Staged Selective Parser Combinators

JAMIE WILLIS, Imperial College London, United Kingdom
NICOLAS WU, Imperial College London, United Kingdom
MATTHEW PICKERING, University of Bristol, United Kingdom

Parser combinators are a middle ground between the fine control of hand-rolled parsers and the high-level almost grammar-like appearance of parsers created via parser generators. They also promote a cleaner, compositional design for parsers. Historically, however, they cannot match the performance of their counterparts.

This paper describes how to compile parser combinators into parsers of hand-written quality. This is done by leveraging the static information present in the grammar by representing it as a tree. However, in order to exploit this information, it will be necessary to drop support for monadic computation since this generates dynamic structure. Selective functors can help recover lost functionality in the absence of monads, and the parser tree can be partially evaluated with staging. This is implemented in a library called Parsley.

CCS Concepts: • **Software and its engineering** → **Functional languages**; **Parsers**.

Additional Key Words and Phrases: parsers, combinators, meta-programming

ACM Reference Format:
Jamie Willis, Nicolas Wu, and Matthew Pickering. 2020. Staged Selective Parser Combinators. *Proc. ACM Program. Lang.* 4, ICFP, Article 120 (August 2020), 30 pages. https://doi.org/10.1145/3409002

1 INTRODUCTION

For functional programmers, parser combinators provide a clean and flexible approach to constructing parsers. Whilst parser generator libraries are often capable of embedding host language code into the grammar [Gill and Marlow 1995], parser combinators use the host language to construct the grammar itself using higher-order combinators [Hutton 1992; Swierstra 2009; Swierstra and Duponcheel 1996; Wadler 1985]. Unlike hand-written recursive descent parsers [Aho et al. 2006], parser combinators are easier to reason about and maintain. Traditionally, parser combinator libraries are monadic [Hutton and Meijer 1996; Leijen and Meijer 2001][1] which gives them the power to perform context-sensitive parsing, for example, reading non-keyword identifiers:

```
ident :: Parser String
ident = some (oneOf ['a'..'z']) »= (λxs → if isKeyword xs then empty else pure xs)
```

This parser first uses oneOf to read an alphabetical character, and greedily repeats it one or more times with some. Then (»=) (pronounced "bind") is used to give the letters the name xs and proceeds to verify if xs is a keyword or not: if it is then fail, otherwise return xs. Input is consumed without implicit backtracking. This has abstracted a lot of the parsing machinery away from the programmer but has a cost: as well as executing the combinators themselves it is necessary to

[1] Other members of the parsec family such as `megaparsec` and `attoparsec` are also monadic

Authors' addresses: Jamie Willis, Imperial College London, United Kingdom, j.willis19@imperial.ac.uk; Nicolas Wu, Imperial College London, United Kingdom, n.wu@imperial.ac.uk; Matthew Pickering, University of Bristol, United Kingdom, matthew.pickering@bristol.ac.uk.

2475-1421/2020/8-ART120
https://doi.org/10.1145/3409002

execute the functions that build them, often repeatedly. This is the main disadvantage of parser combinators: the job of writing the parser is straightforward but results in a lot of extra interpretive overhead. As we show, the interpretive overhead can be removed at compile time by static analysis. The catch is that the dynamic structure generated by the (≫=) renders static-analysis impossible!

The problem is that functions are generally not inspectable. As such a static analyser treats all functions as black boxes: for some abstract f, the result of p ≫= f is only known once p has been executed at runtime: static analysis is unable to analyse anything that happens past a (≫=).

One way of recovering static analysis is by removing monadicity and falling back on applicative functors [McBride and Paterson 2008], where parsers always have a purely static structure. However, it is no longer possible to write the ident parser above. Instead, there must exist some form of filtering combinator that provides static structure with dynamic runtime characteristics:

```
filteredBy :: Parser a → (a → Bool) → Parser a      (≫=) :: Parser a → (a → Parser b) → Parser b
```

The result of p `filteredBy` f is the value x parsed by p if f x holds true and failure otherwise. This has a static structure: the analyser will know that filteredBy either returns some value, or fails, but not which branch is executed. In contrast, the type of (≫=) allows for unrestricted generation of structure depending on the result of p. The ident parser can now be rewritten:

```
ident = some (oneOf ['a' .. 'z']) `filteredBy` (not · isKeyword)
```

This is both more concise and more easily optimised than the monadic version.

As we will see moving forward, the secret to implementing the filter combinator and other static behaviours with dynamic choices is to restrict our power to *selective functors* [Mokhov et al. 2019]. This will give us access to some control-flow constructs, but still permit the static analysis needed to compile away all of the abstraction and overhead.

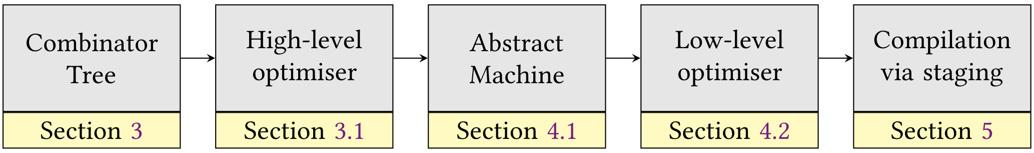

Fig. 1. Parsley's Compilation Pipeline

This paper exploits static analysis and information by compiling combinators to optimised code; this is achieved by a compilation pipeline (Figure 1) that performs optimisation passes before compilation via a final staged interpreter. The combinators are smart-constructors for the nodes of an abstract syntax tree (AST) called the combinator tree: on this representation high-level grammar optimisations and analysis can be performed. Then, the combinator tree is transformed into another AST representing an abstract machine where the structure of the grammar is lost, but the control flow of the parser is readily available: this allows for more low-level optimisations and control-flow analysis. Finally, to leverage the full benefits of static knowledge and eliminate the traditional overhead of parser combinators the evaluation of the machine is staged, producing high-performance code. For instance, here is the code our pipeline produces for the ident parser:

```
ident :: String → Maybe String
ident input =
  let loop (c : cs) dxs finish | isAlpha 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 isKeyword (c : xs) then Nothing else Just (c : xs))
    _                  → Nothing
```

The above code is absent of any of the original combinators and there is no evidence of either of the two AST representations. Instead, there is a tail-recursive function loop that reads alphabetical characters – collecting them in a difference list – provided with a continuation finish that verifies the characters do not form a keyword. Here we use Haskell as our lingua-franca, but our approach generalises to other languages: for instance, the staged meta-programming framework in Dotty has facilitated a similar implementation. The effectiveness of the approach, however, relies on compile-time code generation and the quality of the optimiser processing the generated code.

Contributions. After introducing the relevant background (Section 2), this paper introduces Parsley, a new library that leverages selective functors to permit analysis and static compilation. In addition, our key contributions are that:

- We leverage our AST to perform rich domain-specific optimisations and analysis (Section 3).
- We additionally ensure that the types of the AST during optimisation and compilation are preserved with indexed functor fixpoints (Section 3).
- We use a CPS abstract machine with a fully typed stack for intermediate results (Section 4).
- We remove costly overheads of the pipeline and interpreter with staging whilst still maintaining the above benefits (Section 5).

The paper concludes by providing benchmarks to support our claims (Section 6) and by discussing the effectiveness of the approach as well as future and related work (Section 7).

2 BACKGROUND

Parser combinators provide a way to write top-down recursive parsers without worrying about the machinery of the underlying parser. There is no need to thread state, consider how to consume tokens, or perform any careful book-keeping of non-determinism or backtracking. They are high-level, retaining far more of the structure of a grammar than the hand-written equivalent. Unlike parser generators, they leverage the principle that parsers should be first-class values: this is a powerful technique that allows for writing parsers with minimal duplication and promotes compositionality and abstraction. This section will review basic parser combinators and selective functors in preparation for Section 3, which introduces the combinator tree.

Applicatives. When constructing parsers, recognising strings is a ubiquitous operation; with parser combinators, this can be done with the string combinator:

```
string :: String → Parser String
```

The parser string "ab" will succeed if the input starts with "ab", and fails otherwise. The beauty of combinators is that string itself can be built from smaller building blocks in the host language. To build string, it is necessary to have a way of recognising a single character on its own; more generally, parser combinator libraries expose the satisfy combinator as this primitive:

```
satisfy :: (Char → Bool) → Parser Char
```

This combinator will, when provided a predicate on characters, create a parser that reads a single character matching that predicate and fails otherwise. Using this it is easy to define a combinator for recognising a specific character:

```
char :: Char → Parser Char                 (*>)  :: Applicative f ⇒ f a → f b → f b
char c = satisfy (≡ c) *> pure c           pure :: Applicative f ⇒ a → f a
```

The implementation of char could be written as just a satisfy, but, in this instance, it is beneficial for static analysis to explicitly state what character the parser will result in: the compiler is unable to inspect the predicate given to satisfy in general. To this end, the applicative (*>) sequences two parsers and returns the result of the second, and pure produces a parser which always succeeds with a given value and reads no input. Now the definition for string can be given in terms of char:

```
                                      (⟨∗⟩)      :: Applicative f ⇒ f (a → b) → f a → f b
string :: String → Parser String      traverse  :: Applicative f ⇒ (a → f b) → [a] → f [b]
string = traverse char                traverse f [ ]      = pure [ ]
                                      traverse f (x : xs) = pure (:) ⟨∗⟩ f x ⟨∗⟩ traverse f xs
```

The traverse function is defined in terms of both pure and (⟨∗⟩). The applicative parser pf ⟨∗⟩ px sequences the parsers pf and px and applies their results together. In essence, traverse takes a list of values, transforms them into parsers and then sequences all their results together into a list. For instance, string "ab" expands to pure (:) ⟨∗⟩ char 'a' ⟨∗⟩ (pure (:) ⟨∗⟩ char 'b' ⟨∗⟩ pure [ ]). Transforming a String into a list of parsers that recognise the individual characters and sequencing them together results in a parser that recognises the String itself.

Parsers can be as granular as recognising single characters, and parser combinators compose them together. Consequently, lexing and parsing are not distinguished, since these phases only differ by the tokens they accept. By leveraging lexers as yet another building block, this allows for context-sensitive lexing, which can be useful: the programmer can now be more explicit about what specific lexemes are valid within the rules of a grammar.

Alternatives. Sequentiality alone is not enough to write most parsers: there needs to be an operation for choice. The *alternative* operations provide this behaviour:

```
empty :: Alternative f ⇒ f a
(⟨|⟩)   :: Alternative f ⇒ f a → f a → f a
```

These operations permit failure and branching, respectively.

As an example, matching the regular expression A | B, can be achieved with char 'A' ⟨|⟩ char 'B'. There are many different options for the semantics of (⟨|⟩), in particular in this paper we refer to the deterministic-choice operation with backtracking. This operation is a criteria for being able to parse PEG grammars [Adams and Ağacan 2014; Ford 2004]. The operation has a caveat though: backtracking is only permitted when no input was consumed during a failed branch. This allows for more useful error reporting [Ford 2002], as it restricts error messages to the greediest parse.

In order to facilitate backtracking in instances where greedy parsing is not desirable, many parser combinator libraries provide a try operation, which rolls back consumed input on failure:

```
try :: Parser a → Parser a
```

This example try (string "aa") ⟨|⟩ string "ab" will backtrack after reading an 'a' so that reading the string "ab" can be attempted. Without the try this parser would fail when given "ab".

Parser combinators belong to wider classes of abstractions called applicative and alternative functors that adhere to a set of laws (Figure 2). Read from left-to-right, these laws serve as a guide for optimisation strategies and normal forms that are used in the high-level optimiser (Section 3.1). The concrete deterministic semantics of (⟨|⟩) requires some laws that are not generally applicable to all alternatives (Laws 7 and 8) [Gibbons and Hinze 2011].

Selectives. As previously outlined (Section 1), it is nice to have access to at least some context-sensitivity. Full context-sensitivity is enabled by monads – as they can generate new structure based on previously parsed results – but monads are not suitable for the approach given in this paper. Recently, however, selective functors have been inserted into the hierarchy between applicatives and monads [Mokhov et al. 2019]. In addition to applicatives, Mokhov et al. [2019] introduced a

$$\mathrm{pure}\ \mathrm{id} \mathbin{<*>} p = p \tag{1}$$

$$\mathrm{pure}\ f \mathbin{<*>} \mathrm{pure}\ x = \mathrm{pure}\ (f\ x) \tag{2}$$

$$u \mathbin{<*>} \mathrm{pure}\ x = \mathrm{pure}\ (\lambda f \to f\ x) \mathbin{<*>} u \tag{3}$$

$$u \mathbin{<*>} (v \mathbin{<*>} w) = \mathrm{pure}\ (\cdot) \mathbin{<*>} u \mathbin{<*>} v \mathbin{<*>} w \tag{4}$$

$$(p \mathbin{<|>} q) \mathbin{<|>} r = p \mathbin{<|>} (q \mathbin{<|>} r) \tag{5}$$

$$\mathrm{empty} \mathbin{<|>} p = p \mathbin{<|>} \mathrm{empty} = p \tag{6}$$

$$\mathrm{empty} \mathbin{<*>} p = \mathrm{empty} \tag{7}$$

$$\mathrm{pure}\ x \mathbin{<|>} p = \mathrm{pure}\ x \tag{8}$$

$$\mathrm{branch}\ (\mathrm{pure}\ (\mathrm{Left}\ x))\ p\ q = p \mathbin{<*>} \mathrm{pure}\ x \tag{9}$$

$$\mathrm{branch}\ (\mathrm{pure}\ (\mathrm{Right}\ y))\ p\ q = q \mathbin{<*>} \mathrm{pure}\ y \tag{10}$$

$$\mathrm{branch}\ b\ (\mathrm{pure}\ f)\ (\mathrm{pure}\ g) = \mathrm{pure}\ (\mathrm{either}\ f\ g) \mathbin{<*>} b \tag{11}$$

$$\mathrm{branch}\ (x \mathbin{*>} y)\ p\ q = x \mathbin{*>} \mathrm{branch}\ y\ p\ q \tag{12}$$

$$\mathrm{branch}\ b\ p\ \mathrm{empty} = \mathrm{branch}\ (\mathrm{pure}\ \mathrm{swap} \mathbin{<*>} b)\ \mathrm{empty}\ p \tag{13}$$

$$\begin{aligned}&\mathrm{branch}\ (\mathrm{branch}\ b\ \mathrm{empty}\ (\mathrm{pure}\ f))\ \mathrm{empty}\ k = \mathrm{branch}\ (\mathrm{pure}\ g \mathbin{<*>} b)\ \mathrm{empty}\ k\ \mathbf{where}\\ &\qquad g = \mathrm{either}\ (\mathrm{const}\ (\mathrm{Left}\ ()))\ (\mathrm{either}\ (\mathrm{const}\ (\mathrm{Left}\ ()))\ \mathrm{Right} \cdot f)\end{aligned} \tag{14}$$

Fig. 2. Applicative (1, 2, 3, 4), Alternative (5, 6, 7, 8) and Selective Parser laws (9, 10, 11, 12, 13, 14).

function branch :: Selective f ⇒ f (Either x y) → f (x → a) → f (y → a) → f a[2]. The idea is that given branch b l r, the value returned by b governs which of the two other arguments is executed. If b gives back a Left then l is executed, and if it gives back Right then r is executed. All three of these arguments can be known statically, but the behaviour of the combinator is only known at runtime. This allows for static analysis, but the control flow of these combinators can only be approximated. Using branch, the filter combinator advertised in Section 1 can be implemented:

```
ident = some (oneOf ['a' .. 'z']) `filteredBy` (not · isKeyword)
```

```
filteredBy :: (Selective f, Alternative f) ⇒ f a → (a → Bool) → f a
filteredBy mx f = branch (pure cond <*> mx) empty (pure id)
  where cond x = if f x then Right x else Left ()
```

This implementation first executes mx and maps a function over the result which tests the predicate f injecting the result into an Either () a. The Left branch represents failure, and Right represents success. The left branch then performs empty, the failure operation, and the right branch returns the result from the Right unchanged. Selectives can handle many of the monadic patterns, though naturally not all of them. They can perform anything that involves a localised choice, but they cannot generate new parsers based on the results of old ones. As such, our library is an extension of PEG grammars that also supports a form of context-sensitive decision making. Our library satisfies the selective laws and makes use of additional properties to form optimisations (Figure 2) .

Parser laws. As well as the laws governing applicatives and selectives, there are some laws that are true for parsers in particular (Figure 3). These reference two more primitive parsing operations:

```
look :: Parser a → Parser a
negLook :: Parser () → Parser ()
```

These operations perform positive and negative lookahead respectively: negative lookahead negLook p succeeds when the given parser p fails, consuming no input. For example, it is possible to form the eof parser with negLook (satisfy (const True) *> pure ()), which ensures that by looking forward, there is no character at all. The interactions between look and negLook in particular are interesting: notice the inversion between interactions with look/negLook and pure/empty in Laws 17, 18, 19 and 20; the operations have a correspondence to id and not as boolean functions, with pure and empty as True and False. This continues, as (<|>) corresponds to (∨) and (*>) corresponds to (∧), giving rise to De Morgan's laws instead of distributivity for negative look-ahead's interactions with

[2] The original formulation uses the equivalent select :: Selective f ⇒ f (Either x a) → f (x → a) → f a function instead of branch: we choose branch as it is a more efficient implementation and more intuitively maps to the machine in Section 4.

$$\text{try (satisfy f)} = \text{satisfy f} \tag{15}$$
$$\text{try (negLook p)} = \text{negLook p} \tag{16}$$
$$\text{look empty} = \text{empty} \tag{17}$$
$$\text{look (pure x)} = \text{pure x} \tag{18}$$
$$\text{negLook empty} = \text{pure ()} \tag{19}$$
$$\text{negLook (pure x)} = \text{empty} \tag{20}$$
$$\text{look (look p)} = \text{look p} \tag{21}$$
$$\text{look p} <\!|\!> \text{look q} = \text{look (try p} <\!|\!> \text{q)} \tag{22}$$
$$\text{negLook (negLook p)} = \text{look p} \tag{23}$$
$$\text{look (negLook p)} = \text{negLook (look p)} = \text{negLook p} \tag{24}$$
$$\text{negLook (try p} <\!|\!> \text{q)} = \text{negLook p} *\!> \text{negLook q} \tag{25}$$
$$\text{negLook p} <\!|\!> \text{negLook q} = \text{negLook (look p} *\!> \text{look q)} \tag{26}$$

Fig. 3. Try (15, 16), Lookahead (17, 18, 21, 22), Negation (19, 20, 23, 24) and De Morgan's Laws (25, 26).

(<|>) and (*>). This is related to Kleene-algebras with tests [Kozen 1997], but where commutativity of ($\wedge$) does not hold, as it is sequencing in the Kleene-algebra portion.

3 THE COMBINATOR TREE

The combinators the users interact with are represented as nodes of an abstract syntax tree (AST). The tree preserves, refines and exploits all of the semantic information about the original grammar. The AST is a recursive type formed by taking the fixpoint of a non-recursive *syntactic functor*. This can be traversed in a structured fashion [Hagino 1987; Hinze et al. 2013] to perform optimisations and analysis. The combinator tree is not the best way of expressing the control-flow information of the parser. This structure will be transformed in Section 4 to accommodate a better expression of control-flow information.

Syntax. The transformations on the combinator tree for optimisation, analysis, and compilation are folds over the language syntax [Gibbons and Wu 2014]. Expressing the syntax as fixpoints of functors allows traversals to be defined in terms of algebras, which not only simplify the definition of recursive functions but also allow fusion to occur. Instead of a fixpoint with monomorphic syntax, an indexed fixpoint is leveraged providing polymorphic syntax [McBride 2011]:

```
newtype Fix (syn :: (∗ → ∗) → (∗ → ∗)) (a :: ∗) where In :: syn (Fix syn) a → Fix syn a
```

The Fix syn a structure represents syntax trees with nodes shaped like syn where the tree is indexed with the type a. The syntactic indexed-functor given to Fix for parsers is:

```
newtype Parser a = Parser (Fix ParserF a)
data ParserF (k :: ∗ → ∗) (a :: ∗) where
  Pure     :: a → ParserF k a
  Satisfy  :: (Char → Bool) → ParserF k Char
  Try      :: k a → ParserF k a
  Look     :: k a → ParserF k a
  NegLook  :: k () → ParserF k ()
  (:<*>:)  :: k (a → b) → k a → ParserF k b
  (:*>:)   :: k a → k b → ParserF k b
  (:<*:)   :: k a → k b → ParserF k a
  (:<|>:)  :: k a → k a → ParserF k a
  Empty    :: ParserF k a
  Branch   :: k (Either x y) → k (x → a) → k (y → a) → ParserF k a
```

The type ParserF k a represents a single layer of a parser AST. The type index k represents the type of the children inside each node: when considering the AST usually k = Fix ParserF following from the specialised In :: Parser (Fix ParserF) a → Fix ParserF a. The type index a represents the result type of the parser, for instance Pure 7 :: ParserF k Int for all types k. The role of this type index a, and the reason why an indexed fixpoint is used, is that it ensures that the nodes of the AST can only go together if the combinators are well-typed: this information is preserved, and can be called upon again when performing pattern matches on the nodes.

These constructors mirror the structure of the parser combinators themselves: the operations that the user interacts with are all functions (smart constructors) that serve as wrappers around the combinator tree constructors. It is worth noting that Item :: ParserF k Char is a possible alternative for Satisfy, as satisfy f = item \`filteredBy\` f, where item reads any single character. Satisfy is chosen as it retains more information than Branch would, namely the character is not modified in any way: consider the difference in the guarantees about behaviour between satisfy f and the equivalent branch (pure g <*> item) empty (pure h) for abstract f, g and h. Satisfy also interacts nicely with termination analysis, unlike selectives.

Semantics. The recursion principle cata provides a clean way to work with the Fix ParserF a structure: it removes recursion from any passes over the structure, allowing the function to focus on a single layer at once and guarantees no re-computation of any results. It is defined as follows:

```
imap :: IFunctor f ⇒ (∀i.a i → b i) → f a i → f b i
cata :: IFunctor syn ⇒ (∀i.syn a i → a i) → Fix syn i → a i
cata alg (In x) = alg (imap (cata alg) x)
```

The function cata says that, given a function alg (called an algebra) capable of transforming a layer (the shape of which given by syn) of the structure without having knowledge of its concrete type index i, an entire Fix syn i can be folded into a value of type a i. As an example, here is a function which computes the size of a parser:

```
data Const a k = Const a                unConst :: Const a k → a
size :: Fix ParserF a → Int             unConst (Const x) = x
size = unConst · cata alg where
  alg :: ParserF (Const Int) a → Const Int a
  alg (Pure _)                = Const 1
  alg (Satisfy _)             = Const 1
  alg (Try (Const n))         = Const (n + 1)
  alg (Const n :<*>: Const m) = Const (m + n + 1)
  alg ...
```

The function size is a fold over the combinator tree with the function alg, which non-recursively combines results computed from its children. The Const datatype is used to ensure that the type index of each combinator node is preserved as required by the type of cata.

3.1 Combinator Tree Optimisation

The advantage of having the deep-embedded combinator tree is that it permits inspection of the grammar as well as optimisation. Optimisations performed on the AST are based on the laws governing applicative, alternative and selective parsers (Figures 2 and 3). Most combinators in the library are composite operations formed out of many constructors, however the redundant (:*>:) operation is included to provide symmetry with Pure (Section 4), and to permit useful optimisations requiring more information: as usual, static analysis is not able to distinguish between the composite (*>) and (<*) as they both have the shape pure f <*> p <*> q where f is not inspectable.

Fusion. In particular, two useful forms of optimisation are the applicative and selective fusion laws. Applicative fusion [Delbianco et al. 2012; Kiss et al. 2018] is the idea that any pure computation can be lifted out of an expression and combined in one place, as a pure. Essentially, Law 2 performs the actual fusion, Law 3 moves pure to the front of the computation and Law 4 re-associates operations after a use of Law 3. Applicative fusion is very useful in simplifying common idiomatic parsers. Recall the idiomatic definition of the string combinator presented in Section 2:

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

The unrolled code for string "ab" would be pure (:) ‹∗› char 'a' ‹∗› (pure (:) ‹∗› char 'b' ‹∗› pure [ ]). However, with applicative fusion optimisations in play the actual resulting parser is satisfy (≡ 'a') ∗› satisfy (≡ 'b') ∗› pure "ab", which involves far less redundant computation. This is a common theme with our library, the simpler or more naïve the definition, the better the optimiser performs. This is in contrast to libraries such as megaparsec, where the authors provide specialised combinators to improve performance, but now users must be aware of these combinators and understand how to use them.

In a deep embedding, the implementation of such optimisations is relatively simple. Optimisations are applied by a bottom-up fold over the AST using the following algebra:

```
opt :: ParserF (Fix ParserF) a ⟶ Fix ParserF a
opt (In (Pure f) :‹∗›: In (Pure x)) = In (Pure (f x))
opt (u :‹∗›: In (v :‹∗›: w))         = opt (opt (opt (In (Pure (·)) :‹∗›: u) :‹∗›: v) :‹∗›: w)
opt (u :‹∗›: In (Pure x))            = opt (In (Pure (flip ($)) :‹∗›: u))
 ...
opt p                                = In p
```

The function opt attempts to match each law or derived optimisation in turn and applies the optimised direction if it matches. A law has an optimised direction when it reduces the size of the AST or results in code using cheaper abstract machine instructions. Where the use of a law results in a reduction, opt is re-performed in an attempt to apply new optimisations. This terminates since a normal form will be reached [McBride and Paterson 2008], with finite reductive optimisations taking place in-between: there are finitely many re-associations that can be made before everything is left-associative and all the pures will be on the left, and subsequently reduced.

The parser laws also form a useful set of optimisations: negLook is relatively expensive in terms of instructions (look is cheap), so applications of Laws 23 and 26 reduce the number of instructions generated. Selective fusion is where filteredBy operations are fused into a single branch: Law 13 is used to establish a normal-form and then Law 14 performs fusion.

3.2 Recursion and Let-bindings

In practice, recursive or iterative parsers are very common and useful. However, in order to perform finite traversals over the trees, it is necessary to have finite parsers. Essentially, it is necessary to find recursion points in parsers and replace them with a named hole. Finding recursion points is an instance of a more general problem, however: finding any parsers that were let-bound by the user. The reason finding let-bound parsers is useful is to prevent code explosion caused by inlining repeated parsers in multiple places. Consider the following combinator chainl1, which parses left-associative applications of binary operators:

```
chainl1 :: Parser a ⟶ Parser (a ⟶ a ⟶ a) ⟶ Parser a
chainl1 p op = postfix p (pure flip ‹∗› op ‹∗› p)
```

This combinator will first read a p, then read many ops applied on the right to a p, binding most tightly to the left. Notice how the parser p appears twice in the body of the combinator: if chainl1 is used to parse expressions, then every new precedence level would double the parser size. By identifying that p is the same parser in both places it will be factored out, preventing code blow-up. To put it in perspective, an expression parser for a language like C generated with let-bindings can be thousandths of the size of the expanded version. The approach is as follows:

- Assign a unique identifier to each node in the combinator tree.
- Traverse the tree to identify nodes which are referenced multiple times.
- Factor these nodes into Let nodes within the tree.

Stable Names. One way of identifying let-bindings would be to require the user to name each parser that they wished to bind. This mechanism would be clunky and vulnerable to mistakes. In a language with pass-by-reference, like Dotty, the solution is straightforward: first assume every parser is unique, then if two references are equal then they must come from a common shared source – i.e. a let-binding. In Haskell, Gill [2009] first suggests using GHC's StableNames to do this: a StableName can be requested of any value and if requested again is guaranteed to be the same. Furthermore, each value has a unique StableName: this fulfils the previous criteria for identifying let-bound variables. The following datatype is used to represent the identifiers of parsers:

```
data ParserName = ∀a.ParserName (StableName (Fix ParserF a))
```

The datatype ParserName encapsulates a stable name for any parser: by universally quantifying the type parameter, it states that the type of a parser does not matter, only its name.

Defining Let-bindings. Let-bindings are found by inspecting a control-flow graph constructed by recursive descent. The return type of this traversal is as follows:

```
data LetFindSt = LetFindSt { refs :: HashMap ParserName Int, recs :: HashSet ParserName
                           , before :: HashSet ParserName }
type LetFindCtx = HashSet ParserName
newtype LetFinder a = LetFinder (StateT LetFindSt (ReaderT LetFindCtx IO) ())
```

The ParserNames are generated by IO. The LetFinderCtx provides the names that have already been processed within a local branch of the traversal. The LetFinderSt threaded through the traversal is used: to track how many times a particular name was referenced (refs); keep track of which recursive names have been identified (recs); and ensure that names that have already been processed in a different part of the tree are not processed again (before).

The traversal itself is a bottom-up fold over the AST. The algebra is as follows:

```
findLets :: Fix ParserF a → ParserF LetFinder a → LetFinder a
findLets orig p = do name ← makeParserName orig
                     addReference name
                     ifSeen name ({- then -} addRec name)
                                 ({- else -}  ifNotProcessedBefore name
                                   (do addName name (case p of
                                         pf :<*>: px → do pf; px
                                         p :*>: q    → do p;  q
                                         ...
                                         _           → do return ())
                                       doNotProcessAgain name))
```

The idea is as follows: walk down the combinator tree, every time a node is encountered down a particular branch of the the tree, increment its reference count. If at any point a node is encountered that is already present in the seen set, it is a recursion point and is marked as such. If a parent node has been found in multiple parts of the tree, it should have a larger reference count, but its children should not: they are children of a let-bound node, but are possibly not themselves let-bound. As a result, the traversal only processes the children of nodes once.

After the traversal, all nodes with a single reference are discarded: having one reference is either an indication that the name is not let-bound or that it should be trivially inlined. What remains is a set of recursive parsers and a set of let-bound parsers.

Generating bindings. Once all the let-bindings have been identified, a second traversal is used to refactor the AST to include a new Let :: Bool → MuVar a → ParserF k a constructor, where MuVar a represents a concrete name and its type. The Bool is used to indicate recursive bindings. This traversal is supported by a second monad:

```
newtype LetInserter a =
  LetInserter (FreshT Word
                (StateT (HashMap ParserName Word, DMap MuVar (Fix ParserF)) IO)
                (Fix ParserF a))
```

Here, FreshT is a monad which supports the generation of fresh variable names of type Word. The State stores a map from ParserNames to their generated identifier and a dependent map that maps a MuVar to its associated combinator tree: this preserves type information – a MuVar a always maps to a Fix ParserF a, for any a.

```
letInsert orig p = do name ← makeParserName orig
                      let bound = HashSet.member name lets
                      let recu = HashSet.member name recs
                      vs ← gets fst
                      if bound ∨ recu then case HashMap.lookup name vs of
                        Just v    → let μ = MuVar v in return (In (Let recu μ))
                        Nothing → do v ← newVar
                                     modify (first (HashMap.insert name v))
                                     q ← postprocess p
                                     modify (second (DMap.insert (MuVar v) q))
                                     return (In (Let recu (MuVar v)))
                      else do postprocess p
```

The first task is to acquire the name for the original AST node and establish whether or not this name is let-bound or recursive. If it is, then, if a Word has already been generated for this name return a new Let node, otherwise ask for a fresh variable with newVar and insert into the relevant maps. The postprocess algebra inserts lets into subtrees and optimises. After this phase, the IO is safe to eliminate with unsafePerformIO, since StableNames are no longer required.

Now that this phase has been ran on the combinator tree, it is guaranteed to be finite. This permits the optimisation and analysis phases to run over this tree and it is guaranteed that this will terminate. It also allows for the generation of a finite machine (Section 4) which is crucial for staging away overheads in Section 5.

3.3 Analysis

Aside from optimisation and normalisation, it is possible to perform various forms of analysis on a parser. These are enabled by the deep embedding of the combinators and, by using compile-time staging (Section 5), these analysis phases do not impact the runtime for the user. This section will describe two specific analyses: consumption analysis and termination analysis, though many other analyses are possible, including backtracking analysis and error message analysis.

3.3.1 Consumption Analysis. Programs written using parser combinators usually incorporate lexing and parsing together: the resulting parsers consume long strings of characters of a known length. Instead of requiring a runtime length check for each character, it would be better to collect all these length checks together into a single, larger, check.

Consumption analysis is a great example of how every part of the pipeline from combinators down to Haskell code has a part to play, so it will be revisited later in the paper as more structure becomes available. The first step for this analysis is to identify the points in the grammar at which cuts take place – called cut-points. This information allows later stages of the analysis to determine at which points length-checks cannot commute to earlier parts of the grammar: no length-check savings can cross a cut, as this would change the semantics of the failure. To see why this is the case, consider the parser `string "aa" <|> string "ab"`: if the length checks were factored out, then the parser first checks if there are at least 2 characters in the input. Otherwise, it backtracks to another part of the wider grammar. However, if the length checks are left in, and an `'a'` is read, then if another `'a'` is not read this parser is no longer able to backtrack (the semantics state that backtracking after consuming input is illegal) – this is different behaviour.

Annotating the cut-points in the grammar is performed by a single fold through the combinator tree. The information propagated is the cut-fulfilling properties of sub-branches and whether or not the parent node in the tree demanded a cut (and the sibling branches have not yet satisfied this demand). When a branch of a tree requires a cut from a parent and it satisfies it, this is marked as a meta-node in the tree and its siblings are not required to handle any cuts themselves. Only a Satisfy node may fulfil a demand for a cut. A non-indexed excerpt from the algebra is presented below:

```
cutAlg :: ParserF (Bool → (Fix ParserF, Bool)) → Bool → (Fix ParserF, Bool)
cutAlg (Pure x)      _     = (In (Pure x),              False)
cutAlg (Satisfy f) True    = (mkCut (In (Satisfy f)), True)
cutAlg (Satisfy f) False   = (In (Satisfy f),           True)
cutAlg (l :*>: r)    cut   = let (l′, handled)  = l cut
                                 (r′, handled′) = r (cut ∧ not handled)
                             in (In (l′ :*>: r′), handled ∨ handled′)
cutAlg (p :<|>: q)   cut   =                         -- This is the case which doesn’t force a cut
  let (q′, handled) = q cut                          -- But it might have to handle a cut from a parent
  in (In (fst (p False)) :<|>: q′, handled)         -- q will handle the cut if p fails
```

In the Pure case, a cut is never satisfied, but in the Satisfy case, a cut is always satisfied and a cut meta-node is generated if necessary. In the (:*>:) case (and similar for other sequencing operations) then the right-hand side only needs to handle a cut if the left did not. In the (:<|>:) case, which does not backtrack, the first branch is not required to handle cuts.

The next task is to collect up the amount of input that is consumed across a continuous path through the grammar. This is difficult to perform on the combinators, however, because nodes usually associate to the left, and so tracking control flow is very involved. As it will turn out, the correct domain to find this information will be in the machine (Section 4.3).

3.3.2 Termination Analysis. An easy mistake to make when writing parsers with a parser combinator library is to accidentally write a *left-recursive* parser. This is a parser that consumes no input before recursing and trying the same grammar rule again. Another example of non-termination is a parser which iterates over only pure computations: this parser will never fail, so the iteration never ends. Termination analysis aims to catch instances of this problem and report them back to the programmer. The basic idea is to use abstract interpretation to execute a parser, determining whether or not there exists a branch which self-recurses without having consumed any input at all. This can also be done as a bottom-up traversal on the combinator tree.

The problem with this analysis lies in the fundamental incomputability of the halting problem, of which this is a variant. The formal description of the analysis is to guarantee that a parser consisting of only total functions will terminate when given finite input. Notably, termination of purely applicative parsers is decidable: the control flow through the parser is completely statically determinable. For selective parsers, however, termination is non-decidable; consider the following:

```
bad :: Parser ()
bad = branch (char 'a' $> Left () <|> pure (Right ())) (const <$> bad) (const <$> bad)
```

Recall that any functions or values in a parser are treated as black boxes, so the Parsley compiler is unable to distinguish between the Left and Right values in the condition. The human can see that bad does not loop in the Left branch, as an 'a' must have been read, but does loop in the Right branch, because no character was read. However, the machine cannot tell whether the input was consumed for the Left or the Right branch. In this instance it will over-approximate suggesting that – since at least one of the recursive branches consumes no input, and the condition may not consume input – the parser is left-recursive. The analysis can suggest false positives: consider removing the recursion on the Right branch of the example, creating a non-left-recursive grammar. The parser would still be erroneously reported as left-recursive and, as such, approximation yields warnings instead of errors. As such, while this analysis is necessarily imperfect, it is a nice example of how static analysis with selective functors works [Mokhov et al. 2019], and the trade-off is whether selective instances of this problem are just ignored, or the compiler is overly cautious in reporting them. Regardless of which approximation is chosen, this analysis will still provide accurate feedback for grammars that are purely context-free.

4 COMPILATION TO ABSTRACT MACHINE

The representation of parsers as a deep embedding of the combinators allows the compiler to perform complex analysis and law-based optimisation of the trees. At the same time, however, it has some disadvantages: firstly, the structure of the grammar is available, but the control flow of the parser is not explicit; and, secondly, the combinators are more rigid building blocks that lead to redundancy in the representation.

The abstract machine introduced in this section addresses these concerns: by converting the combinators into a continuation-passing style (CPS) instruction set [Kennedy 2007], the sequence of operations required to perform a parser becomes explicit. To translate scoped operations such as (<|>), try, and look, multiple instructions are used but, as a result, it becomes very difficult to recover the original scoping of the parser again. Fortunately, all the analysis that relies on scope and the grammar will already have been performed on the combinator tree and is readily available. These new instructions are more general and expressive than the combinators they represent, allowing the machine to represent a wider variety of high-level operations with less instructions.

4.1 The Machine

Since our library has ousted monadic combinators, context-sensitive grammars are no longer a concern: as such, a natural abstraction for the parsers is a push-down automaton – a machine with a single stack with which to store tokens and results. The abstract machine presented here is similar to a push-down automaton: it has a operand stack storing the results of the intermediate parsers, but also has a mechanism to handle failures and backtracking as well as recursion.

When designing the combinator tree, a key design point was to preserve the types of the combinators within the nodes of the tree. This provided a light-weight assurance that the optimisations and transformations preserved the original type of the parser. This property must be carried forward into the abstract machine as well. In this case, the CPS transformation of the combinators naturally reveals the lifetime of a value produced by a parser: how long it must be kept around before being used to complete an operation. In terms of a stack machine, the lifetime of a value is how long it remains on the stack, and the machine better preserve the types of the values on the stack as the computation progresses [Benton 2005].

In addition to the stack however, the type of calls to recursive parsers must also be preserved within the machine: when a parser recurses, it will be expected to return a result of a certain type, and it must be the case that this value will make it onto the stack when the parser returns. Enforcing this relationship will be examined more later.

Applicatives. Like the combinator tree, the machine is a syntactic functor, except it has a different kind with more indices (for clarity, Fix is overloaded). Here is the definition of a portion of the applicative instruction for the machine functor M:

```
data M (k :: [∗] → ∗ → ∗ → ∗) (xs :: [∗]) (r :: ∗) (a :: ∗) where
  Halt  :: M k [a] Void a
  Push :: x → k (x : xs) r a → M k xs r a
  Pop   :: k xs r a → M k (x : xs) r a
```

The type M has four type indices: k is the same as in the combinator tree, it represents the shape of the values contained within each node (often Fix M); xs is a type-level list representing the types of the values required on the stack upon entry to the given instruction; r represents what type the machine returns to the caller in the case that this is a recursive call; type a is the final "goal" of the machine, in other words it directly corresponds to the type of the top-level parser that was compiled to generate this machine.

The instructions Push and Pop form two of the instructions needed for applicative combinators. They demonstrate how interactions with the stack are preserved at the type-level: Push expects the machine that is executed after it to require an x on the front of some stack xs, so it is itself a machine which just requires xs and promises to put the value of type x onto the stack; Pop is similar in that it makes a promise that it will somehow remove the first value from the stack to make it compatible for the next machine. The Halt instruction is the final instruction executed by the machine and demands that the stack must contain precisely the goal value of type a; in addition it enforces that there is no more recursion to perform by setting the return type r to be Void.

Using the type index xs ensures that instructions which operate on intermediate values can only be put together in the right way. For example, (push () · push ()) halt[3] will not compile, since halt has type Fix M (b : [ ]) Void b and push () · push () has type Fix M (() : () : xs) r a → Fix M xs r a, but it is not possible to unify the types () : () : xs and b : [ ]. This provides a sanity check that the translation from high-level combinators to low level stack machine at least preserves the intended user-operations and that the implementation of these instructions affect the stack in the advertised way. This guarantee is provided by the polymorphic types in the instruction: without any concrete knowledge about the types, it is not possible for a malevolent code-generator to replace any of the user's functions with its own. It is not perfect however, the malevolent code generator could

[3] From here on, we reference instructions by a smart constructor, as an example push x = In · Push x

replace any part of the user's code with the failure instruction Fail. Instead of being a concrete proof, this mechanism serves as a light-weight verification [Morrisett et al. 2002].

Compilation. Before introducing any more instructions, the compile function that translates the combinator tree to the machine will be introduced, highlighting the translation from pure, (:∗›:) and (:‹∗:) into Push, Pop and Halt. As more instructions are presented, the relevant translation from compile will also be given, solidifying the relationships between the two domains:

```
compile :: Fix ParserF a → Fix M [ ] Void a
compile = cata compAlg halt

type CodeGen a x = ∀xs r.Fix M (x : xs) r a → Fix M xs r a

compAlg :: ParserF (CodeGen a) x → Fix M (x : xs) r a → Fix M xs r a
compAlg (Pure x)  = push x
compAlg (p :∗›: q) = p · pop · q
compAlg (p :‹∗: q) = p · q · pop
```

First, notice the type of the algebra: given a parser immediately producing a value of type x in some wider context producing a result of type a, provide a machine that will consume only that x and create a machine which requires the rest of some stack xs to achieve the goal a. This encodes an important property of the machine used as a lemma in proofs of correctness: a successful parser should push exactly one item to the stack.

The execution order of the parsers is made clear here as the machines are read left to right: p ∗› q should be the same as performing p, popping its single result from the stack and then performing q; and p ‹∗ q should perform p then q and then pop the top of the stack which is q's result. The pure combinator should just push a value to the stack and continue. The applicative instructions are completed by the Lift2 and Swap instructions:

```
data M (k :: [∗] → ∗ → ∗ → ∗) (xs :: [∗]) (r :: ∗) (a :: ∗) where
  ...
  Lift2  :: (x → y → z) → k (z : xs) r a → M k (y : x : xs) r a
  Swap :: k (x : y : xs) r a → M k (y : x : xs) r a
```

$$\underline{\mathrm{app}} = \mathrm{lift2\ id}$$

$$\mathrm{compAlg\ (pf :\langle{*}\rangle{:}\ px) = pf \cdot px \cdot}\ \underline{\mathrm{app}}$$

The Lift2 instruction is used to reduce the top two elements of the stack with a provided function. It can be used to implement (:‹∗›:) with the id :: (a → b) → a → b function. It is preferred instead of an App instruction for its versatility: app = lift2 id, but lift2 f = push f · swap · app · app. The reason that (:‹∗›:) was chosen for the combinator tree, however, is because the laws are more easily expressed in terms of (‹∗›) as opposed to liftA2. The Swap instruction is a simple stack exchange operation, which will be used for compiling other operations.

Selectives. In addition to the applicative instructions, Case is used to implement branch:

```
data M (k :: [∗] → ∗ → ∗ → ∗) (xs :: [∗]) (r :: ∗) (a :: ∗) where
  ...
  Case :: k (x : xs) r a → k (y : xs) r a → M k (Either x y : xs) r a
```

$$\mathrm{compAlg\ (Branch\ b\ l\ r)} = \lambda \mathrm{k} \rightarrow \mathrm{b\ (case\ (l\ (swap\ (}\underline{\mathrm{app}}\ \mathrm{k)))\ (r\ (swap\ (}\underline{\mathrm{app}}\ \mathrm{k))))}$$

Given two machines, each accepting one half of a co-product, Case forms a machine that pattern matches on the co-product to choose which branch to take. The translation shows that branch b l r first performs b, then uses Case to arbitrate between the left l and right r branches applying them to the corresponding value from the Either x y on the stack. The continuation machine k is given to both branches. This will be compiled to a Haskell **case** expression.

Alternatives. There are three instructions associated with the scoped operations:

```
data M (k :: [∗] → ∗ → ∗ → ∗) (xs :: [∗]) (r :: ∗) (a :: ∗) where
  ...
  Fail   :: M k xs r a
  Catch  :: k xs r a → k (String : xs) r a → M k xs r a
  Commit :: k xs r a → M k xs r a

handle :: (Fix M xs r a → Fix M (x : xs) r a) → Fix M (String : xs) r a
       → Fix M (x : xs) r a → Fix M xs r a
handle p h k = catch (p (commit k)) h

compAlg (p :<|>: q) = λk → handle p (parsecHandle (q k)) k
compAlg Empty       = const fail
```

Failure and backtracking is managed by exception style *handlers*. These are managed on another stack within the machine (Section 5). These three instructions interact with the handlers: Fail corresponds directly to empty, it will execute the next handler; Catch m h will push h to the handler stack and will attempt m within this context; and Commit is used to remove handlers from the stack, closing the scope of the handler. Both Catch and Commit interact with each other using the handle function: this encloses the first argument p in a failure catching scope handled by the second argument h, continuing with the third argument k as a continuation. The contents of the handler parsecHandle is discussed in Section 4.1.1.

Primitive instructions. The instructions corresponding to satisfy, try, look and negLook are outlined here, as well as those required to implement recursive parsers.

```
data M k (xs :: [∗]) r a where
  ...
  Sat  :: (Char → Bool) → k (Char : xs) r a → M k xs r a
  Tell :: k (String : xs) r a → M k xs r a
  Seek :: k xs r a → M (String : xs) r a
  Ret  :: M k [r] r a
  Call :: MuVar x → k (x : xs) r a → M k xs r a

compAlg (Satisfy p) = sat p
compAlg (Try p)     = handle p (seek fail)
compAlg (Look p)    = tell · p · swap · seek
compAlg (Let _ μ)   = call μ
```

The Sat instruction maps to Satisfy directly and is the only source of conditional failure within the instruction set. The instructions Tell and Seek work together to realise look: Tell pushes the current input onto the stack, then p is performed (and its result placed second onto the stack) and then Seek restores the input back to its original state, leaving the result from p. Since these do not rely on failure handlers, this operation is virtually free. The try operation is implemented in terms of already existing instructions by resetting the input back to the start of the scope and failing again.

The final two instructions are Ret and Call, a pair used to implement recursion within the machine. To prevent an infinite machine, let-bound parsers are compiled (using Ret instead of

Halt) and stored in a type-indexed map indexed on the return type parameter r. The instructions Ret :: M k [r] r a and Halt :: M k [a] Void a are similar except Ret denotes there is yet more to do before the goal of type a is reached (and instead the value of type r should be passed back to the caller). Call μ k fetches the corresponding machine for μ from the map and executes it, setting k as the return continuation: this is where the interaction with r is made very explicit, the machine k must expect a value on the stack of the same return type x as the machine μ and the current return type is left untouched.

Negative Lookahead. It is interesting to look more closely at the compilation of negLook. The combinator is compiled using the same components as look, (<|>), and try but without care it is possible to implement incorrectly. Consider the following plausible definition of negLook:

negLook p = try (look p *> empty) <|> pure ()

This definition does not appear problematic at first glance: negative look-ahead tries to look-ahead for p, and if it succeeds fails and returns () otherwise. However, this has a subtle bug: the empty inside the (<|>) will actually be handled by the (<|>) and, as such, this parser always succeeds. Traditionally, in monadic parser combinator libraries such as parsec, join :: Parser (Parser a) → Parser a is used to fix this problem:

negLookM p = join (try (look p *> pure empty) <|> pure (pure ()))

The monadic definition works because the empty and pure () are wrapped up inside parsers themselves. If look p fails then the fragment inside the join will return pure (), and if it succeeds empty is returned. After the scope of the (<|>) has been exited, the join combinator executes the resulting parser. However, join is monadic and, as such, is not expressible in our language. The original problem was caused by the scope imposed by the combinators being too rigid, and join loosens this to allow the floating of parsers through a scope boundary. However, as the instructions work in pairs to delimit scope they are more flexible than the combinators, illustrated by the compileAlg for NegLook:

compAlg (NegLook p) = λk → handle (tell · p · pop · seek) (seek (push () k)) fail

The first argument to handle performs the look p, but the Fail instruction has been placed as the continuation for the handle instead, after the handler has been discarded. This is a major advantage of compiling to the abstract machine as opposed the the combinator tree: there is much finer grained control over the control flow of the parser.

4.1.1 Handlers. Throughout the previous section, there have been references to failure handlers, these are regular machines defined in terms of the instructions used for the combinators. Handlers will perform some combination of resetting state, propagating failure, or recovery.

$\underline{\text{fmap}}$ f = push f · swap · $\underline{\text{app}}$
$\underline{\text{if}}$ t e = $\underline{\text{fmap}}$ (λb → **if** b **then** Left () **else** Right ()) (case (pop t) (pop e))

parsecHandle :: Fix M xs r a → Fix M (String : xs) r a
parsecHandle k = tell (lift2 (≡) ($\underline{\text{if}}$ k fail))

The $\underline{\text{if}}$ smart constructor builds a machine which requires a Bool on the stack, if that value is true it performs the first machine else it performs the second. Using this, parsecHandle can be defined obeying parsec semantics: if no input has been read since the scope was entered, then continue with the provided machine, else fail. The fact that the handlers themselves can be encoded with the instruction set designed for the core combinators highlights another strength of the approach – especially given that only one of the look instructions are needed to implement each handler.

4.2 Machine Optimisation

After the translation to the abstract machine, more optimisation opportunities are evident; some of transformations that are possible on the abstract machine are now discussed [Appel 2007].

Join points. It is worth noting that this approach to compilation suffers from a code explosion problem. Observe that in the compAlg case for (:<|>:), the continuation k is referenced twice: this means the generated code for this continuation will be generated twice. This is far from ideal, so the machine also includes join points (ϕ-nodes) which can be generated by branching constructs like Case and Catch as well as executed with the Join ϕ instruction [Maurer et al. 2017]. In terms of the machine implementation (Section 5), this system is omitted for clarity: it obscures while adding no extra insights. As another optimisation, a ϕ-node of the form ϕ_1 = Join ϕ_2 can be elided and all occurrences of ϕ_1 can be replaced by ϕ_2. Furthermore, ϕ-nodes of the form ϕ = Ret or ϕ = Halt can also be inlined, since these instructions have a small code footprint.

Tail call optimisation (TCO). Tail calls are recursive calls that are executed at the very end of a function. As they are the last thing to be performed, it is not necessary to save the state of the function to perform them, as that state will be unused and discarded upon return. Traditionally, some compilers are able to optimise tail calls by transforming them into jumps, making the function into a raw loop. It is possible to do this as well in this machine: after introducing another instruction Jump μ, the alg case for Let can be adjusted as follows:

```
compAlg (Let _ μ) (In Ret) = jump μ
compAlg (Let _ μ) k        = call μ k
```

Furthermore, as the error handling stack is rolled back on a Ret instruction (since scopes are exited), Commit instructions before a Ret are also removed, providing more opportunities for Jump instructions. Using Jump is especially important for languages without sophisticated TCO.

Deep inspection. The composition of the handlers for (:<|>:) and Try form the handler for negLook. Recognising this pattern and generating specialised code with the improved handler can help reduce the size of the code that the compiler will have to optimise – though GHC often generates equivalent code this method is more robust and portable. However, these patterns are more than one layer of AST nodes deep: a key property of cata is that it does not give access to sub-structures at each level of recursion. As a result, to perform these sort of peephole optimisations, it is necessary to change to a different recursion scheme. In the spirit of structured recursion, it should not be possible to retraverse the structure, instead the past results should be inspectable. This allows the peephole to match a nested pattern, but retain the original results. This fold is called a *histomorphism* [Hinze and Wu 2013; Uustalu and Vene 1999], and the non-indexed definition is given below:

```
data Memo f a = a ◂ f (Memo f a)
extract :: Memo f a → a
extract (x ◂ _) = x

histo :: Functor f ⇒ (a → b) → (f (Memo f a) → a) → Fix f → a
histo gen alg = extract · cata (λx → alg x ◂ x)
```

This is a cata where the intermediate result of every step is stored in a memoising structure alongside the subtree itself: the intermediate Memo f a structure preserves the history of the fold. The peephole algebra will search for patterns that can be compiled with fewer instructions:

```
peephole :: ParserF (Memo ParserF (CodeGen a)) x → Maybe (Fix M (x : xs) r a → Fix M xs r a)
  -- identify fmap, and use Fmap, improving locality
```

```
peephole ((_ ◃ Pure f) :<*>: (p ◃ _)) = Just (p · fmap f)
  -- identify liftA2, and use Lift2 directly
peephole ((_ ◃ ((_ ◃ Pure f) :<*>: (p ◃ _))) :<*>: (q ◃ _)) = Just (p · q · lift2 f)
  -- identify try p <|> q, and use more specialised handler
peephole (_ ◃ Try (p ◃ _)) :<|>: (q ◃ _) = Just (λk → handle p (seek (q k)) k)
peephole _ = Nothing
```

This algebra looks back into the history to see if what has already been transformed matches common patterns like fmap or liftA2. It then applies the optimised generation and returns it in Just to signify a match was found. If it returns Nothing then the Memo is removed and passed onto the original compAlg by creating the algebra λx → fromMaybe (compAlg (imap extract x)) (peephole x).

4.3 Consumption Analysis Revisited

In Section 3.3.1, an analysis pass was presented which aimed to remove as many length checks as possible in the generated parser. To do this, the parser was augmented with meta-data about where cuts occur within a grammar. This was computed to ensure the optimisation preserved the cut semantics. The next stage of this analysis is to compute how much input is consumed along a path through the grammar. This is difficult to perform on the combinator AST for two reasons: the associativity of the nodes in the tree are not representative of the data flow through those trees; and the join points within the grammar (that occur after a choice) are not readily available – indeed, this structure does not even exist conceptually at that level. However, the conversion to CPS form machine instructions reveals hidden structure in the grammar: simply walking through the machine is enough to determine the control flow, and join points of the grammar are made very explicit with the ϕ-nodes. Conversely, the structure that enabled the annotation of the cuts has been lost in this conversion as scopes have been flattened. Computing the number of tokens consumed makes use of the following algebra[4]:

```
inputConsumed :: Fix M → Int
inputConsumed = cata alg where alg :: M Int → Int
                               alg Halt            = 0
                               alg (Push _ k)      = k
                               alg (Sat _ k)       = k + 1
                               alg (Catch p q)     = min p q
                               alg (Call _ _)      = 0
                               alg (MkJoin ϕ b k)  = b + k
                               alg (Join ϕ)        = 0
                                ...
```

The logic for this function is very simple: The Sat instruction is the only one to increase the count because Sat is the only action which consumes input. For Call, the idea is that it is unknown whether a recursion will satisfy the cut requirements, so it does not consume any input. When the parser branches (as in Catch), as much of the input should be factored as possible, which will be the minimum of both branches. When join points are generated, the binding's (b) consumed input is added on to the machine where that binding can occur (k). It will be the responsibility of the code generator to ensure the input consumed on the branches is correctly handled.

The compile function can now be augmented with this new information to correctly insert meta-instructions that perform the actual length checks, given the information about cuts generated by

[4] The type indices xs, r, and a have been removed for clarity, only k remains

the last phase. In the end, the meta-instructions inserted by the code generation will be compiled out during staging, and length checks will be sprinkled sparingly throughout the generated code, according to the instructions. The consumption analysis is a nice example of data being computed and refined flowing through all 3 stages of the library's compilation. Doing all the work in only one of these stages is difficult, but together it is easy.

5 STAGED INTERPRETATION

The abstract machine has been introduced syntactically (Section 4), but its implementation has not been addressed. This section: briefly outlines how the machine is executed; describes the problems with the approach; and gives their solutions using staging, with incremental improvements.

5.1 Interpreting a Machine

A parser of type Parser a by this point has been transformed into a value of type Fix M [ ] Void a. This machine must now be interpreted to produce the final denotation of a parser, function of type String → Maybe a.

```
type Eval xs r a = Γ xs r a → Maybe a
eval :: Fix M [ ] Void a → (String → Maybe a)
eval m = λinput → cata alg m (Γ input HNil [ ] (error "Empty call stack"))
  where alg :: M Eval xs r a → Eval xs r a
        alg Halt = evalHalt
        alg (Push x k) = evalPush x k
        alg ...
```

The eval function folds the machine into a function of type Γ [ ] Void a → Maybe a, where Γ represents the state of the machine parameterised by the type indices of the instructions at each step. The interpreter runs by providing the input to an otherwise empty state Γ and feeding this to folded machine. The type Γ is defined as follows:

```
data Γ xs r a = Γ {input :: String,                 ops :: HList xs
                  , hs :: [String → Maybe a], retCont :: r → String → Maybe a}
data HList (xs :: [∗]) where
  HNil   :: HList [ ]
  HCons :: x → HList xs → HList (x : xs)
```

The state consists of four parts: the parser's input (input); the operand stack (ops), a heterogeneous list (HList) indexed by the types of the values on the stack; a failure handler stack (hs), which is a list of functions that, when given input, may handle failure and continue the parser – handlers are machines which have been partially evaluated; the return continuation (retCont) is the function that is called when a parser returns from a recursive call – it is also a partially evaluated machine.

The definition of the evaluation functions for each instruction mirror the types of the instructions themselves. A selection of instructions covering a range of concepts within the machine are provided below to serve as example:

```
evalHalt :: (Γ [a] Void a → Maybe a)
evalHalt = λγ → let HCons x _ = ops γ in Just x
```

The Halt instruction is the successful terminal operation of the machine and produces a value of type Maybe a by returning the top of the stack ops wrapped in a Just.

```
evalLift2 :: (x → y → z) → Eval (z : xs) r a → (Γ (y : x : xs) r a → Maybe a)
evalLift2 f k = λγ → let HCons y (HCons x xs) = ops γ in k (γ {ops = HCons (f x y) xs})
```

The Lift2 instruction extracts the top two elements of the stack and uses its given function f to create a value of type z required on the stack for the partially evaluated continuation machine k.

```
evalFail :: (Γ xs r a → Maybe a)
evalFail = λγ → case hs γ of h : _ → h (input γ)
                             [ ]   → Nothing
```

The Fail instruction will check the handler stack hs to see if there is a handler present that can attempt to recover from this failure. In the event that there is some handler h, it is executed by providing the input from the point that the machine failed. If there are no handlers then this is a terminal operation which returns Nothing :: Maybe a.

```
evalSat :: (Char → Bool) → Eval (Char : xs) r a → (Γ xs r a → Maybe a)
evalSat f k = λγ → case input γ of
  c : cs | f c → k (γ {input = cs, ops = HCons c (ops γ)})
  _            → evalFail γ
```

The Sat instruction checks to see if the first character c of the input matches its given predicate f, and, if so, pushes it onto the stack and provides it to the continuation machine k. If the character did not match, or the input is empty, it will defer to the Fail instruction.

This is a working implementation of the evaluator, but it incurs heavy interpretive overhead. Without any further action parsers incur the cost of translation of the combinators through the different intermediate representations and optimisation phases in addition to paying the cost of interpretation: folding the machine; building the next state Γ between each instruction; and pushing and popping to the stack all contribute heavily to the runtime. Any benefits from the earlier optimisations will be dwarfed by this significant penalty. To overcome this issue, the interpreter should perform all static analysis of the grammar at compilation time and leave only the execution of the parser on an unknown input at runtime. This can be done using staging.

5.2 Background: Staging

A key part of this work is to use principled meta-programming techniques [Rompf and Odersky 2010; Sheard and Jones 2002; Taha and Sheard 1997] to not only entirely eliminate the overhead of the pipeline and interpretation but produce code that appears almost handwritten, creating a bridge between abstraction and practicality. In particular, we make use of compile-time staged meta-programming via Typed Template Haskell: a form of partial-evaluation that can be easily reasoned about via explicit annotations. Staging allows the programmer to identify parts of their program with *static* structure – code that does not depend on *dynamic* runtime values – and remove the overhead of performing this computation in advance, possibly at compile time.

In the world of staging, a program fragment is a first-class value given type Code a for a piece of code producing a value of type a. It is possible to manipulate this code using ordinary functions, but importantly code can be synthesised and combined using the primitive operations *quoting* and *splicing*. Roughly speaking, if x :: a then ⟦x⟧ :: Code a and if qx :: Code a then $(qx) :: a.

The traditional example of staged meta-programming is the power function where the function power n x calculates x^n, for a positive n. Here is its definition, alongside the staged version power′:

```
power :: Nat → (Int → Int)             power′ :: Nat → Code (Int → Int)
power 0 = λx → 1                       power′ 0 = ⟦λx → 1⟧
power n = λx → x ∗ power (n − 1) x     power′ n = ⟦λx → x ∗ $(power′ (n − 1)) x⟧
```

The function is structurally inductive on the exponent and this structure can be readily exploited by staging: usually, the exponent n is known in advance but the base x is not. This means that the exponent is *static* information and the base is *dynamic* information and, as such, the definition can be massaged to form a partial evaluator.

The function power′ n returns code that represents a function which raises its argument to a specific nth power. When code is spliced in, it has the effect of inlining it, meaning that the non-zero case returns a function which multiplies x by the unrolled function for n − 1. For instance:

```
power5 = $(power′ 5) = $(⟦λx → x ∗ x ∗ x ∗ x ∗ x ∗ 1⟧) = λx → x ∗ x ∗ x ∗ x ∗ x ∗ 1
```

In essence, power5 is turned into a regular function by a top-level splice, partially evaluating it at compile time, so that at run time it is ready to use its argument in the unrolled definition.

5.3 Staging the Interpreter

It is important to note that the ability to stage the final evaluator was a primary concern design of the combinator language. The use of selective functors over monads ensures that the *structure* of the parser is statically known and this permits the definition of an abstract machine that reflects this static structure as well as the control flow explicitly – this is well suited to staging. In the rest of this section, structure that is present at run-time is highlighted using a grey box, and everything else is information known at compile time that will be removed by the process of staging the evaluator.

The necessary binding-time analysis has been performed and so, given that the structure of the parser is *statically* known but the input is only known *dynamically*, the eval function can be staged in the usual fashion. As such, by a gentle massaging of the types the eval function can be transformed from interpreter to compiler:

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

eval′ :: Fix M xs Void a → Code (String → Maybe a)
```

The folding of the AST now produces *code* that implements a function from machine state Γ to a result of type Maybe a. The definition changes appropriately:

```
eval′ m = ⟦λinput → $(cata alg′ m) (Γ input HNil [ ] noret)⟧ where ...

evalHalt′ :: Code (Γ [a] Void a → Maybe a)
evalHalt′ = ⟦λγ → let HCons x _ = ops γ in Just x⟧

evalLift2′ :: Code (x → y → z) → Eval′ (z : xs) r a → Code (Γ (y : x : xs) r a → Maybe a)
evalLift2′ qf qk = ⟦λγ → let HCons y (HCons x xs) = ops γ
                          in $(qk) (γ {ops = HCons $(qf) x y xs})⟧
```

The instructions remain mostly unchanged: they now return Haskell code instead of performing an operation. As illustrated by evalLift2′, the code produced by the *partial* evaluation of the continuation machine k must be spliced into the code that should be generated for Lift2 f k as a whole. Recall that join-points were offered as a means to prevent code explosion in abstract machine generation: this allows for multiple splices of the same code to not cause duplication by generating a regular let-binding. Roughly, ⟦$(x) ... $(x)⟧ would become ⟦ **let** y = $(x) **in** y ... y⟧.

Removing state. The only code that the end user runs will be that found within the quotations. At this point, all the interpretive overhead of folding and executing the machine instruction has been eradicated. However, the machine state Γ still appears within the quotes in the above implementation and, as such, will exist at runtime. The structure of Γ is statically known to be a record of four fields, however; instead, a function with four arguments could be generated. Skipping the changes to the wider code for brevity, the type of Γ itself becomes:

```
data Γ′ xs r a =
  Γ′ {input :: Code String,               ops      :: Code (HList xs)
     , hs   :: Code [String → Maybe a] , retCont :: Code (r → String → Maybe a) }
type Eval″ xs r a = Γ′ xs r a → Code (Maybe a)
eval″ m = ⟦λinput → $(cata alg″ m (Γ′ ⟦input⟧ ⟦HNil⟧ ⟦[ ]⟧ ⟦noret⟧))⟧ where ...
```

Here, the dynamic Code has been pushed into each of the fields of Γ, so that the actual record itself is now statically known information. This now means that the record accessors used in the execution of instructions come for free. In fact, even more information is known statically about the Γ′ record: since the types of the values on the stack are known at compile time, this also indicates that the shape of the stack itself is static information. This is also true for the handler stack. As such, even these structures can be staged away at compile time:

```
data QList (xs :: [∗]) where
  QNil  :: QList [ ]
  QCons :: Code x → QList xs → QList (x : xs)
data Γ″ xs r a =
  Γ″ {input :: Code String,                ops      :: QList xs
     , hs   :: [Code (String → Maybe a) ], retCont :: Code (r → String → Maybe a) }
```

Now, the Code type constructor has been pushed inside the hs and ops fields themselves: hs is now a list of code instead of code of a list; and the heterogeneous list ops is now a QList, which stores code for each element instead of a value in the heterogeneous list. The implementation of the instructions and evaluation function must now be updated to reflect these changes:

```
eval‴ m = ⟦λinput → $(cata alg‴ m (Γ″ ⟦input⟧ QNil [ ] ⟦noret⟧))⟧ where ...
evalHalt‴ γ = let QCons qx _ = ops γ in ⟦Just $(qx)⟧
```

The change to eval is simple: instead of providing ⟦HNil⟧ and ⟦[ ]⟧ to Γ, QNil and [ ] are provided instead. For Halt, the stack operation is performed outside the quotation, and the code on the stack is wrapped up in a Just constructor within the quotation. The Lift2 instruction is similar:

```
evalLift2‴ qf k γ = let QCons qy (QCons qx xs) = ops γ
                    in k (γ {ops = QCons ⟦($(qf) $(qx) $(qy))⟧ xs})
```

Again, the stack operations have been moved outside of the quotations: the elements of the stack are obtained at compile time. The only work performed in this instruction at run-time is the application f x y, even pushing this new value back onto the stack happens at compile time.

```
evalFail‴ γ = case hs γ of qh : _ → ⟦$(qh) $(input γ)⟧
                           [ ]    → ⟦Nothing⟧
```

The story for Fail is similar, establishing whether or not a failure handler exists is an operation performed at compile time: if one exists the instruction returns the code which corresponds to the application of this handler to the input otherwise returns code representing Nothing.

```
evalSat‴ qf k γ = ⟦ case $(input γ) of
   c : cs | $(qf) c → $(k (γ {input = ⟦cs⟧, ops = QCons ⟦c⟧ (ops γ)}))
   _                → $(evalEmpt γ)⟧
```

In contrast, Sat illustrates an instruction which is almost entirely dynamic in nature: Checking whether the input contains a character matching the predicate is still performed inside the quotations, since the input is not known at compile time. However, the act of pushing this character onto the stack still happens at compile time, as it is within a splice. What is left after staging is something very refined with no extraneous overhead. The act of staging has enabled the complete compilation pipeline to be moved to compile time, so the consumer of the parser does not pay any of the cost of compilation or optimisation as is usually the case in parser combinator libraries.

6 BENCHMARKS

This section presents several benchmarks against a variety of libraries to support claims made about this library's performance. To perform these benchmarks, the Haskell library `criterion`[5] was used: this library ensures that results are forced and inputs are computed ahead of the benchmarks. It performs benchmarks multiple times and then performs statistical analysis on these results.

In order to be as fair as possible, parsers will follow the same grammars and produce the same outputs. Where this is not otherwise possible (such as comparing between C and Haskell), the parsers will be implemented as recognisers for the language, i.e. Parser (). It is worth noting that where libraries like `megaparsec` have specialised combinators for improving the speed of parsers, this will not be taken into account for these benchmarks. This reflects the advantage of our library that no specialist knowledge of the combinators is needed to write parsers: this seems favourable to the user, at least. The follow benchmarks are performed:

(1) A simple Branflakes benchmark, which serves to compare the most primitive operators as well as iteration and basic recursion
(2) A JavaScript benchmark, which provides a more comprehensive stress test of common patterns, such as recursion, expression parsing and non-trivial lexemes
(3) In order to compare the performance of C and Haskell, a Nandlang[6] recogniser was written in both Bison/Flex and our library

6.1 Branflakes

Branflakes is a language with 6 single-character operations (`+`, `-`, `.`, `,`, <, >) and a scoped operation (`[]`). Everything else in the language is considered a comment. This is a very simple grammar with a mix of iteration for linear sequences of operations and recursion when a `[` is encountered.

There are two benchmarks testing both `String` and `Text` as the input mediums: `String` is the traditional list of characters representation whereas `Text` is an often faster representation backed by a list of arrays. As `attoparsec` does not support `String` parsing, it is omitted from the first test, whereas `happy` is only tested on `String`s. Figure 4 shows that, relative to our library, other parser combinators are consistently worse and `happy` is at least 2x slower. While `attoparsec` can outperform `megaparsec`, it is still trailing behind Parsley. The improved relative performance of

[5] https://hackage.haskell.org/package/criterion
[6] https://github.com/Jellonator/Nandlang

Parsley in the `hello world` benchmark is due to the number of comments, for which fast code is generated. The compiler benchmark is slightly worse as it relies on non-tail recursion more.

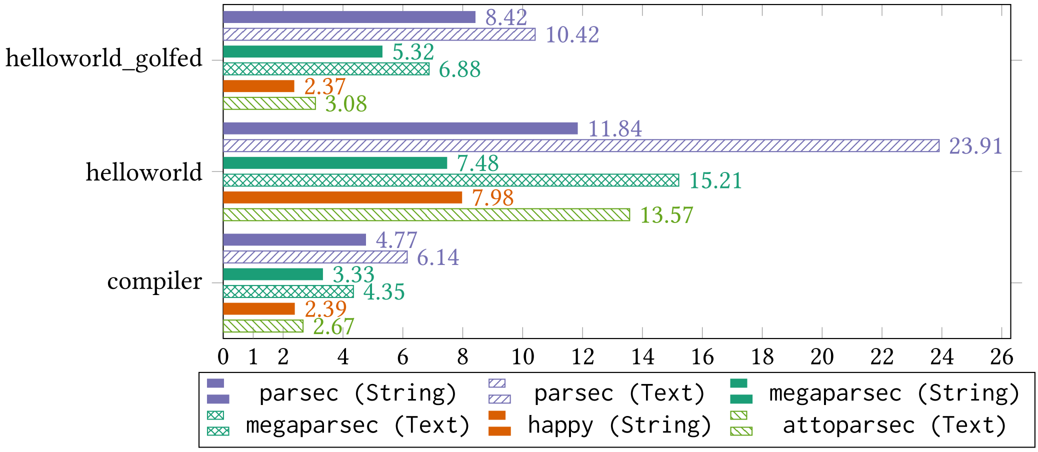

Fig. 4. Performance of libraries parsing Branflakes, time relative to Parsley

6.2 JavaScript

In order to more fairly test a larger parser, the second benchmark involves parsing JavaScript. The parsers themselves have been constructed carefully, which has minimised the chances of backtracking. This ensures good performance from all the libraries. The `happy` parser is given a hand-written and optimised lexer. The `happy` grammar does have some shift-reduce conflicts from the precedence and associativities of expressions, but the parser picks rules in the correct ordering. This should not affect performance. Note that the parser combinator libraries do not suffer from this problem, but care has been taken to use the `chainl1` combinator to prevent left-recursion.

Again, Figure 5 shows that our library routinely outperforms other parser combinator libraries by at least 4x and is anywhere from 1.5x to 2x faster than Happy. The fact that `megaparsec` and `parsec` perform similarly here may suggest that `megaparsec` is suffering from not using its specialised combinators in the `Text` benchmark, but this is not an advantage it receives with `String` anyway. Even still, as previously mentioned, it is simply an advantage of our library that the naïve parsers perform very well and do not require the user to learn how to use intrinsic combinators. Attoparsec results were not produced since the combinators return Text instead of String, making a fair comparison difficult here. The input files used are arbitrary JavaScript programs.

6.3 Nandlang

In the final benchmark, our library is being put up against a bottom-up generated C parser. The expectation is that C should be faster, but it provides an indication of how much further the library has left to go with optimisations. The language chosen for this benchmark is Nandlang, which is a simple language which has only a single operator: NAND. Otherwise, it still has the traditional looping constructs, conditionals and functions. The only literals in the language are `0`, `1` or an array.

In terms of the parsers themselves, the implementation is very similar, with a key difference that the `bison` parser accepts a slightly wider variant of the language due to the lexer not being selective of which tokens should be allowed in a given context. Essentially, since parser combinators do not often have a separate lexing stage, context-sensitive lexing comes a lot more naturally: you may know, for instance, that within an array index, the only valid token is an integer literal. Since two different languages are competing in this benchmark, neither will produce an AST. In C's

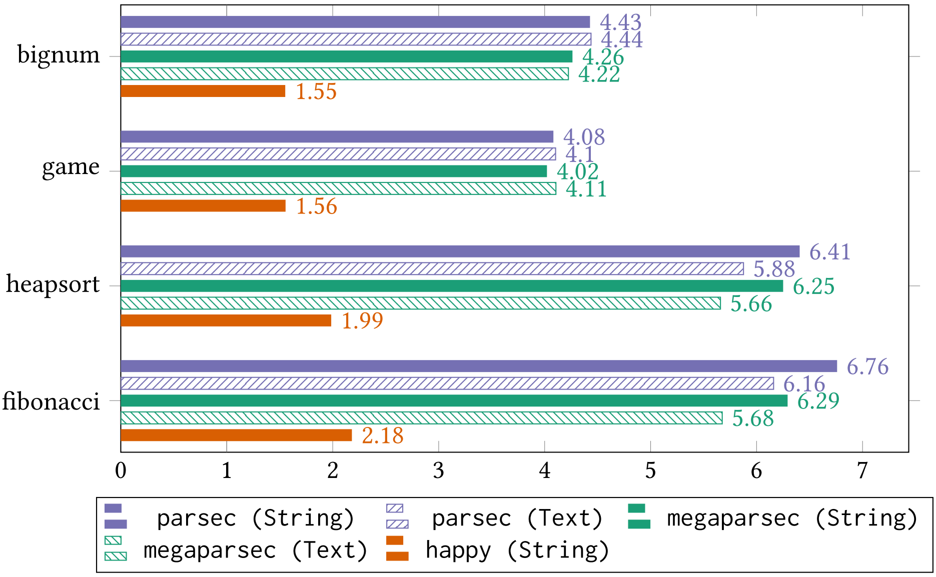

Fig. 5. Performance of libraries parsing JavaScript, time relative to Parsley

case, the parser will return `bool` and in Haskell's case it will return Maybe (). In order to make use of `criterion` here, the C parser is invoked using the FFI, and the return value is converted to Maybe (). Since the input needs to be sent from Haskell to C, both parsers make use of `Bytestring`.

Figure 6 shows that our library falls short of performance parity with `bison`. It is within 30% of the performance of C, however. Again the test files are arbitrary Nandlang programs.

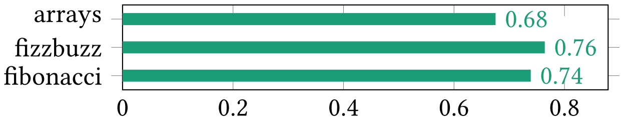

Fig. 6. Performance of `bison` parsing Nandlang, time relative to Parsley (as `Bytestring`)

7 RELATED AND FUTURE WORK

Parser generators and combinators are examples of *domain-specific languages* (DSLs) [Fowler 2010], with the latter an embedded DSL [Hudak 1996] where the DSL is hosted in some other language. Parser combinator libraries are usually *shallow* embedded DSLs, meaning the operations in the DSL provide a semantic action immediately. The problem is that there is overhead in the shallow embedding and by using static analysis this could be removed, but there is no structure left to inspect. In contrast, in a *deep* embedding the operations of the language are themselves just syntax: this allows for inspection and semantics are provided by interpretation [Gibbons and Wu 2014; Leijen and Meijer 1999]. The advantage of deep-embedded DSLs is that it is now possible to use the domain knowledge of the language to perform analysis and optimisation. Writing parser combinators as deep embeddings has been done in the context of free monads where effect handlers provide semantics [Wu et al. 2014]. Deep embeddings have been shown to be useful in performing a variety of parser optimisations and allows for a translation into a more efficient language before being interpreted [Willis and Wu 2018]. Our work further leverages the opportunities presented by using a deep embedding and performs analysis and optimisation by removing monadic operations and replacing some of the functionality with selectives.

Baars and Swierstra [2004] use a deep embedding to represent grammars, which allows them to perform transformations and analysis such as removing left-recursion. Further to this, Devreise and Piessens [2012] interpret these deep embeddings with explicit recursion using a finally-tagless style. Our work builds on this by modelling non-recursive let-bindings in addition to observable recursion, which facilitates the efficient staging of the parser.

The inspiration behind the join-points stems from both traditional compilers and let-insertion [Yallop 2017] in the staged world. While the problems arising from let-insertion were not covered in this paper, a good way to handle mutual recursion and staging has already been implemented in MetaOCaml [Yallop and Kiselyov 2019], but this is yet to make it to Haskell. Ljunglöf [2002] detects shared sections of a grammar by using unsafe IORefs, which increment a counter whenever they are touched. However, children of a let-binding will have higher reference counts resulting in unnecessary bindings. By using StableNames, a more accurate depiction of the sharing is achieved.

The semantics of the parsers themselves are modelled similarly to Parsec [Leijen and Meijer 2001], following the failure semantics very closely in particular. Libraries in the Parsec family are continuation based, and are all shallow embedded DSLs. Our machine is also continuation based, but this abstraction is largely staged away. The reason the failure semantics are useful is because it allows the parser to provide better error messages. This is the expected behaviour of contemporary parser combinator libraries in Haskell, and it will be important for staged error messages.

Staged parser combinators were pioneered in Scala using Light-weight Modular Staging (LMS) [Jonnalagedda et al. 2014]. This work illustrates how to add staging annotations to remove the abstraction overheads of the combinators. In addition to this they used similar techniques as us to remove the intermediate data-structures used in the parsing process. They achieved impressive performance which easily outperforms their contemporaries in Scala, highlighting the benefits of the approach. By using LMS [Rompf and Odersky 2010], they can rely on free optimisations like common subexpression elimination and they are able to pattern match on the generated code in order to perform domain-specific optimisations. However, by pattern matching on their generated representation, they have lost the high-level structure of the grammar: domain specific optimisations like the applicative laws are possible in either approach, but arise very naturally from the combinators themselves with minimal pattern-matching. By operating on a deep-embedded AST, our approach is able to repeatedly traverse and analyse the grammar, permitting richer optimisation opportunities including factoring out repeated length checks and fusing pure computations. By using the deep approach, it is also possible to avoid an explicit recursion combinator and allows for easy factoring out of common grammar rules. Our technique can be used to enhance the LMS approach even further, but it does come at a cost of additional staging overheads that are otherwise eliminated by compile-time code generation. However, by performing run-time code generation, they can support monadic operations, which provide a powerful tool for protocol-based parsing. At present, however, our library has been deliberately restricted to remove full monadic power, trading-off for compile-time code generation and more powerful analysis opportunities, an already documented technique [Marlow et al. 2014; McBride and Paterson 2008].

In addition, a staged parser combinator library has been developed in OCaml [Krishnaswami and Yallop 2019], however they have only tackled non-ambiguous grammars which can be realised by purely applicative combinators. Our work handles an extension of PEG with contextual decision making and backtracking. Their work is a deep embedding with a largely different API and no analysis or optimisation. They make use of a explicit fix-point operation instead of let-binding analysis and make use of mutual let-bindings from MetaOCaml. They make explicit use of “The Trick” [Danvy et al. 1996] – a dynamic value within a static range can be turned static by enumerating all possibilities – interestingly, we also do this, but *implicitly* by enumerate alternatives with the sbind :: (Selective f, Bounded a, Enum a) ⇒ f a → (a → f b) → f b operator.

Work done on parsing with derivatives [Adams et al. 2016; Brzozowski 1964; Henriksen et al. 2019] also has subtle connections to our work, where the CPS translation of AST to Machine is similar to differentiating with respect to that non-terminal. In particular, some derivatives work [Adams et al. 2016] also uses analysis to identify the recursion points in parsers. Recently, it has been suggested that derivative-based parsing may form new foundations for parsing in general [Henriksen et al. 2019]: this may suggest that CPS based abstract parsing machines are also foundational.

The translation to CPS has the effect of performing a co-density transformation [Hinze 2012; Voigtländer 2008] on the parsers themselves: if the machine had a (»=) operation, then every (»=) would be associated to the right during compilation, this is a well known optimisation for many monadic programs.

Both Viera et al. [2008] and Willis and Wu [2018] have implemented parser libraries by embedding the grammars into abstract syntax trees. This allows them to process and transform the grammars by performing optimisation and refactorings like left-factoring. The former uses template Haskell to convert their grammars into parsers, and the latter interprets an optimised machine with warm-up overhead. Our library produces optimised Haskell code directly, which improves performance and unlocks more low-level optimisation opportunities.

7.1 Future Work

While selectives have proved effective at removing the need for monads in many cases, they are not a catch-all. In order to recover the ability to interact with previous results multiple times or look back into the past, general purpose registers will be added, using an applicative state pattern with rank-2 polymorphism [McCracken 1984] to guarantee well-scopedness [Launchbury and Peyton Jones 1994]. This will allow the library to be fully context-sensitive and recover all the power of monads without loss of static analysis.

At the moment, error messages are not supported. In future, we aim to make use of the static information of the grammar to stage away space-leaking error message building at runtime, instead generating and inserting high-quality error messages during the compilation of the parser.

8 CONCLUSION

This paper presented the design and implementation of a parser combinator library with optimisation, analysis, and static compilation using staging. To accomplish this, it was necessary to make use of selective functors as this maximises the amount of static structure in the grammar. By expressing the parser as a deep embedding, laws of applicatives and selectives can be used to reduce the code size of the program. Analysis can be used to factor out length checks and identify non-terminating parsers. Any overhead is compiled away by staging, leaving an efficient core.

The benchmarks showed that this approach is effective, outperforming the conventional parser combinator libraries in Haskell and the parser generator happy with full optimisation. At the very least, this paper has hopefully highlighted the usefulness of the selective functor abstraction within the realm of parsers. But moreover, it demonstrated that by leveraging meta-programming, we can make use of the cleanest abstractions at our disposal without sacrificing any performance.

ACKNOWLEDGEMENTS

The authors would like to dedicate this paper to the memory of Doaitse Swierstra. He was an inspiring academic and a wonderful friend that we will dearly miss. We would also like to thank all the anonymous reviewers for their helpful and constructive comments on draft versions of this paper and Jonathan Protzenko for shepherding this paper. This work has been supported by EPSRC grant number EP/S028129/1 on “SCOPE: Scoped Contextual Operations and Effects”.

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

Michael D. Adams, Celeste Hollenbeck, and Matthew Might. 2016. On the Complexity and Performance of Parsing with Derivatives. *SIGPLAN Not.* 51, 6 (June 2016), 224–236. https://doi.org/10.1145/2980983.2908128

Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. 2006. *Compilers: Principles, Techniques, and Tools (2nd Edition).* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

Andrew W. Appel. 2007. *Compiling with Continuations.* Cambridge University Press, USA.

Arthur I. Baars and S. Doaitse Swierstra. 2004. Type-Safe, Self Inspecting Code. In *Proceedings of the 2004 ACM SIGPLAN Workshop on Haskell* (Snowbird, Utah, USA) *(Haskell âĂŹ04).* Association for Computing Machinery, New York, NY, USA, 69–79. https://doi.org/10.1145/1017472.1017485

Nick Benton. 2005. A Typed, Compositional Logic for a Stack-Based Abstract Machine. 364–380. https://doi.org/10.1007/11575467_24

Janusz A. Brzozowski. 1964. Derivatives of Regular Expressions. *J. ACM* 11, 4 (Oct. 1964), 481–494. https://doi.org/10.1145/321239.321249

Olivier Danvy, Karoline Malmkjær, and Jens Palsberg. 1996. Eta-expansion Does The Trick. *ACM Trans. Program. Lang. Syst.* 18, 6 (Nov. 1996), 730–751. https://doi.org/10.1145/236114.236119

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.

Dominique Devreise and Frank Piessens. 2012. Finally tagless observable recursion for an abstract grammar model. *Journal of Functional Programming* 22, 6 (2012), 757–796. https://doi.org/10.1017/S0956796812000226

Bryan Ford. 2002. *Packrat Parsing : a Practical Linear-Time Algorithm with Backtracking by.* Ph.D. Dissertation.

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

Martin Fowler. 2010. *Domain Specific Languages* (1st ed.). Addison-Wesley Professional.

Jeremy Gibbons and Ralf Hinze. 2011. Just Do It: Simple Monadic Equational Reasoning. *SIGPLAN Not.* 46, 9 (Sept. 2011), 2–14. https://doi.org/10.1145/2034574.2034777

Jeremy Gibbons 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* (Gothenburg, Sweden) *(ICFP '14).* ACM, New York, NY, USA, 339–347. https://doi.org/10.1145/2628136.2628138

Andy Gill. 2009. Type-Safe Observable Sharing in Haskell. In *Proceedings of the 2nd ACM SIGPLAN Symposium on Haskell* (Edinburgh, Scotland) *(Haskell '09).* Association for Computing Machinery, New York, NY, USA, 117–128. https://doi.org/10.1145/1596638.1596653

Andy Gill and Simon Marlow. 1995. Happy: the parser generator for Haskell.

Tatsuya Hagino. 1987. *Category theoretic approach to data types.* Ph.D. Dissertation. PhD thesis, University of Edinburgh.

Ian Henriksen, Gianfranco Bilardi, and Keshav Pingali. 2019. Derivative Grammars: A Symbolic Approach to Parsing with Derivatives. *Proc. ACM Program. Lang.* 3, OOPSLA, Article 127 (Oct. 2019), 28 pages. https://doi.org/10.1145/3360553

Ralf Hinze. 2012. Kan Extensions for Program Optimisation Or: Art and Dan Explain an Old Trick. In *Mathematics of Program Construction*, Jeremy Gibbons and Pablo Nogueira (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 324–362.

Ralf Hinze and Nicolas Wu. 2013. Histo- and Dynamorphisms Revisited. In *Proceedings of the 9th ACM SIGPLAN Workshop on Generic Programming* (Boston, Massachusetts, USA) *(WGP '13).* Association for Computing Machinery, New York, NY, USA, 1–12. https://doi.org/10.1145/2502488.2502496

Ralf Hinze, Nicolas Wu, and Jeremy Gibbons. 2013. Unifying Structured Recursion Schemes. In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming* (Boston, Massachusetts, USA) *(ICFP '13).* Association for Computing Machinery, New York, NY, USA, 209–220. https://doi.org/10.1145/2500365.2500578

Paul Hudak. 1996. Building Domain-specific Embedded Languages. *ACM Comput. Surv.* 28, 4es, Article 196 (Dec. 1996). https://doi.org/10.1145/242224.242477

Graham Hutton. 1992. Higher-order functions for parsing. *Journal of Functional Programming* 2, 3 (1992), 323–343. https://doi.org/10.1017/S0956796800000411

Graham Hutton and Erik Meijer. 1996. *Monadic Parser Combinators.* Technical Report NOTTCS-TR-96-4. Department of Computer Science, University of Nottingham.

Manohar Jonnalagedda, Thierry Coppey, Sandro Stucki, Tiark Rompf, and Martin Odersky. 2014. Staged Parser Combinators for Efficient Data Processing. *SIGPLAN Not.* 49, 10 (Oct. 2014), 637–653. https://doi.org/10.1145/2714064.2660241

Andrew Kennedy. 2007. Compiling with Continuations, Continued. In *Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming* (Freiburg, Germany) *(ICFP '07).* Association for Computing Machinery, New York, NY, USA, 177–190. https://doi.org/10.1145/1291151.1291179

Csongor Kiss, Matthew Pickering, and Nicolas Wu. 2018. Generic Deriving of Generic Traversals. *Proc. ACM Program. Lang.* 2, ICFP, Article 85 (July 2018), 30 pages. https://doi.org/10.1145/3236780

Dexter Kozen. 1997. Kleene Algebra with Tests. *ACM Trans. Program. Lang. Syst.* 19, 3 (May 1997), 427–443. https://doi.org/10.1145/256167.256195

Neelakantan R. Krishnaswami and Jeremy Yallop. 2019. A Typed, Algebraic Approach to Parsing. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Phoenix, AZ, USA) *(PLDI 2019)*. ACM, New York, NY, USA, 379–393. https://doi.org/10.1145/3314221.3314625

John Launchbury and Simon L. Peyton Jones. 1994. Lazy Functional State Threads. *SIGPLAN Not.* 29, 6 (June 1994), 24–35. https://doi.org/10.1145/773473.178246

Daan Leijen and Erik Meijer. 1999. Domain Specific Embedded Compilers. *SIGPLAN Not.* 35, 1 (Dec. 1999), 109–122. https://doi.org/10.1145/331963.331977

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

Peter Ljunglöf. 2002. *Pure Functional Parsing.* Ph.D. Dissertation. Chalmers University of Technology and Göteborg University.

Simon Marlow, Louis Brandy, Jonathan Coens, and Jon Purdy. 2014. There is No Fork: An Abstraction for Efficient, Concurrent, and Concise Data Access. *SIGPLAN Not.* 49, 9 (Aug. 2014), 325–337. https://doi.org/10.1145/2692915.2628144

Luke Maurer, Paul Downen, Zena M. Ariola, and Simon L. Peyton Jones. 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*, Albert Cohen and Martin T. Vechev (Eds.). ACM, 482–494. https://doi.org/10.1145/3062341.3062380

Conor McBride. 2011. Functional pearl: Kleisli arrows of outrageous fortune. *Journal of Functional Programming (accepted for publication)* (2011).

Conor McBride and Ross Paterson. 2008. Applicative programming with effects. *Journal of Functional Programming* 18, 1 (2008), 1–13. https://doi.org/10.1017/S0956796807006326

Nancy McCracken. 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., New York, NY, USA, 301–315. http://dl.acm.org/citation.cfm?id=1096.1107

Andrey Mokhov, Georgy Lukyanov, Simon Marlow, and Jeremie Dimino. 2019. Selective Applicative Functors. *Proc. ACM Program. Lang.* 3, ICFP, Article 90 (July 2019), 29 pages. https://doi.org/10.1145/3341694

Greg Morrisett, Karl Crary, Neal Glew, and David Walker. 2002. Stack-Based Typed Assembly Language. *J. Funct. Program.* 12, 1 (Jan. 2002), 43–88. https://doi.org/10.1017/S0956796801004178

Tiark Rompf and Martin Odersky. 2010. Lightweight Modular Staging: A Pragmatic Approach to Runtime Code Generation and Compiled DSLs. *SIGPLAN Not.* 46, 2 (Oct. 2010), 127–136. https://doi.org/10.1145/1942788.1868314

Tim Sheard and Simon Peyton Jones. 2002. Template Meta-programming for Haskell. *SIGPLAN Not.* 37, 12 (Dec. 2002), 60–75. https://doi.org/10.1145/636517.636528

S. Doaitse Swierstra. 2009. *Combinator Parsing: A Short Tutorial.* Springer Berlin Heidelberg, Berlin, Heidelberg, 252–300. https://doi.org/10.1007/978-3-642-03153-3_6

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, 184–207. http://dl.acm.org/citation.cfm?id=647699.734159

Walid Taha and Tim Sheard. 1997. Multi-stage Programming with Explicit Annotations. *SIGPLAN Not.* 32, 12 (Dec. 1997), 203–217. https://doi.org/10.1145/258994.259019

Tarmo Uustalu and Varmo Vene. 1999. Primitive (Co)Recursion and Course-of-Value (Co)Iteration, Categorically. *Informatica* 10 (1999), 5–26.

Marcos Viera, S. Doaitse Swierstra, and Eelco Lempsink. 2008. Haskell, Do You Read Me? Constructing and Composing Efficient Top-down Parsers at Runtime. In *Proceedings of the First ACM SIGPLAN Symposium on Haskell* (Victoria, BC, Canada) *(Haskell '08)*. Association for Computing Machinery, New York, NY, USA, 63–74. https://doi.org/10.1145/1411286.1411296

Janis Voigtländer. 2008. Asymptotic Improvement of Computations over Free Monads. In *Mathematics of Program Construction*, Philippe Audebaud and Christine Paulin-Mohring (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 388–403.

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 and Nicolas Wu. 2018. Garnishing Parsec with Parsley. In *Proceedings of the 9th ACM SIGPLAN International Symposium on Scala* (St. Louis, MO, USA) *(Scala '18)*. ACM, New York, NY, USA, 24–34. https://doi.org/10.1145/3241653.

3241656

Nicolas Wu, Tom Schrijvers, and Ralf Hinze. 2014. Effect Handlers in Scope. In *Proceedings of the 2014 ACM SIGPLAN Symposium on Haskell* (Gothenburg, Sweden) *(Haskell âĂŹ14)*. Association for Computing Machinery, New York, NY, USA, 1–12. https://doi.org/10.1145/2633357.2633358

Jeremy Yallop. 2017. Staged Generic Programming. *Proc. ACM Program. Lang.* 1, ICFP, Article 29 (Aug. 2017), 29 pages. https://doi.org/10.1145/3110273

Jeremy Yallop and Oleg Kiselyov. 2019. Generating Mutually Recursive Definitions. In *Proceedings of the 2019 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation* (Cascais, Portugal) *(PEPM 2019)*. ACM, New York, NY, USA, 75–81. https://doi.org/10.1145/3294032.3294078