gamma\rangle}$$

$$Sat_1\ \frac{(i, o, X) \leftarrow \Gamma \mid p(In[o])}{\langle \texttt{Sat}(p), \Gamma\rangle \longrightarrow \langle I[i+1], \Gamma(i \mapsto i+1, o \mapsto o+1, X \mapsto In[o] :: X)\rangle} \qquad Sat_2\ \frac{(i, o, X) \leftarrow \Gamma \mid o \geq |In| \vee \neg p(In[o])}{\langle \texttt{Sat}(p), \Gamma\rangle \longrightarrow \langle \texttt{Empty}, \Gamma\rangle}$$

$$Look_1\ \frac{(i, s, \_ :: H', o' :: \Sigma') \leftarrow \Gamma \mid s = \text{Good}}{\langle \texttt{Look}, \Gamma\rangle \longrightarrow \langle I[i+1], \Gamma(i \mapsto i+1, o \mapsto o', H \mapsto H', \Sigma \mapsto \Sigma')\rangle} \qquad Look_2\ \frac{(s, \_ :: \Sigma') \leftarrow \Gamma \mid s = \text{Recover}}{\langle \texttt{Look}, \Gamma\rangle \longrightarrow \langle \texttt{Empty}, \Gamma(\Sigma \mapsto \Sigma')\rangle}$$

$$Call\ \frac{(i, F) \leftarrow \Gamma}{\langle \texttt{Call}(P), \Gamma\rangle \longrightarrow \langle copy(P.I)[0], \Gamma(i \mapsto 0, F \mapsto (I, i+1) :: F)\rangle}$$

$$GoSub\ \frac{(i, F) \leftarrow \Gamma}{\langle \texttt{GoSub}(l), \Gamma\rangle \longrightarrow \langle I[l], \Gamma(i \mapsto l, F \mapsto (I, i+1) :: F)\rangle} \qquad Return\ \frac{(I', i') :: F' \leftarrow \Gamma}{\langle \texttt{Return}, \Gamma\rangle \longrightarrow \langle I'[i'], \Gamma(i \mapsto i', F \mapsto F')\rangle}$$

$$Step_1\ \frac{(s, x :: X', F) \leftarrow \Gamma \mid i \geq |I| \wedge F = \epsilon \wedge s = \text{Good}}{\langle I[i], \Gamma\rangle \longrightarrow \langle Success(x)\rangle} \qquad Step_2\ \frac{(i, s) \leftarrow \Gamma \mid i \geq |I| \wedge s = \text{Good}}{\langle I[i], \Gamma\rangle \longrightarrow \langle \texttt{Return}, \Gamma\rangle}$$

$$Step_3\ \frac{s \leftarrow \Gamma \mid s = \text{Failed}}{\langle \_, \Gamma\rangle \longrightarrow \langle Failure(error)\rangle} \qquad Jump\ \frac{}{\langle \texttt{Jump}(l), \Gamma\rangle \longrightarrow \langle I[l], \Gamma(i \mapsto l)\rangle}$$

Figure 2. Full Semantics for Parsley

identical semantics to the sequential application of many character instructions. The instruction `Sat` is used to perform a predicated match on the next character in the input stream (Sat_1) and fails if the predicate is false or there is no remaining input (Sat_2). The *Char* rules follow a similar pattern and, in fact, `char(c) = satisfy(_ == c)`. The reason a standalone instruction was made is to make it slightly more efficient. The `lookAhead` combinator is implemented using both the `PushHandler` and `Look` instructions. The *Look* rules describe how this combinator works: if the state of the machine is successful on entry to `Look` then the state is restored to how it was when the handler was pushed ($Look_1$). On the other hand, failure will not restore the state and instead cleans up the context ready for the next handler ($Look_2$).

4.7 Proving the Laws

Section 2 stated that our parsers follow several laws outlined in Figure 1. These laws can be proven correct using the semantics described in Figure 2. The proofs of the alternative and monad laws are very long and involved, as they must cover each of the cases for the rules corresponding to the utilised instructions. As a result, these many page long proofs are placed into Appendices C and D, however, the proofs of the first two applicative laws will be presented to illustrate the process. The proofs rely on the so-called *single-push assumption* of our parser combinators: each complete combinator will result in exactly one item pushed to the stack on success. This assumption has been proven by induction and the proof will be briefly outlined.

4.7.1 Applicative Laws

The general structure of these proofs is to first compile the law without loss of generality. Then it is shown that the state of the machine at the beginning and end of the parser is the same in both sides of the law. The simplest applicative law is Law (4): `pure(f) <*> pure(x) = pure(f(x))`. This will be compiled to `Push(f); Push(x); Apply` and `Push(f(x))` respectively. The only stack referenced by these two instructions is the operational stack and, as such, this is the only stack considered.

```
{X}
  Push(f)
{f::X} // Rule Push
  Push(x)
{x::f::X} // Rule Push
  Apply
{f(x)::X} // Rule Apply
=
{X}
  Push(f(x))
{f(x)::X} // Rule Push
```

The next law, Law (5), compiles to `U; Push(x); Apply = Push(f => f(x)); U; Apply`. Starting with a stack X it follows that:

```
{X}
  U
{u::X} // By single-push assumption (SPA)
  Push(x)
{x::u::X} // Rule Push
  Apply
{u(x)::X} // Rule Apply
=
{X}
  Push(f => f(x))
{(f => f(x))::X} // Rule Push
  U
{u::(f => f(x))::X} // By SPA
  Apply
{(f => f(x))(u)::X} // Rule Apply
{u(x)::X} // Simplification
```

The final applicative law–Law (6)–follows similarly but is slightly more verbose. Like the proof of Law 5, it makes use of the single-push assumption to produce results on the stack for the three abstract parsers referenced in the law. It is proven in Appendix B.

4.7.2 Single-Push Assumption

The single-push assumption is proven inductively. The base case is every primitive in the language: `pure`, `char`, and so on. From inspection of the associated rules, it is clear to see they all push exactly one item to the operational stack on success. Our inductive hypothesis states that if a parser pushes exactly one item to the stack on success then any combinator that uses it will also push exactly one item to the stack. This is very straightforward, so one of the cases, `<*>`, will be provided and the remainder left to Appendix E. Consider `pf <*> px` which is compiled to `Pf; Px; Apply` and, assuming initial stack X:

```
{X}
  Pf
{f::X} // From inductive hypothesis
  Px
{x::f::X} // From inductive hypothesis
  Apply
{f(x)::X} // Rule Apply
```

Since this parser ended with a single item on the stack, `<*>` follows the single-push assumption. Similar proofs follow for the remaining combinators in the language.

5 Benchmarks

This section presents a variety of benchmarks that support the claims made about Parsley's performance. In order to

provide a fair representation of different libraries, we benchmarked a variety of parsers for different styles of grammar, including JSON parsing, simple expression languages and full programming languages. The input sizes are varied to compare how performance scales with input size as well as see how parser compilation affects the overall performance.

Hot versus Cold Hot parsing is where the parser is compiled once and use it many times whereas cold parsing involves compiling the parser each time it is used. Hot parsers will be more common for long-running server applications or compilers which parse multiple files in one invocation.

5.1 JSON Parsers

The first benchmark compares Parsley, running both hot and cold, against FastParse and Haskell's Attoparsec and Parsec. Whilst other libraries were tested, namely *Atto*, *Scala Parser Combinators*, and *Parsec for Scala*, they performed very poorly (over 30 times slower than Parsley in the best case) and are omitted from the graph for brevity.

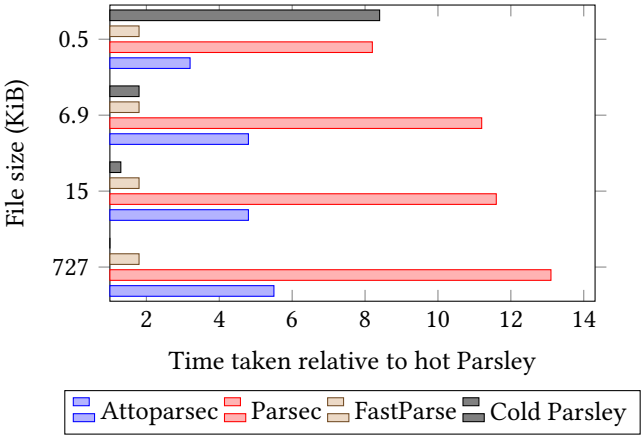

Figure 3. Performance of libraries parsing JSON

Figure 3 shows that a hot run of Parsley is firmly in the lead compared to the other test libraries, around 1.8x faster than FastParse. As file size increases, the cost of cold compilation relative to the cost of hot runs decreases. Both Parsley and FastParse scale very well and get faster per-byte as the size of the file increases. This is not true for Attoparsec or Parsec, however. Further independently derived results show that Parsley can even outperform Parboiled2[5] in this benchmark[6].

5.2 Expression Parsers

This benchmark is designed to stress test parsers for grammars with multiple precedence levels working in a typically left-recursive fashion. The grammar utilised has five different precedence levels consisting of eight operators. For this benchmark we tested two different versions for Parsley, one using the sub-routines mentioned in Section 3.3 and one without, to see what impact they have.

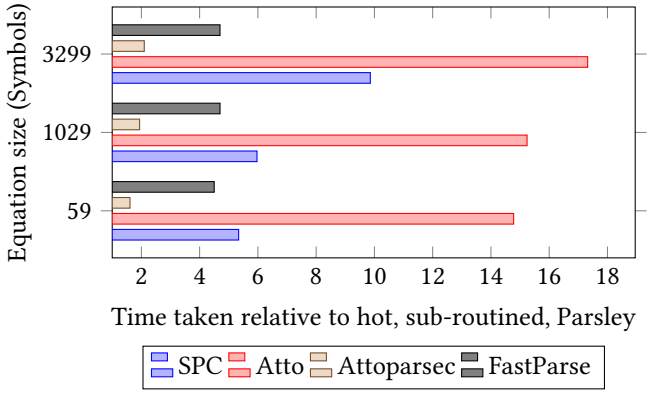

Figure 4. Performance parsing expressions

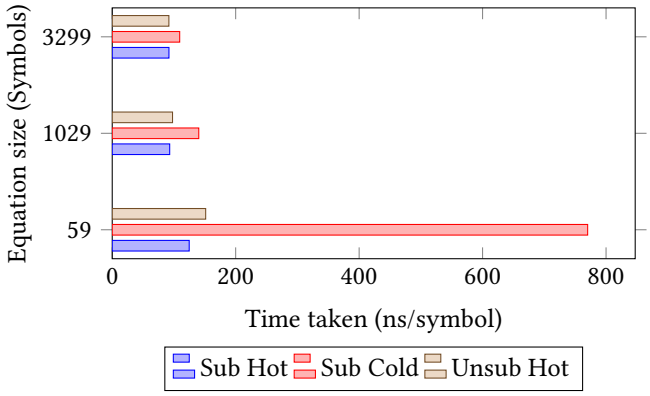

Figure 5. Performance difference for hot and cold sub-routined Parsley (Sub Hot/Cold) and the effect of no sub-routines (Unsub Hot)

Figure 4 again shows that a hot run of parsley is the fastest out of the libraries tested. As with the JSON benchmark, the library shows good scaling for larger expression sizes. Figure 5 shows that using sub-routines improves the performance of even hot parsers (due to less copying of stateful instructions during recursive calls). The effect of no sub-routines on cold parsing was omitted from the graph as it is *significantly* worse than its counterpart.

5.3 Language Parsers

The final benchmark tests performance for parsing full programming languages. The languages selected are Nandlang[7] and Javascript. Nandlang is a language similar to Javascript but much simpler and with no real expression parser requirements. Both languages were tested on a variety of input sizes.

[5] https://github.com/sirthias/parboiled2

[6] https://tom91136.github.io/scala-parser-benchmarks/report.html?sources=data.json

[7] https://github.com/Jellonator/Nandlang/

The handwritten parser used in the Nandlang compiler was also benchmarked after conversion to Scala.

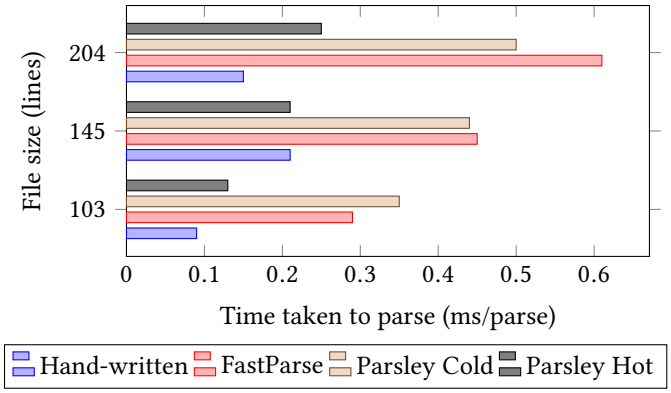

Figure 6. Performance parsing Nandlang

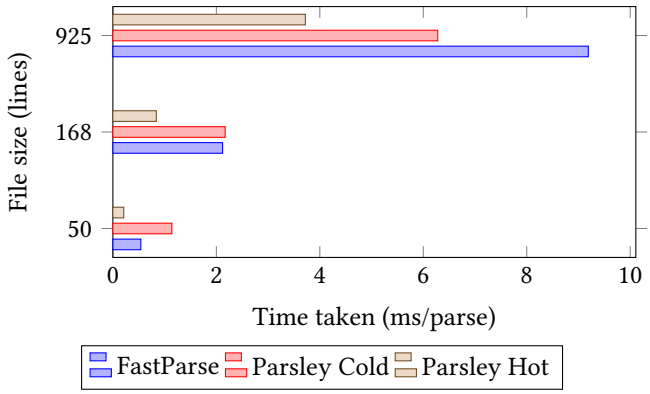

Figure 7. Parsing various Javascript files of different sizes

For Nandlang, Parsley manages to remain up to 2.4x faster than FastParse and cold parsing manages to remain competitive for larger input sizes (Figure 6). Parsley also competes well against the hand-written parser for Nandlang, less than twice as slow. Figure 7 shows yet more promising results, which a cold run of Parsley managing to outpace FastParse at around 200 lines for a source file. Again, both libraries maintain good scaling with input size.

6 Future Work

Parsley can be further expanded in a number of ways.

Firstly, it is desirable to support streaming parsers. It will be necessary to construct a mechanism for streaming which is capable of performing backtracking within the stream while preserving the performance of non-streamed parsing.

Secondly, if the library were able to automatically left-factor grammars by utilising the chain combinators discussed in Section 2, then this allows parsers to remain closer to grammars and not sacrifice efficiency or semantics.

Thirdly, it would be advantageous to be able to serialise parsers. This would, essentially, eliminate the need for cold parsing and allow the compiler to perform even more expensive optimisation without affecting the runtime of the program. Closures are hard to serialise, however, and this is key to serialising the bytecode. The Spores library [Miller et al. 2013] provides a promising avenue for exploration.

7 Conclusion

In this paper, we have introduced the foundations behind Parsley, a fast parser combinator library written in Scala. The main idea is that Parsley is a compiled DSL for parsing: a compiler operates over an AST creates by the use of the combinators, optimising it and then producing bytecode for the machine. In order for this machine to successfully handle recursive parsers, it was necessary to ensure the entire AST was lazy so that fix-point analysis can be performed. This resulted in inefficiencies from lazy value lookup, which were mitigated by creating one-time lazy barriers. Additionally, the prevention of stack overflow was achieved by converting the compiler into continuation passing style and making use of a trampoline. In order to improve the efficiency of parsers with duplication of subtrees, sub-routining was introduced to ensure parsers are only compiled once in these cases.

This paper also described the underlying machine itself and presented a formal small-step structural semantics for the machine's instruction set, linking this back to the high-level combinators. This semantics described how the machine operates and why each part of it is required. It serves to show how the original desired semantics of Parsec is realised by our machine. Proofs of the applicative laws were outlined as well as a key assumption for the proof, which was shown to be proven inductively.

Finally, a set of benchmarks was presented, which illustrated that Parsley's approach results in parsers with competitive performance for many different styles of parsing problem. These parsers also scale well, at least linearly, and tend to become more efficient as size increases.

With Parsley, Scala programmers can remain close to a grammar and at the same time continue to work in the domain they are comfortable in. The semantics presented ensure that the machine works correctly and, as a result, programmers are able to write parsers which perform reliably and sensibly whilst, at the same time, also being very fast and robust to design choice. Parsley allows for context-sensitive parsing, important for parsing indentation sensitive languages like Python, without much effort on the programmer's part by allowing for use of *for comprehensions* to simplify this procedure. This is something that is not so straightforward using parser generators. The independence from macros allows Parsley to additionally be robust entering the Dotty phase of the Scala lifecycle.

Acknowledgements

The authors would like to acknowledge Tom Lin for his work on expanding the benchmarks and ensuring our own benchmarks were non-biased. We also thank Sam Russell for his useful comments on the drafts of this paper.

References

Michael D. Adams and Ömer S. Ağacan. 2014. Indentation-sensitive Parsing for Parsec. *SIGPLAN Not.* 49, 12 (Sept. 2014), 121–132. https://doi.org/10.1145/2775050.2633369

Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. 1986. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

Henry G. Baker. 1995. CONS Should Not CONS Its Arguments, Part II: Cheney on the M.T.A. *SIGPLAN Not.* 30, 9 (Sept. 1995), 17–20. https://doi.org/10.1145/214448.214454

Germán Andrés Delbianco, Mauro Jaskelioff, and Alberto Pardo. 2012. Applicative Shortcut Fusion. In *Trends in Functional Programming*, Ricardo Peña and Rex Page (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 179–194.

Jeroen Fokker. 1995. Functional Parsers. In *Advanced Functional Programming, First International Spring School on Advanced Functional Programming Techniques-Tutorial Text.* Springer-Verlag, Berlin, Heidelberg, 1–23. http://dl.acm.org/citation.cfm?id=647698.734153

Bryan Ford. 2004. Parsing Expression Grammars: A Recognition-based Syntactic Foundation. *SIGPLAN Not.* 39, 1 (Jan. 2004), 111–122. https://doi.org/10.1145/982962.964011

Li Haoyi. 2014. FastParse: Writing Fast Parsers Fast in Scala. http://www.lihaoyi.com/fastparse/. (2014).

Graham Hutton and Erik Meijer. 1996. *Monadic parser combinators.* Technical Report. University of Nottingham.

Edward A. Kmett. 2011. Trifecta: Parser combinators with slicing and diagnostic support. http://hackage.haskell.org/package/trifecta. (2011).

Daan Leijen and Erik Meijer. 2001. *Parsec: Direct Style Monadic Parser Combinators For The Real World.* Technical Report. Microsoft.

Paolo Martini and Daan Leijen. 2015. Megaparsec: Monadic parser combinators. https://hackage.haskell.org/package/megaparsec. (2015).

Conor Mcbride and Ross Paterson. 2008. Applicative Programming with Effects. *J. Funct. Program.* 18, 1 (Jan. 2008), 1–13. https://doi.org/10.1017/S0956796807006326

Heather Miller, Martin Odersky, and Philip Haller. 2013. SIP-21 – Spores. https://docs.scala-lang.org/sips/spores.html. (2013).

Bryan O'Sullivan. 2012. Attoparsec: Fast combinator parsing for bytestrings and text. https://hackage.haskell.org/package/attoparsec. (2012).

S. Doaitse Swierstra and Luc Duponcheel. 1996. Deterministic, Error-Correcting Combinator Parsers. In *Advanced Functional Programming, Second International School-Tutorial Text.* Springer-Verlag, London, UK, UK, 184–207. http://dl.acm.org/citation.cfm?id=647699.734159

Philip Wadler. 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*, Jean-Pierre Jouannaud (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 113–128.

Jamie Willis. 2016. Parsec for Scala. https://bitbucket.org/J_mie6/parsec-for-scala/wiki/Home. (2016).