Parsing with Zippers (Functional Pearl)

PIERCE DARRAGH, University of Utah, USA
MICHAEL D. ADAMS, University of Michigan, USA

Parsing with Derivatives (PwD) is an elegant approach to parsing context-free grammars (CFGs). It takes the equational theory behind Brzozowski's derivative for regular expressions and augments that theory with laziness, memoization, and fixed points. The result is a simple parser for arbitrary CFGs. Although recent work improved the performance of PwD, it remains inefficient due to the algorithm repeatedly traversing some parts of the grammar.

In this functional pearl, we show how to avoid this inefficiency by suspending the state of the traversal in a zipper. When subsequent derivatives are taken, we can resume the traversal from where we left off without retraversing already traversed parts of the grammar.

However, the original zipper is designed for use with trees, and we want to parse CFGs. CFGs can include shared regions, cycles, and choices between alternates, which makes them incompatible with the traditional tree model for zippers. This paper develops a generalization of zippers to properly handle these additional features. Just as PwD generalized Brzozowski's derivatives from regular expressions to CFGs, we generalize Huet's zippers from trees to CFGs.

The resulting parsing algorithm is concise and efficient: it takes only 31 lines of OCaml code to implement the derivative function but performs 6,500 times faster than the original PwD and 3.24 times faster than the optimized implementation of PwD.

CCS Concepts: • **Software and its engineering** → **Parsers**; *Syntax*; • **Theory of computation** → *Grammars and context-free languages*.

Additional Key Words and Phrases: Parsing; Derivatives; Zippers; Parsing with Derivatives

ACM Reference Format:
Pierce Darragh and Michael D. Adams. 2020. Parsing with Zippers (Functional Pearl). *Proc. ACM Program. Lang.* 4, ICFP, Article 108 (August 2020), 28 pages. https://doi.org/10.1145/3408990

1 INTRODUCTION

Parsing with Derivatives (PwD) [Might et al. 2011] generalizes from Brzozowski derivatives [Brzozowski 1964] of *regular expressions* to derivatives of *context-free grammars* (CFGs). Strings of input tokens can then be parsed by taking derivatives with respect to successive tokens. The result is an elegant technique for parsing CFGs. However, early implementations of PwD were too slow for practical use [Might et al. 2011]. Later work implemented a number of optimizations that improved performance but sacrificed elegance and concision in the code [Adams et al. 2016].

Even with these improvements, there remains an inefficiency: successive derivatives each start processing at the root of the grammar and often follow a path nearly identical to the immediately previous derivative. Both Might et al. [2011] and Adams et al. [2016] take a derivative starting at the

Authors' addresses: Pierce Darragh, University of Utah, School of Computing, 50 S Central Campus Drive, Room 3190, Salt Lake City, Utah, 84112, USA; Michael D. Adams, University of Michigan, Computer Science and Engineering, Electrical Engineering and Computer Science, College of Engineering, Bob and Betty Beyster Building, 2260 Hayward Street, Ann Arbor, Michigan, 48109-2121, USA.

2475-1421/2020/8-ART108
https://doi.org/10.1145/3408990

```
let derive (p : pos) (t : tok) ((e', m) : zipper) : zipper list =
  let rec d↓ (c : cxt) (e : exp) : zipper list =
    if p == e.m.start
    then (e.m.parents <- c :: e.m.parents;
          if p == e.m.end then d'↑ e.m.result c else [])
    else (let m = { start = p; parents = [c]; end = p⊥; result = e⊥ } in
          e.m <- m;
          d'↓ m e.e')

  and d'↓ (m : mem) (e' : exp') : zipper list =
    match e' with
    | Tok (t')         -> if t = t' then [(Seq (t, []), m)] else []
    | Seq (s, [])      -> d↑ (Seq (s, [])) m
    | Seq (s, e :: es) -> let m' = { start = m.start; parents = [AltC m];
                                     end = p⊥; result = e⊥ } in
                          d↓ (SeqC (m', s, [], es)) e
    | Alt (es)         -> List.concat (List.map (d↓ (AltC m)) !es)

  and d↑ (e' : exp') (m : mem) : zipper list =
    let e = { m = m⊥; e' = e' } in
    m.end <- p;
    m.result <- e;
    List.concat (List.map (d'↑ e) m.parents)

  and d'↑ (e : exp) (c : cxt) : zipper list =
    match c with
    | TopC                          -> []
    | SeqC (m, s, es, [])           -> d↑ (Seq (s, List.rev (e :: es))) m
    | SeqC (m, s, es_L, e_R :: es_R) -> d↓ (SeqC (m, s, e :: es_L, es_R)) e_R
    | AltC (m)                      -> if p == m.end
                                       then match m.result.e' with
                                            | Alt (es) -> es := e :: !es; []
                                       else d↑ (Alt (ref [e])) m

  in d↑ e' m
```

Fig. 1. Algorithm for derivatives using zippers. See Figure 2 for the corresponding types and constants.

root of the grammar, traverse down the grammar to find a token expression matching the current token, consume that token expression, and finally traverse up the grammar back to the root. For the next token, these algorithms again start at the root and traverse down the grammar to find a token expression matching the next token. In practice, these repeated traversals are redundant due to the next matching expression often being close to the previous one.

This paper uses zippers [Huet 1997] to avoid the redundant re-traversals in Might et al. [2011] and Adams et al. [2016] and in so doing eliminates this extra performance cost. The resulting algorithm is both more concise than PwD and outperforms PwD. Our algorithm is 6,500 times faster than the original PwD [Might et al. 2011] and 3.24 times faster than an optimized PwD [Adams et al. 2016]. The implementations in Might et al. [2011] and Adams et al. [2016] are, respectively, 196 and 238 lines long once you include all the macros and helpers. However, our implementation of the derivative is so short that we include it in its entirety as Figure 1 and Figure 2, which are 31 and 14 source lines of code respectively. Even when we include the driver loop and the conversion from grammars to abstract syntax trees it is still just 83 lines of code (see Appendix B).

However, arriving at the version of the algorithm shown in Figure 1 is not immediately straightforward. In this paper, we develop Parsing with Zippers (PwZ) incrementally, providing the reasoning behind each development so that the final algorithm can be understood with clarity.

```
type exp = { mutable m : mem; e' : exp' }
and exp' = Tok of tok
         | Seq of sym * exp list
         | Alt of (exp list) ref
and cxt  = TopC
         | SeqC of mem * sym * exp list * exp list
         | AltC of mem
and mem  = { start : pos;
             mutable parents : cxt list;
             mutable end : pos;
             mutable result : exp }

type zipper = exp' * mem

let rec e⊥ = { m = m⊥; e' = Alt (ref []) }
    and m⊥ = { start = p⊥; parents = []; end = p⊥; result = e⊥ }
```

Fig. 2. Types and constants for derivatives using zippers

1.1 Overview

Basics. First, we review the original PwD by starting with derivatives over simple grammars consisting of only tokens and sequencing (Section 2). Though this grammar is almost trivial, its simplicity aids in converting the code to use a zipper (Section 3). We then extend our grammar with choices between alternates (Section 4).

Optimizations. The algorithm in Section 4 is correct for non-cyclic grammars but takes exponential time in the worst case. To fix the exponential behavior, we introduce memoization for expressions (Section 5) and contexts (Section 6). We then show that this memoization also makes the algorithm handle cycles automatically (Section 7). Finally, we show how to eliminate these memoization tables (Section 8), which results in the code in Figure 1 and Figure 2.

Discussion. We discuss the results returned by our algorithm, some differences from traditional PwD, whether our algorithm truly generalizes the zipper, and our algorithm's asymptotic complexity (Section 9). We then benchmark our implementation (Section 10), discuss related work (Section 11), and conclude (Section 12).

Appendices. Appendix A contains the artifact for this paper as an embedded file. It includes a full implementation, several example grammars used as stress tests, and the benchmarks used in Section 10. Appendix B contains an ASCII version of the code in this paper including driver code and convenience functions.

1.2 Notation

We use the following notational conventions:

- When a diagram or piece of code is a modification of a previous diagram or piece of code, we typeset the changes in bold and the unchanged parts in gray.
- Variables are named according to the naming conventions in Figure 3. Variables with the same type are distinguished by subscripts.
- The types sym, tok, and pos are abstract, and our algorithm does not depend on their concrete implementation. However, readers can think of them as string, string, and int, respectively.
- The constants $\texttt{s}_\perp$, $\texttt{t}_\perp$, and $\texttt{p}_\perp$ represent dummy sym, tok, and pos values, respectively.
- Uppercase letters (e.g., A, B, or C) represent concrete tokens. This is in contrast to t, which is used for program *variables* containing tokens.
- An s suffix in a variable name indicates a list (e.g., e is one exp, but es is a list of exp).

`s ∈ sym`	Symbols for labeling constructors (Section 2)
`t ∈ tok`	Tokens, where `tok` ⊆ `sym` (Section 2)
`e ∈ exp`	Expressions (Section 2)
`c ∈ ctx`	Contexts (Section 3 and Section 4)
`z ∈ zipper`	Zippers (Section 3 and Section 4)
`p ∈ pos`	Input positions (Section 5 and Section 6)
`m ∈ mem`	Memoization records (Section 5 and Section 6)
`e' ∈ exp'`	Expressions without `mem` (Section 8)

Fig. 3. Types and variable names. Types are explained in the sections noted.

2 PARSING WITH DERIVATIVES

The basic idea behind Parsing with Derivatives (PwD) in both Brzozowski [1964] and Might et al. [2011] is quite simple. For a given `e` that is either a regular expression (as in Brzozowski [1964]) or context-free grammar (as in Might et al. [2011]), we first consider the set of strings accepted by `e` (i.e., its language, $[\![\mathtt{e}]\!]$). Given an input token `t`, we start parsing by computing $D_{\mathtt{t}}([\![\mathtt{e}]\!])$, the Brzozowski derivative of $[\![\mathtt{e}]\!]$ with respect to `t`. This is defined as the set of strings that, when prefixed with `t`, produce strings in $[\![\mathtt{e}]\!]$. Formally, $D_{\mathtt{t}}([\![\mathtt{e}]\!]) = \{w \mid \mathtt{t}w \in [\![\mathtt{e}]\!]\}$, where w is any string of tokens. For example, the derivative of $[\![\mathtt{e}]\!] = \{\mathtt{FOO}, \mathtt{BAR}, \mathtt{BAZ}\}$ with respect to `B` is $D_{\mathtt{B}}([\![\mathtt{e}]\!]) = \{\mathtt{AR}, \mathtt{AZ}\}$.

The Brzozowski derivative allows us to consider what strings (e.g., `AR` and `AZ`) are allowed to appear after the current input token (e.g., `B`). We can parse a string of tokens by successively taking derivatives with respect to each token in that string. Then we check whether the final language contains the empty string. If it does, then the string formed by the sequence of tokens is in the original language, and the parse is considered successful.

We do not directly compute the sets of strings produced by these derivatives as they can be infinitely large. Instead, Brzozowski [1964] and Might et al. [2011] show that we can compute these derivatives in terms of grammar expressions (i.e., `e`) instead of languages (i.e., $[\![\mathtt{e}]\!]$).

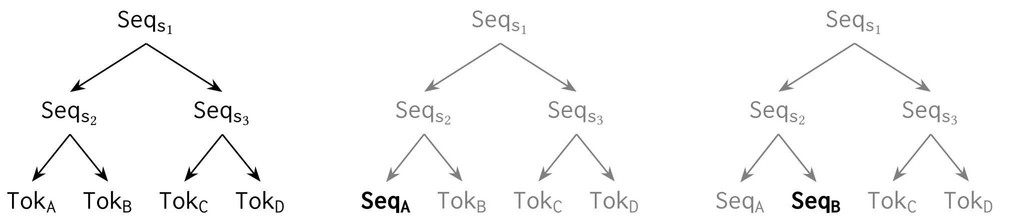
As an example, consider the grammars expressible by the `exp` type in Figure 4. The `Tok` constructor is the grammar that accepts a single token, and the `Seq` constructor is the grammar that accepts a sequential concatenation of grammars. These `Seq` constructors are labeled by symbols of type `sym` that we write in this paper as a subscript of `Seq` (e.g., `s` in $\mathtt{Seq}_{\mathtt{s}}$).

The symbol (`sym`) in the `Seq` constructor serves two purposes. First, by labeling the parse tree it allows clients of the parser to determine which part of a grammar was used for a particular parse. Second, as explained below, the derivative of a `Tok` constructor with respect to a token that matches it is a `Seq` constructor. In this case, we label the `Seq` constructor with the token that matched. This allows clients of the parser to inspect the tokens that were parsed. For example, clients may want to inspect the source locations of tokens. Thus, we require that tokens (`tok`) are a subtype of symbols (`sym`) (i.e., `tok` ⊆ `sym`).

As an example of the `exp` type, consider the grammar in Figure 5a where we draw arrows from each `Seq` constructor to those expressions in its list of children. The only input accepted by this grammar is the string `ABCD`. We can take the derivative of this grammar with respect to a token, say `A`, by traversing this grammar from left to right until we find a token constructor (`Tok`). If the `Tok` constructor contains a token that matches the token by which we are taking the derivative (`A` in this example), we replace the `Tok` constructor with a `Seq` constructor that has no children and contains the current input token (`A`) in its `sym` argument.

```
type exp = Tok of tok
         | Seq of sym * exp list
```

Fig. 4. Grammar expressions with only tokens and sequencing (i.e., concatenation)



(a) Initial grammar (b) Grammar after the derivative with respect to token A (c) Grammar after the derivative with respect to token B

Fig. 5. Example grammar and its derivatives

In Figure 5b we have taken the derivative of the grammar from Figure 5a with respect to the token A. Thus the Tok A expression is replaced by a Seq$_A$ constructor with no children. A childless Seq constructor is a grammar that accepts only the empty string. Thus, the grammar in Figure 5b accepts only the string BCD, which is what we expect from taking the derivative with respect to the token A of a language accepting only the string ABCD.

Finally Figure 5c is the derivative of Figure 5b with respect to B. Similar to Figure 5b this involves replacing Tok$_B$ with Seq$_B$ and results in a grammar that accepts only the string CD.

On the other hand, if the Tok constructor does *not* contain the token by which we are taking the derivative (e.g., if we take the derivative of the grammar in Figure 5a with respect to the token E), then the input should be rejected. To represent this potential for failure, the algorithm's return type in Section 3 is an `option` type, and the algorithm returns None when the derivative fails. From Section 4 onward, we instead return a list type and return an empty list when the derivative fails.

To parse an entire string of tokens, we take the derivative of the original expression with respect to the first token in the string. If that succeeds, we get back a new expression. We then take the derivative of that returned expression with respect to the second token. We repeat this process of taking a derivative with respect to each successive input token until either a parse fails or all of the input tokens are consumed. Finally, we accept the input string if and only if the empty string is in the language of the grammar produced by the final derivative.

3 ZIPPERS

As described in Section 2, derivatives start at the root of a grammar and traverse down the grammar to find a token expression (Tok). A new exp is created as the result of this derivative. The next derivative starts at the root of this new exp and performs nearly the same traversal while searching for the next token (Tok) expression.

Consider the grammar Figure 5. When parsing the token A, we traverse down the exp in Figure 5a through Seq$_{s_1}$ and Seq$_{s_2}$ to find Tok A. Parsing the second token (e.g., B) traverses down the exp in Figure 5b through Seq$_{s_1}$, Seq$_{s_2}$, and Seq$_A$ before finding Tok B. This duplicates the first parse's traversal of Seq$_{s_1}$ and Seq$_{s_2}$.

We avoid this repeated traversal by using a zipper.

```
type exp = Tok of tok
         | Seq of sym * exp list
type cxt = TopC
         | SeqC of cxt * sym * exp list * exp list

type zipper = exp * cxt
```

Fig. 6. Types for the derivative using zippers. Changes relative to Figure 4 in bold.

A zipper [Huet 1997] pairs a tree (called the *focus*) with a *context*. A context contains three parts: a list of left siblings of the focus, a list of right siblings of the focus, and a parent context. A zipper represents the position of an edge connecting two nodes in a tree, where the focus is the child of that edge and the context is the parent of that edge. Thus a zipper can efficiently save a position in a traversal and later resume the traversal from that position.

In Figure 6, we implement our zipper as a pair of an expression (exp) and the context containing the parts of the grammar around that expression (cxt). The cxt type represents a context by pointing from child nodes up to parent nodes, starting from the parent of the current focus and continuing up to the root. The SeqC constructor in cxt contains a parent context that points to the rest of the context going further up the grammar. It also contains a symbol of type sym obtained from the original Seq node represented by that SeqC, as well as two lists of exp representing the left and right siblings of the focus. The TopC context corresponds to the root of the grammar and so does not itself contain any parent context.

In our figures, we represent the zipper with a black dot (e.g., between TopC and Seq_{s_1} in Figure 8a). The arrows going up and down from the dot point to the cxt and exp of the zipper, respectively.

To parse the grammar in Figure 5a using our zipper, we first pair the grammar with the TopC context, producing the zipper in Figure 8a. We then proceed down the leftmost child of Seq_{s_1}, following the same traversal pattern that we used in Section 2. This produces the zipper in Figure 8b, where the focus is Seq_{s_2} and the context is SeqC_{s_1}. The SeqC_{s_1} context has TopC as the next higher context, an empty list for the focus's left siblings (because there are no sub-expressions of Seq_{s_1} that are to the left of Seq_{s_2}), and a singleton list containing Seq_{s_3} for the right siblings. Next, in Figure 8c, we have moved the focus to the left child of Seq_{s_2} (Tok A), and Seq_{s_2} has become the context SeqC_{s_2}. SeqC_{s_2} points to Tok B as a right sibling and up to SeqC_{s_1} as its parent context. Finally, supposing the input token is A, in Figure 8d the derivative replaces Tok A with $\mathsf{Seq}_{\mathsf{A}}$. A childless Seq is a grammar that accepts only the empty string and therefore is used to represent a successful parse. We also indicate the input token that resulted in this parse by recording it as the label in the Seq. At this point, we stop and return the zipper as it is: we have successfully parsed the first token.

If we were to take a derivative with respect to the next token using the algorithm in Section 2, we would have to re-traverse the entire grammar from the top to get to the next Tok. However, since we returned a zipper, we can instead use its context to resume exactly where we left off, getting to Tok B in just one step with the resulting zipper shown in Figure 8e. Thus, we have avoided the repeated traversal from the root of the grammar as well as any logic needed to handle the already-parsed $\mathsf{Seq}_{\mathsf{A}}$.

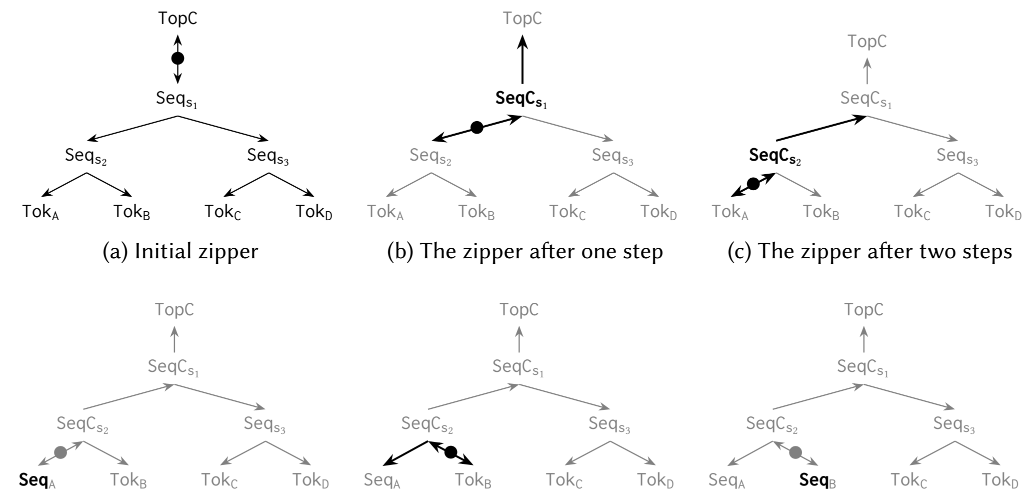
The code to implement this zipper-based derivative is in Figure 7. Its execution starts with the last line of code, $\mathsf{d}_\uparrow$ e c, and continues with the two mutually recursive functions, $\mathsf{d}_\downarrow$ and $\mathsf{d}_\uparrow$, that handle the parts of the traversal going down and up the grammar, respectively. Note that throughout this paper we follow the convention that the first and second arguments are being traversed out of and into, respectively. For example, $\mathsf{d}_\downarrow$ is traversing out of (i.e., down *from*) its first

```
let derive (t : tok) ((e, c) : zipper) : zipper option =
  let rec d↓ (c : cxt) (e : exp) : zipper option =
    match e with
    | Tok (t')         -> if t = t' then Some (Seq (t, []), c) else None
    | Seq (s, [])      -> d↑ (Seq (s, [])) c
    | Seq (s, e :: es) -> d↓ (SeqC (c, s, [], es)) e

  and d↑ (e : exp) (c : cxt) : zipper option =
    match c with
    | TopC                          -> None
    | SeqC (c, s, es, [])           -> d↑ (Seq (s, List.rev (e :: es))) c
    | SeqC (c, s, es_L , e_R :: es_R) -> d↓ (SeqC (c, s, e :: es_L , es_R)) e_R

  in d↑ e c
```

Fig. 7. Code for the derivative using zippers



(a) Initial zipper (b) The zipper after one step (c) The zipper after two steps

(d) The zipper after three steps and taking a derivative by token A (e) The zipper after four steps (f) The zipper after five steps and taking a derivative by token B

Fig. 8. Using a zipper to compute derivatives

argument c and is traversing into (i.e., down *to*) its second argument e. On the other hand, $d_\uparrow$ is traversing out of (i.e., up *from*) its first argument e and is traversing into (i.e., up *to*) its second argument c.

As an example of how this code works, the transition from Figure 8a to Figure 8b is handled by the second Seq clause in $d_\downarrow$, which simply moves the zipper to the first child of the Seq. The transition from Figure 8b to Figure 8c is handled the same way. The transition from Figure 8c to Figure 8d is handled by the Tok clause of $d_\downarrow$. Instead of continuing the traversal upwards by calling $d_\uparrow$, we save our current position by returning the zipper corresponding to Figure 8d. (If the token does not match, we instead return None to indicate a parse failure.) When processing the next token, `derive` resumes this traversal by calling $d_\uparrow$ with the exp and cxt saved in the zipper so that the traversal continues from where it left off. Then the second SeqC clause of $d_\uparrow$ would call $d_\downarrow$ on

```
type exp = Tok of tok
         | Seq of sym * exp list
         | Alt of exp list
type cxt = TopC
         | SeqC of cxt * sym * exp list * exp list
         | AltC of cxt

type zipper = exp * cxt
```

Fig. 9. Types for the derivative using zippers with alternates. Changes relative to Figure 6 in bold.

the next right-hand sibling. This moves the zipper into `Tok B`, giving us Figure 8e. Lastly, `Tok B` is replaced by $\texttt{Seq}_\texttt{B}$ and we get Figure 8f.

Since the derivative traverses from left to right, the left siblings of a `SeqC` do not affect whether a subsequent parse will be successful or not. However, they are the remains of the successfully-matched parts of the grammar and could be useful to clients of the parser. Once all the tokens of a string are parsed and the zipper has gone up to `TopC`, the resulting focus is the final parse tree to be returned.

Note the call to `List.rev` in Figure 7. OCaml uses the "cons" lists typical in functional languages, where prepending to the left side of a list is much more efficient than appending to the right. This means that when we finish traversing a child `exp` of a `Seq` and add it to the corresponding `SeqC`'s list of left children (as in the second `SeqC` clause of $\texttt{d}_\uparrow$), it is faster to put that `exp` at the *head* of the list of left children. Thus, when all children have been traversed (i.e., when the first `SeqC` clause of $\texttt{d}_\uparrow$ runs), we need to reverse the list of children that we use in the new `Seq` so its children are in the proper order.

The `derive` function returns a zipper when the parse succeeds (i.e., when execution reaches the `true` branch of the `if` expression in the `Tok` clause of $\texttt{d}_\downarrow$). This zipper will then be passed to `derive` when we attempt to parse the next token. Consequently, `derive` assumes that the zipper passed in as argument represents an in-progress derivative that has just successfully parsed a token and needs to traverse *up* the grammar, which is why the first action in `derive` is a call to $\texttt{d}_\uparrow$. This poses a problem for us when parsing the very first token of a string as the parse needs to start by traversing *down* the grammar. To resolve this, we construct the following zipper (where `e` is the grammar to be parsed) that is used as the initial zipper when parsing starts:

```
(Seq (s⊥, []), SeqC(TopC, s⊥, [], [e])
```

Traversing *up* this zipper causes the second `SeqC` clause of $\texttt{d}_\uparrow$ to run, which starts a traversal *down* `e`. As this is a special case for only the first token, we gloss over this detail in the rest of this paper.

4 ALTERNATES

Grammars made up of only token expressions (`Tok`) and sequencing (`Seq`) can accept only a single string. To support grammars that accept multiple strings, we add the `Alt` and `AltC` constructors in Figure 9 to the `exp` and `cxt` types, respectively. (As an invariant, we forbid cycles until Section 7.)

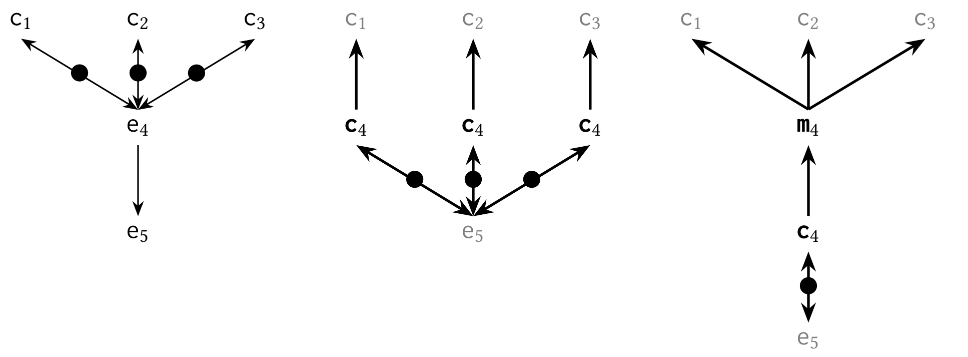
When $\texttt{d}_\downarrow$ comes to an `Alt`, one might expect that we would traverse into the leftmost child just as we would for `Seq`, but this does not reflect the meaning of `Alt`. Unlike `Seq`, whose children represent sequential parts of the input, the children of `Alt` represent alternate paths for the *same* part of the input. Rather than traversing an `Alt`'s children sequentially, we should traverse them simultaneously! This is a departure from the usual zippers but can be thought of as if each `Alt` represents a non-deterministic choice. When we encounter an `Alt`, we can lift that non-determinism

```
let derive (t : tok) ((e, c) : zipper) : zipper list =
  let rec d↓ (c : cxt) (e : exp) : zipper list =
    match e with
    | Tok (t')         -> if t = t' then [(Seq (t, []), c)] else []
    | Seq (s, [])      -> d↑ (Seq (s, [])) c
    | Seq (s, e :: es) -> d↓ (SeqC (c, s, [], es)) e
    | Alt (es)         -> List.concat (List.map (d↓ (AltC c)) es)

  and d↑ (e : exp) (c : cxt) : zipper list =
    match c with
    | TopC                        -> []
    | SeqC (c, s, es, [])         -> d↑ (Seq (s, List.rev (e :: es))) c
    | SeqC (c, s, esL , eR :: esR) -> d↓ (SeqC (c, s, e :: esL , esR)) eR
    | AltC (c)                    -> d↑ (Alt [e]) c

  in d↑ e c
```

Fig. 10. Code for the derivative using zippers with alternates. Changes relative to Figure 7 in bold.



(a) Initial zippers

(b) Zippers after a move down from Figure 11a that *does not* memoize the shared exp

(c) Zippers after a move down from Figure 11a that *does* memoize the shared exp

Fig. 11. Memoizing shared expressions

into the traversal by creating a list of zippers (one for each child of the Alt) that then proceed independently of each other.

This is implemented in Figure 10 where we change the types of derive, $d_\downarrow$, and $d_\uparrow$ to return zipper list instead of zipper option. In $d_\downarrow$, we also change the Tok clause and add an Alt clause. The clauses for Seq remain the same as before, and the Tok clause is changed only slightly to return a list instead of an option. The interesting case is Alt, where we use List.concat and List.map to derive each child and concatenate the lists of zippers that result.

The AltC context created by this code does not track the other zippers created from the same Alt by $d_\downarrow$. Once these zippers split off from each other, they are completely independent, and each behaves as if it were the only child in the Alt. In $d_\uparrow$, the AltC clause thus matches this behavior by creating an Alt with only one child. This means that each child of an Alt parsed going down will produce a separate single-child Alt when going back up.

```
type exp = Tok of tok
         | Seq of sym * exp list
         | Alt of exp list
type cxt = TopC
         | SeqC of mem * sym * exp list * exp list
         | AltC of mem
and mem  = { mutable parents : cxt list; result : pos ⇰ exp list }

type zipper = exp * mem

let mems : (pos * exp) ⇰ mem = ∅
```

Fig. 12. Types and values for the derivative using zippers with shared expressions. Changes relative to Figure 9 in bold.

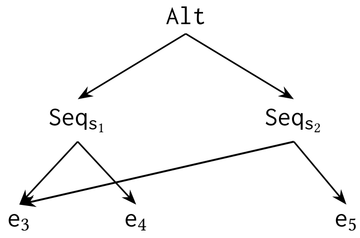


Fig. 13. A grammar that requires populating the memoization table before parsing sub-grammars

For non-cyclic grammars, this algorithm is semantically correct. However, there remain opportunities to significantly improve its computational complexity.

5 MEMOIZING SHARED EXPRESSIONS

The algorithm in Section 4 can take exponential time if some sub-grammar is shared between different parts of the grammar. For example, suppose we have the situation in Figure 11a where multiple zippers have the same focus. Since all of these zippers start parsing at the same place in the grammar and are identical in the expression, their results will be identical. The algorithm in Section 4 does not take advantage of this sharing and traverses into e_4 multiple times. This is shown in Figure 11b where multiple copies of the same context, c_4, that differ only in what parent they point to are created for each call that parses with e_4. Each time this sort of sharing occurs, the number of zippers in our list multiplies. Once we add support for cycles (see Section 7), this can compound and lead to the number of zippers being exponential in the length of the input.

In order to prevent this exponential blowup, we memoize $d_\downarrow$ so that a parse at a particular position (`pos`) with a particular expression (`exp`) is reused by other calls to parse at that position with that expression. This is only temporary, and in Section 8 we show how to eliminate this memoization table. We accomplish memoization with the types and values in Figure 12. The global memoization table `mems` starts empty (∅) and has type `(pos * exp) ⇰ mem`. This type is a mutable map from pairs of input positions (`pos`) and expressions (`exp`) to memoization records (`mem`). The `result` field of `mem` maps input positions (`pos`) to lists of expressions (`exp`). Memoization populates these maps so that if a parse has already been constructed over a region of input from p_{start} to p_{end} with an expression `e`, then `mems.get(`p_{start}`, e).result.get(`p_{end}`)` contains the `exp` returned by that parse. Later calls to parse with that expression can use this memoization to avoid redundantly re-parsing over the same part of the input.

```
let derive (p : pos) (t : tok) ((e, c) : zipper) : zipper list =
  let rec d↓ (c : cxt) (e : exp) : zipper list =
    match mems.get(p, e) with
    | Some (m) -> m.parents <- c :: m.parents;
                  List.concat (List.map (fun e -> d′↑ e c) m.result.get(p))
    | None     -> let m = { parents = [c]; result = ∅ } in
                  mems.put(p, e, m);
                  d′↓ m e

  and d′↓ (m : mem) (e : exp) : zipper list =
    match e with
    | Tok (t')         -> if t = t' then [(Seq (t, []), m)] else []
    | Seq (s, [])      -> d↑ (Seq (s, [])) m
    | Seq (s, e :: es) -> d↓ (SeqC (m, s, [], es)) e
    | Alt (es)         -> List.concat (List.map (d↓ (AltC m)) es)

  and d↑ (e : exp) (m : mem) : zipper list =
    m.result.put(p, e :: m.result.get(p));
    List.concat (List.map (d′↑ e) m.parents)

  and d′↑ (e : exp) (c : cxt) : zipper list =
    match c with
    | TopC                         -> []
    | SeqC (m, s, es, [])          -> d↑ (Seq (s, List.rev (e :: es))) m
    | SeqC (m, s, esL , eR :: esR) -> d↓ (SeqC (m, s, e :: esL , esR)) eR
    | AltC (m)                     -> d↑ (Alt [e]) m

  in d↑ e c
```

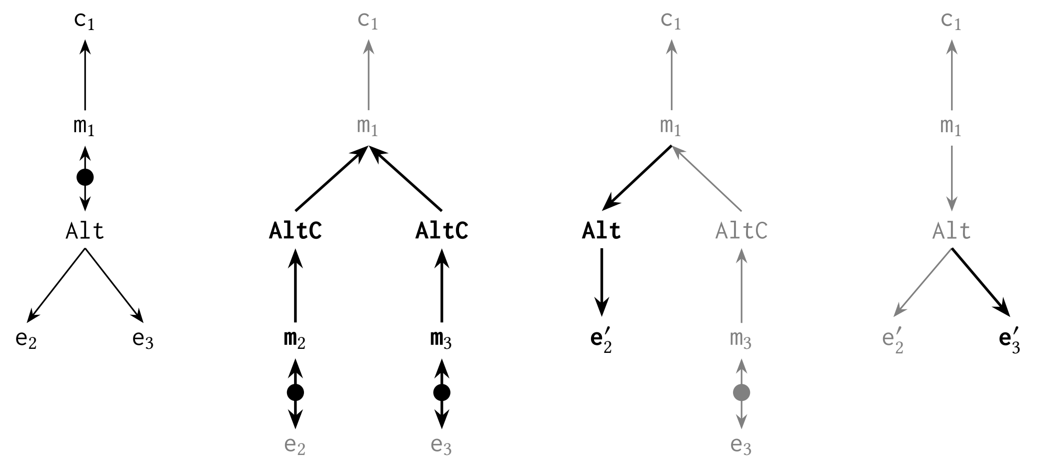
Fig. 14. Code for the derivative using zippers with shared expressions. Changes relative to Figure 10 in bold.

However, there is a complication. Memoization tables are usually populated only once a result is returned, but consider what happens if we do this with the grammar in Figure 13. When parsing with the Alt, the first child (Seq_{s_1}) starts a call to parse with the shared parse expression e_3. Suppose the token being parsed is found in e_3. In that case, a zipper positioned inside e_3 is returned and waits for the next input token. The second child of the Alt (i.e., Seq_{s_2}) also starts a call to parse with e_3. However, the zipper from the first parse has not yet traversed out of e_3, so there is no memoization record, and the parser starts a redundant call to parse with e_3. To prevent this, we need to make this call stop and wait for the result from the first call.

If we add a memoization record to mems *before* any recursive calls, we can detect that a call has already started by checking if there is a corresponding entry in mems. When a result is returned, we restart any calls that were waiting for it. We implement this using the parents field of mem. When a call stops and waits, its cxt is added to parents. When a result exp is returned, that exp is passed to $d_\uparrow$ along with those contexts.

This memoization is implemented in Figure 14 where we add p, the position of the current token, as a parameter to derive and (with slight modifications) rename $d_\downarrow$ and $d_\uparrow$ to $d'_\downarrow$ and $d'_\uparrow$, respectively. The $d'_\downarrow$ and $d'_\uparrow$ functions continue to contain the core parsing logic but are now wrapped with new implementations of $d_\downarrow$ and $d_\uparrow$ that implement memoization. The main difference in $d'_\downarrow$ and $d'_\uparrow$ is that they save and restore m to and from each SeqC and AltC. This tracks what memoization record to pass to any eventual calls to $d_\uparrow$.

In $d_\downarrow$, we first use mems.get(p, e) to check if there is a memoization record for the current input position, p, and expression, e. If mems contains such a record, then the Some clause runs with



(a) Initial zipper (b) Zippers after a move down from Figure 15a (c) Zippers after e_2 in Figure 15b finishes parsing (d) Zippers after e_3 in Figure 15c finishes parsing

Fig. 15. Memoizing shared contexts

the memoization record m. Since there is an entry in mems for p and e, we know that the parse we are about to start has already been started. Therefore, instead of calling $d'_\downarrow$ to do that parse, we add the current context, c, to `m.parents` so that any future results are passed to c. Since some results may already have been produced, we also pass those results to the context c by applying $d'_\uparrow$ to all the exp in `m.result.get(p)`.

If there is no memoization record for p and e, the None clause executes. This means that this is the first call to parse at input position p with expression e. To ensure that later calls stop and wait for our results, we add a memoization record, m, to mems before proceeding to parse with a call to $d'_\downarrow$. This memoization record, m, also contains c as one of its parent contexts waiting for a result.

When $d_\uparrow$ is called with a result e, we simply save e in `m.result` for any later parsing calls, and we pass e to any waiting contexts in `m.parents`.

As an example, consider again Figure 11. When going down from c_1 in Figure 11a, a new memoization record, m_4, is created. When c_2 and c_3 also go down into e_4, they find this memoization record. Instead of traversing into e_4, which would lead to Figure 11b and the exponential behavior we wish to avoid, they add themselves to the `parents` of m_4. This results in the structure in Figure 11c. Any results that eventually come from e_4 are then shared between c_1, c_2, and c_3, which prevents the potentially exponential behavior.

6 MEMOIZING SHARED CONTEXTS

While the memoization table in Section 5 eliminates exponential behavior when going *down* the grammar with $d_\downarrow$ over a shared exp, another exponentiality remains when going *up* with $d_\uparrow$ over a shared cxt. This is due to `Alt` expressions in situations like in Figure 15a. When traversing down the `Alt`, $d'_\downarrow$ creates two different `AltC` contexts that then proceed independently but have a shared context, as shown in Figure 15b. When the parsing process returns a result exp, the shared context is not taken advantage of. Instead, the two results are passed to the context separately. Like with

```
type exp = Tok of tok
         | Seq of sym * exp list
         | Alt of **(exp list) ref**
type cxt = TopC
         | SeqC of mem * sym * exp list * exp list
         | AltC of mem
and mem  = { mutable parents : cxt list; result : pos ⇰ **exp** }

type zipper = exp * mem

let mems : (pos * exp) ⇰ mem = ∅
```

Fig. 16. Types and values for the derivative using zippers with shared expressions and contexts. Changes relative to Figure 12 in bold.

shared expressions, this shared context multiplies the number of zippers in a way that could lead to the number of zippers being exponential in the length of the input.

To solve this, we memoize over `cxt` in addition to `exp`. We then detect sharing of a context and fuse all of the `exp` passed up to that context into a single `Alt` containing all of those results.

We take advantage of the fact that all `AltC` created from an `Alt` share the same memoization record. For example, in Figure 15b this is $\mathtt{m}_1$. When a result is produced, say `Alt` $[\mathtt{e}_2']$ constructed after parsing the $\mathtt{e}_2$ branch of `Alt` $[\mathtt{e}_2;\ \mathtt{e}_3]$, it is added to the `result` field of $\mathtt{m}_1$. This causes the situation in Figure 15c where $\mathtt{m}_1$ points to the `Alt` on the left as one of its results, but the `AltC` on the right has not yet returned and thus points to $\mathtt{m}_1$ as its parent.

Then when another result is produced that ends at the same input position, say the `Alt` $[\mathtt{e}_3']$ produced by parsing the $\mathtt{e}_3$ branch of `Alt` $[\mathtt{e}_2;\ \mathtt{e}_3]$, we do not add it to $\mathtt{m}_1$ as a new result. Instead, we add its lone child as an additional child of the `Alt` that is already in the `result` of $\mathtt{m}_1$. This produces the outcome in Figure 15d where the result is a single `Alt` containing both the $\mathtt{e}_2'$ and $\mathtt{e}_3'$ results instead of having a separate `Alt` result for each.

Figure 16 contains the types and values to implement the memoization of shared contexts. To allow results to be added to an `Alt` after the `Alt` is constructed, the `Alt` constructor now contains a mutable reference to its children that is of type `(exp list) ref`, which can be updated as new children are produced.

Code for this algorithm is in Figure 17. The main change is in the `AltC` clause of $\mathtt{d}_\uparrow'$ where we add each new child to an existing result rather than produce a new `Alt`. Because of this, there is at most one entry at any given input position in the `result` field of a memoization record. The `result` field thus now maps to a single `exp` instead of a list of `exp`. Note that the order in which children of an `Alt` are added to the `result` does not matter because an `Alt` represents non-deterministic choice.

Another change is in the second `Seq` clause of $\mathtt{d}_\downarrow'$ and corresponds to the `Alt` introduced in traditional PwD when the first child of a `Seq` is nullable. Parsing with a single `Seq` can produce multiple new `Seq` that all start and end at the same position, which happens if there are multiple positions that can be the boundary between the children of that `Seq`. For example, consider parsing the string `AAA` with the grammar $\mathtt{e}_1$ `::=` $\mathtt{e}_2\ \mathtt{e}_2$`;` $\mathtt{e}_2$ `::= A | AA`. The first and second children of $\mathtt{e}_1$ could be `A` and `AA`, respectively, or they could be `AA` and `A`. This results in two `Seq` that could be returned. Multiple iterations of this doubling could cause exponential behavior. To avoid this we introduce a parent `AltC` that both `Seq` are returned to. This wraps all the returned `Seq` in a single `Alt`, so parents of $\mathtt{e}_1$ have only one grammar result to deal with. In the code, this is implemented by setting the parents of the `SeqC` passed to $\mathtt{d}_\downarrow$ to be `[AltC m]` instead of `m.parents`.

```
let derive (p : pos) (t : tok) ((e, m) : zipper) : zipper list =
  let rec d↓ (c : cxt) (e : exp) : zipper list =
    match mems.get(p, e) with
    | Some (m) -> m.parents <- c :: m.parents;
                  (match m.result.get(p) with
                   | Some e -> d′↑ e c
                   | None   -> [])
    | None     -> let m = { parents = [c]; result = ∅ } in
                  mems.put(p, e, m);
                  d′↓ m e

  and d′↓ (m : mem) (e : exp) : zipper list =
    match e with
    | Tok (t')         -> if t = t' then [(Seq (t, []), m)] else []
    | Seq (s, [])      -> d↑ (Seq (s, [])) m
    | Seq (s, e :: es) -> let m' = { parents = [AltC m]; result = ∅ } in
                          d↓ (SeqC (m', s, [], es)) e
    | Alt (es)         -> List.concat (List.map (d↓ (AltC m)) !es)

  and d↑ (e : exp) (m : mem) : zipper list =
    m.result.put(p, e);
    List.concat (List.map (d′↑ e) m.parents)

  and d′↑ (e : exp) (c : cxt) : zipper list =
    match c with
    | TopC                         -> []
    | SeqC (m, s, es, [])          -> d↑ (Seq (s, List.rev (e :: es))) m
    | SeqC (m, s, es_L, e_R :: es_R) -> d↓ (SeqC (m, s, e :: es_L, es_R)) e_R
    | AltC (m)                     -> (match m.result.get(p) with
                                        | Some (Alt (es)) -> es := e :: !es; []
                                        | None -> d↑ (Alt (ref [e])) m)

  in d↑ e m
```

Fig. 17. Code for the derivative using zippers with expression and context sharing. Changes relative to Figure 14 in bold.

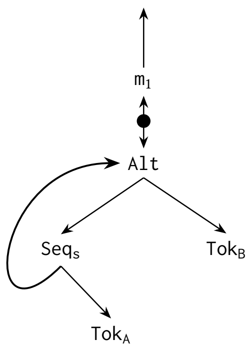
7 CYCLES

Up to this point, we have not addressed what happens when a grammar contains cycles or, more specifically, left recursion. Most recursive-descent parsers have trouble with left recursion as they will follow it in an infinite loop. However, with the memoization added in Section 5 and Section 6, our parser already handles cycles automatically.[1]

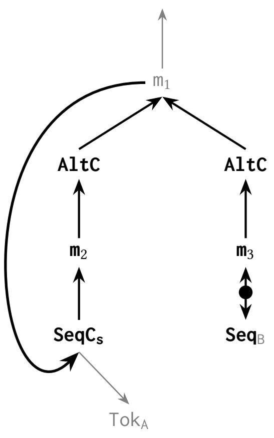
To see how this works, consider the situation in Figure 18a where the zipper is about to start down into the `Alt` in the left-recursive grammar $\texttt{e}_1$ `::=` $\texttt{e}_1$ `A | B`, and suppose the first input token is `B`. When $\texttt{d}_\downarrow$ traverses into the `Alt`, both children are traversed by $\texttt{d}_\downarrow$. The `Tok B` child is straightforward and results in the $\texttt{Seq}_\texttt{B}$ in Figure 18b. (Recall that `Alt` represents non-determinism, so we can process children in any order.) This leaves the $\texttt{Seq}_\texttt{s}$ child. Initially, the derivative proceeds

[1]OCaml's `let rec` supports cyclic values so long as the right-hand sides of the `let rec` bindings are "statically constructive" [Leroy et al. 2020, Section 8.1]. This can be achieved by using only constructor applications and variables in those bindings. For presentational simplicity, this is the approach we use this paper. However, since function calls are *not* statically constructive, this precludes the use of smart constructors and other abstractions.

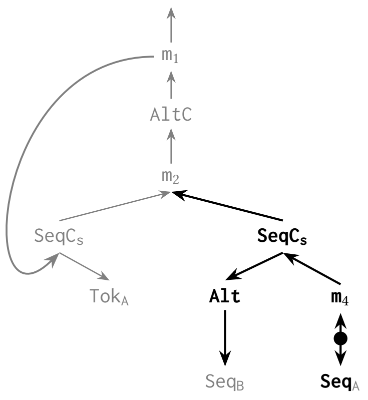
An alternative is to wrap `exp` with the `Lazy.t` type and use the `lazy` keyword to break cycles in the `let rec` bindings. This requires inserting into the code for `derive` a few calls to `Lazy.force` and uses of the `lazy` keyword, but otherwise does not change the algorithm.



(a) Initial zipper



(b) Zipper after Figure 18a consumes the input token B



(c) Zipper after Figure 18b consumes the input token A

Fig. 18. Zippers parsing the string BAA with the left-recursive grammar `e₁ ::= e₁ A | B`

normally: the parser traverses to the Seq$_s$ child of the `Alt` and builds up the `AltC` context on the left of Figure 18b. It then traverses into the Seq$_s$, and then into the left child of Seq$_s$. In the process, it builds the SeqC$_s$ context.

Once d$_\downarrow$ processes the left child of Seq$_s$, the algorithm starts to interact with the cycle. When first traversing the `Alt` in Figure 18a, m$_1$ was added to the `mems` table for that `Alt` at the current input position. When d$_\downarrow$ goes through the cycle and traverses to the `Alt` for a second time, it finds m$_1$. Instead of traversing into the `Alt`, the `Some` clause of d$_\downarrow$ runs and adds SeqC$_s$ to the `parents` field of m$_1$. This adds the curved arrow in Figure 18b and delays the current parse until the previously-started parse completes.

This leaves us with the structure in Figure 18b. Whereas most recursive-descent parsers are faced with the dilemma of choosing how many times to descend into a left recursion, we simply convert the cycle in the *grammar* into a cycle in the *context*. This allows us to postpone making this choice until further parsing reveals how many repetitions are needed.

When the next token is parsed, the zipper below m$_3$ in Figure 18b moves up through the right-hand `AltC` and builds the `Alt` in Figure 18c. As it continues moving up, it finds the two parents in m$_1$: one recursive, and one non-recursive. Traversing up any further causes the algorithm to proceed through both parents.

Traversing up through the non-recursive case simply results in the zipper being moved up to the surrounding context. This case is not specifically relevant here.

In the recursive case, though, an upward traversal causes the algorithm to loop back down to SeqC$_s$, and then into the right-hand child of SeqC$_s$, `Tok A`. In the process of performing this traversal, a new SeqC$_s$ is constructed (as shown in bold in Figure 18c). This SeqC$_s$ contains the result that came from Seq$_B$ through the right-hand `AltC` as its left child, which is an `Alt` with child Seq$_B$. The new (right-hand) SeqC$_s$ also copies the memoization record from the original (left-hand) SeqC$_s$, so now each SeqC$_s$ points to m$_2$ as its parent. At this point, the zipper's focus is `Tok A`. Assuming the next input token is A, `Tok A` is replaced with Seq$_A$, and the traversal state is saved as a zipper whose focus is that new Seq$_A$, leaving us with the resulting structure in Figure 18c.

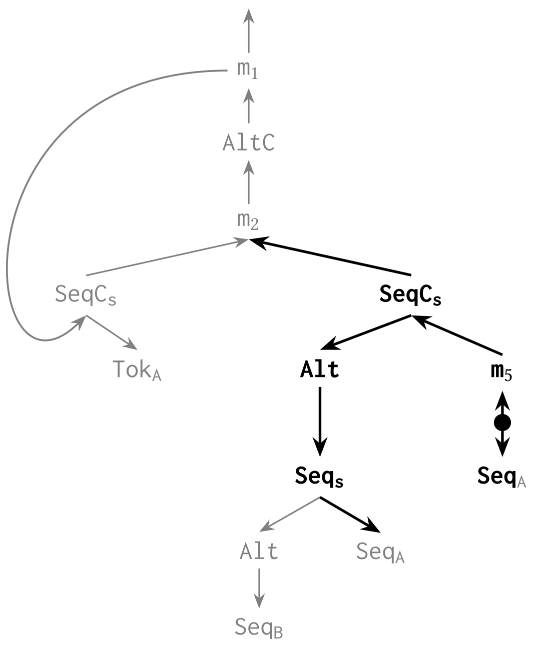


Fig. 19. Zipper after Figure 18c consumes the input token A

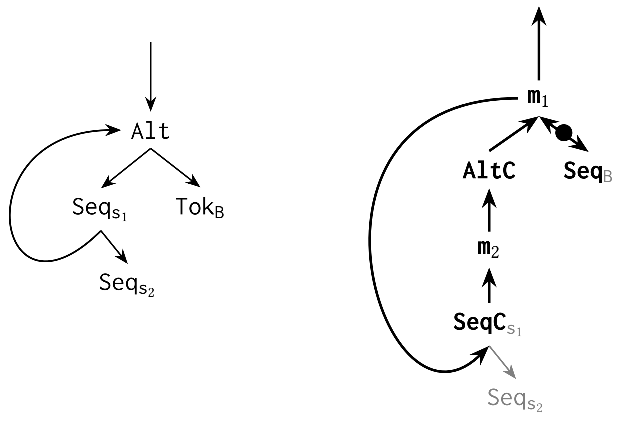
All of this effectively results in unfolding one loop of the cycle. If the next token is another A, another loop will be unfolded to produce Figure 19. In this way, our algorithm transparently handles left recursion by creating a cyclic context that it unfolds only as needed.

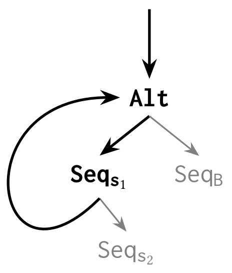
Note that even pathological cases are handled automatically by this algorithm. For example, the grammar in Figure 20a differs from Figure 18a by replacing Tok A with $\mathsf{Seq}_{\mathsf{s}_2}$, a Seq that has no children and thus matches the empty string. As a consequence, when parsing the string containing only B, the final parse result should contain trees for every possible number of recursions through $\mathsf{Seq}_{\mathsf{s}_1}$ (i.e., this part of the grammar is non-expanding). Many parsing algorithms have trouble with this and consider such a grammar to be malformed, but our algorithm automatically handles this case. First, the derivative taken with respect to B produces the zipper in Figure 20b. This zipper has replaced Tok_B with Seq_B and has a loop in its context. That loop represents the indeterminate number of times the parse could have traversed through $\mathsf{Seq}_{\mathsf{s}_1}$. When the next token is parsed, the zipper moves up and out of this part of the grammar and produces the final parse tree in Figure 20c, where the loop in this result represents the arbitrary number of recursions through $\mathsf{Seq}_{\mathsf{s}_1}$ that could have been traversed.

The algorithm is now functionally complete and can handle any grammar or input string. In the next section, we improve performance by removing the memoization tables.

8 ELIMINATING MEMOIZATION TABLES

The algorithm in Figure 17 involves three table lookups. One is `mems.get(p, e)` in $\mathsf{d}_\downarrow$. The other two are `m.result.get(p)` in both the Some clause of $\mathsf{d}_\downarrow$ and the AltC clause of $\mathsf{d}'_\uparrow$. The use of these tables can be costly in terms of both time and space. Fortunately, we can eliminate these tables if we are willing to make exp be mutable.





(a) Initial grammar (b) Grammar after parsing B (c) Final parse tree

Fig. 20. Parsing the string B with a grammar that has a non-expanding production (i.e., $\texttt{e}_1$ ::= $\texttt{e}_1$ $\texttt{e}_2$ | B; $\texttt{e}_2$ ::= ε)

All three table lookups are indexed by p, the current input position. Furthermore, our algorithm never starts parsing a token until it has finished with the preceding token. This means we can throw away table entries for input positions older than the current position, as they are no longer needed. Thus, since `m.result.get(p)` is indexed by only p, we can choose to keep only the most recent entry. Thus in Figure 21, instead of keeping a table in m, we store p as `m.end` along with its corresponding result exp as `m.result`. Whenever parsing a new token causes a new result to be produced, the existing entry is overwritten with the new exp. The p that we store in `m.end` allows us to know whether the result exp is for the current input token.

This also eliminates the p argument in `mems.get(p, e)`, but this lookup is still indexed by e. Fortunately, instead of a table indexed by e, we can store each entry as a mutable field *inside* the exp to which it corresponds.

This leads to the types and values in Figure 21. The exp type is renamed to `exp'`. The new exp type stores a mem record along with an `exp'`. The `start` field of that mem is the input position for which the mem is valid. If `start` is at any point not equal to the current input position, the entry is considered stale and should be ignored.

Likewise, in mem, the `result` mapping is replaced by a `result` field containing only the most recent value, and end is the input position for which `result` is valid. As with `start`, if end is not equal to the current input position, then `result` is stale and should be ignored.

Finally, we define $\texttt{m}_\bot$, which is used by $\texttt{d}_\uparrow$ when creating a new exp and represents when the mems table has no entries for that exp. Because $\texttt{m}_\bot$`.start` does not match the pos of any input token, $\texttt{m}_\bot$ is always considered stale and is replaced with a fresh mem whenever the parser reaches an exp containing it.

These changes produce the final version of our algorithm in Figure 22, which contains no table lookups but can still handle any grammar without exponential time complexity.

```
type exp = { mutable m : mem; e' : exp' }
and exp' = Tok of tok
         | Seq of sym * exp list
         | Alt of (exp list) ref
and cxt  = TopC
         | SeqC of mem * sym *  exp list * exp list
         | AltC of mem
and mem  = { start : pos;
             mutable parents : cxt list;
             mutable end : pos;
             mutable result : exp }

type zipper = exp' * mem

let rec e⊥ = { m = m⊥; e' = Alt (ref []) }
    and m⊥ = { start = p⊥; parents = []; end = p⊥; result = e⊥ }
```

Fig. 21. Types and values for the derivative using zippers without memoization tables. Changes relative to Figure 16 in bold. Except for highlighting, this figure is identical to Figure 2.

9 DISCUSSION

9.1 Differences from Traditional PwD

Aside from the use of a zipper and developing the main technical contribution of this paper (i.e., how to get zippers to work on the shared regions, cycles, and alternates found in CFGs), we note that our algorithm has two differences from the traditional PwD algorithm.

First, the derivative in Brzozowski [1964], Might et al. [2011], and Adams et al. [2016] checks whether the first child of a Seq is a grammar that accepts the empty string (i.e., that child is "nullable"). If the first child is nullable, then the derivative traverses into both the first child *and* the second child of the parent Seq. This is because a nullable grammar corresponds to a successful parse, though it may be possible to parse it further (e.g., due to repetitions or certain kinds of alternates). Thus the derivative also traverses into the second child. Our technique does not check for nullability, though it achieves the same result. Instead of explicitly checking whether the first child is nullable before attempting to traverse into it, PwZ simply traverses into it. If that child is already nullable, our algorithm will automatically continue to the second child.

Second, we use n-ary grammars instead of the binary grammars used by Brzozowski [1964], Might et al. [2011], and Adams et al. [2016]. By this, we mean that our Seq and Alt constructors can have any number of children instead of requiring exactly two. A consequence of this is that where Might et al. [2011] and Adams et al. [2016] use dedicated Epsilon and Empty constructors to represent grammars that accept only the empty string and no strings, respectively, we use childless Seq and Alt constructors, respectively. We do this for two reasons. First, this change simplifies the memoization of shared contexts in Section 6 as well as the handling of the Alt that can be introduced by taking the derivative of a Seq. Second, our benchmarks in Section 10 show that that our algorithm performs 1.4 times faster when using n-ary grammars instead of binary grammars. (A similar speed improvement is *not* seen with traditional PwD.)

9.2 Zipperness

The original zipper [Huet 1997] was designed for traversing trees. However, in this paper we have extended the zipper to support alternates, shared expressions, shared contexts, and cycles, so it is fair to ask whether our data structure is still a zipper. We argue that ours is one possible generalization of zippers, though there may be other generalizations.

```
let derive (p : pos) (t : tok) ((e', m) : zipper) : zipper list =
  let rec d↓ (c : cxt) (e : exp) : zipper list =
    if p == e.m.start
    then (e.m.parents <- c :: e.m.parents;
          if p == e.m.end then d′↑ e.m.result c else [])
    else let m = { start = p; parents = [c]; end = p⊥; result = e⊥ } in
         e.m <- m;
         d′↓ m e.e'

  and d′↓ (m : mem) (e' : exp') : zipper list =
    match e' with
    | Tok (t')         -> if t = t' then [(Seq (t, []), m)] else []
    | Seq (s, [])      -> d↑ (Seq (s, [])) m
    | Seq (s, e :: es) -> let m' = { start = m.start; parents = [AltC m];
                                     end = p⊥; result = e⊥ } in
                          d↓ (SeqC (m', s, [], es)) e
    | Alt (es)         -> List.concat (List.map (d↓ (AltC m)) !es)

  and d↑ (e' : exp') (m : mem) : zipper list =
    let e = { m = m⊥; e' = e' } in
    m.end <- p;
    m.result <- e;
    List.concat (List.map (d′↑ e) m.parents)

  and d′↑ (e : exp) (c : cxt) : zipper list =
    match c with
    | TopC                          -> []
    | SeqC (m, s, es, [])           -> d↑ (Seq (s, List.rev (e :: es))) m
    | SeqC (m, s, es_L, e_R :: es_R) -> d↓ (SeqC (m, s, e :: es_L, es_R)) e_R
    | AltC (m)                      -> if p == m.end
                                       then match m.result.e' with
                                            | Alt (es) -> es := e :: !es; []
                                       else d↑ (Alt (ref [e])) m

  in d↑ e' m
```

Fig. 22. Code for the derivative using zippers without memoization tables. Changes relative to Figure 17 in bold. Except for highlighting, this figure is identical to Figure 1.

Traditional zippers have a plug operation that successively traverses up the context of a zipper until it reaches the top and then returns the resulting expression. This operation can be thought of as "forgetting" the zipper's position, meaning it converts zippers to expressions by reconstructing the material in their contexts. We can implement an equivalent operation for our zippers as shown in Figure 23. Note that the argument p must be a fresh pos that has not previously been used. Also note the similarity to $\mathtt{d}_\uparrow$ and $\mathtt{d}'_\uparrow$ in derive. This is because in both cases, the code is traversing up the grammar. Applying the plug function after taking derivatives with respect to some sequence of tokens produces an expression equivalent (modulo the difference in Section 9.1) to the result produced by Might et al. [2011] and Adams et al. [2016] after taking the derivative with respect to the same tokens. Thus, through the lens of plug, our zipper is comparable to the traditional zipper.

Traditional zippers also allow arbitrarily navigating up, down, left, or right from any given zipper. Although our algorithm only moves left-to-right, our zipper could be navigated the same as regular zippers. However, we did not develop this functionality because it would have no use within the context of our algorithm.

```
let plug (p : pos) (zs : zipper list) : exp list =
  let rec pl (e' : exp') (m : mem) : exp list =
    let e = { m = m⊥; e' = e' } in
    m.end <- p;
    m.result <- e;
    List.concat (List.map (pl' e) m.parents)

  and pl' (e : exp) (c : cxt) : exp list =
    match c with
    | TopC                  -> [e]
    | SeqC (m, s, esL, esR) -> pl (Seq (s, (List.rev esL) @ (e :: esR))) m
    | AltC (m)              -> if p == m.end
                               then match m.result.e' with
                                    | Alt (es) -> es := e :: !es; []
                               else pl (Alt (ref [e])) m

  in List.concat (List.map (fun (e', m) -> pl e' m) zs)
```

Fig. 23. Code for plugging a zipper

Lastly, our algorithm utilizes memoization based on parsed input positions (i.e., pos). It is not clear how this generalizes to non-parsing contexts, but we do not believe this precludes our result from being considered a form of zipper.

Therefore, we claim that our zipper is one generalization of zippers, but other problem domains may require other generalizations of zippers.

9.3 Asymptotic Complexity

We conjecture that our algorithm is cubic in the length of the input parsed assuming no Seq in the initial grammar has more than two children.[2] However, we have not been able to prove it so. For example, we have measured the number of recursive calls when parsing the highly ambiguous grammar `e ::= A | e e` on different lengths of input. This grammar triggers worst-case behavior in most parsing algorithms. The results shown in Figure 24 perfectly match a cubic fitting line.[3] Thus while this test is not proof, it suggests that our algorithm is indeed cubic.

Adams et al. [2016] proved that PwD is cubic by uniquely labeling all allocations and showing that the number of possible labels is cubic. Our algorithm is designed to be a version of PwD that elides the extra traversals in PwD, so a similar analysis might apply to our algorithm. However, there are two complications. First, memoization and in particular the mutable `parents` field in the `mem` type mean the analysis must take account of mutation. Second, unlike PwD where traversals up the grammar are merely the return path of function calls that traverse down the grammar, traversals down the grammar are in $\mathtt{d}_\downarrow$ while their corresponding traversals up the tree are split into the separate $\mathtt{d}_\uparrow$ function.

These complications have stymied our attempt at proving the asymptotic bound of our algorithm, though we hope future work will succeed at doing so.

10 BENCHMARKS

We tested the performance of our parsing algorithm by parsing the files in the Python Standard Library, version 3.4.3 [Python Software Foundation 2015a], using a grammar derived from the

[2]This assumption is required by most parsing algorithms in order for them to be cubic, and there are a number of standard algorithms for converting grammars into such a form.
[3]Specifically the curve $\frac{1}{2}x^3 + 2.5x^2 + 11x + 9$ though we must caution that the constants in this polynomial are dependent on the size of the grammar.

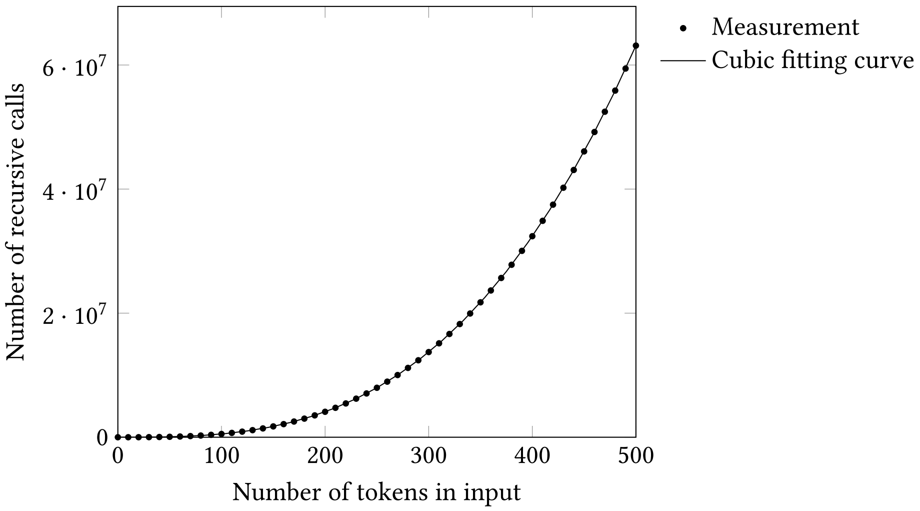


Fig. 24. Number of recursive calls when parsing `e ::= A | e e`

Python 3.4.3 specification [Python Software Foundation 2015b]. The Python Standard Library includes 659 Python files, which have sizes of up to 26,125 tokens.

For efficiency reasons, the version of our parser used for benchmarking adds zippers to a global work list instead of returning lists of zippers as is done in Figure 1. We did not benchmark the algorithms in the other figures as they are intermediate forms and are only for expository purposes.

We also implemented a one-token lookahead and benchmarked our parser both with and without this lookahead. This lookahead was implemented by adding a boolean array to each `exp`. This array is indexed by token types and is true at a particular index if the token type for that index can appear as the first token of that `exp` or the `exp` accepts the empty string and that token type can follow that `exp`. These arrays are pre-computed for the initial grammar, and thus the time to compute these arrays is not included as part of the parse times in our benchmarks.

We compared the performance of our algorithm to PwD [Might et al. 2011], optimized PwD [Adams et al. 2016], Menhir [Pottier and Régis-Gianas 2019], and `dypgen` [Onzon 2012]. Note that Menhir and `dypgen` are parser generators, which gives them a performance advantage over the other implementations.

The PwD implementations are based on versions provided by the authors of Might et al. [2011] and Adams et al. [2016] that we ported from Racket to OCaml. Menhir uses LR(k) parsing, and Dypgen uses GLR parsing. Both are well-known parsing systems for OCaml. To accommodate Menhir's use of LR(k), we reformulated 10 (out of 82) productions in the Python grammar. Including all helper and driver functions, PwD [Might et al. 2011], optimized PwD [Adams et al. 2016], and our algorithm were 196, 238, and 83 lines of OCaml code, respectively.

We ran the tests with OCaml 4.05.0, Menhir version 20200211, and `dypgen` version 20120619. These were run on a Digital Ocean dedicated-CPU "general purpose" machine. It had a 2.30 GHz Intel Xeon Gold 6140 CPU with 8GB of RAM and ran Ubuntu 20.04 LTS. We tokenized files in advance and loaded those tokens into memory before benchmarking started, so that only parsing time was measured when benchmarking. We used the `core_bench` library [Jane Street 2014] to measure timings. PwD was so slow that we timed out any parse longer than 72 seconds. The largest input it parsed without timing out was 439 tokens. The final results are presented in Figure 25 and are normalized to measure parse time per input token.

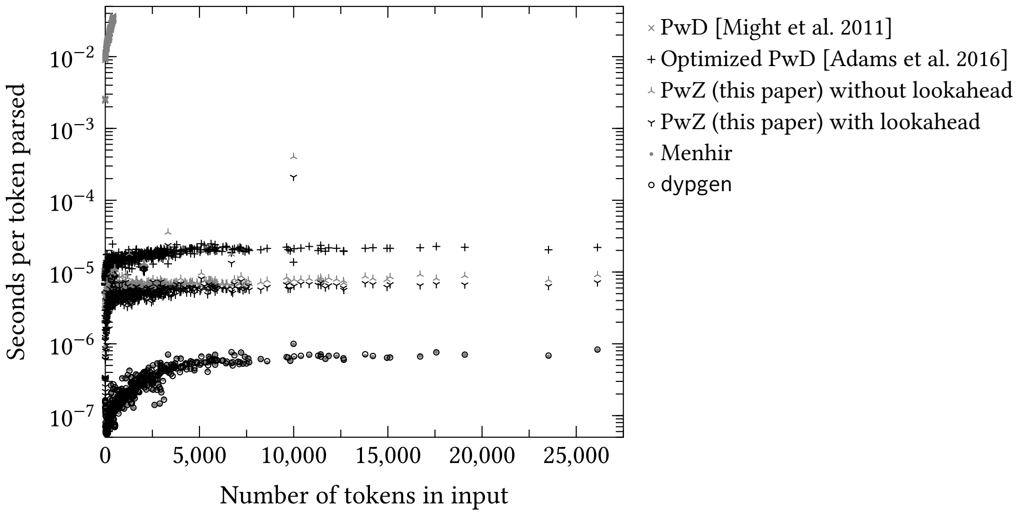


Fig. 25. Performance versus other parsers

Note that PwD shows up only in the upper left corner of Figure 25 around 10^{-2} seconds per token. Also the results for Menhir and dypgen are so similar that many of their symbols (a dot for Menhir and a circle for dypgen) overlap. Finally, the dense part of the graph on the left where the symbols overlap contain what one would extrapolate from the results to the right.

Our parser ran a geometric mean of 1.42 times faster with lookahead than without. With lookahead, our parser ran a geometric mean of 6,500 times faster than PwD, 3.24 times faster than optimized PwD, 24.0 times slower than Menhir, and 24.1 times slower than dypgen.

The relative ease with which our algorithm achieves its performance is notable. This is especially so given that the parser in Adams et al. [2016] requires "compaction" as well as several optimizations and tunings, but our parser surpasses its performance without such effort.

Binary versus n-ary. As mentioned previously, the algorithm we present uses n-ary grammars (i.e., grammars supporting any number of children) whereas Might et al. [2011] and Adams et al. [2016] use binary grammars (i.e., grammars with only two children). This choice is based on two considerations. First, handling the `AltC` introduced when traversing down a `Seq` is more complicated in the binary version of PwZ. Second, the performance measurements in Figure 26 and Figure 27 show that, for PwZ, n-ary is 1.41 times faster than binary. We implemented both binary and n-ary versions of PwZ as well as both PwD [Might et al. 2011] and optimized PwD [Adams et al. 2016]. They were run on the same system and with the same system configuration as the results in Figure 25. Neither PwZ implementation uses lookahead in this comparison.

We note that using n-ary grammars significantly harms the performance of PwD, does not significantly affect the performance of optimized PwD, and slightly helps the performance of PwZ. We do not know why the effect is different in all three cases, but we conjecture that it might be due to an effect similar to that observed by both Might et al. [2011] and Adams et al. [2016], where stacks of intermediate grammar nodes accumulate and thus slow down traversal through the grammar. Due to its multi-way branching, n-ary PwZ is less prone to these sorts of accumulations. Might et al. [2011] and Adams et al. [2016] solve this by a what they call a "compaction" algorithm. In theory, we could implement the same for PwZ, but the interaction with the zipper, contexts, and memoization makes this a non-trivial task. It would be interesting future work to see whether this improves the performance of binary PwZ to match that of n-ary PwZ.

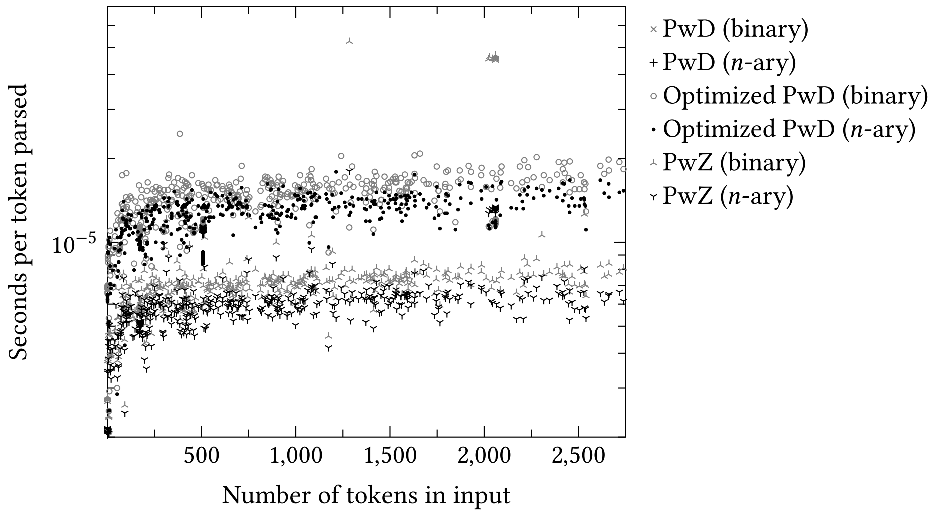


Fig. 26. Performance of binary versus n-ary parsers

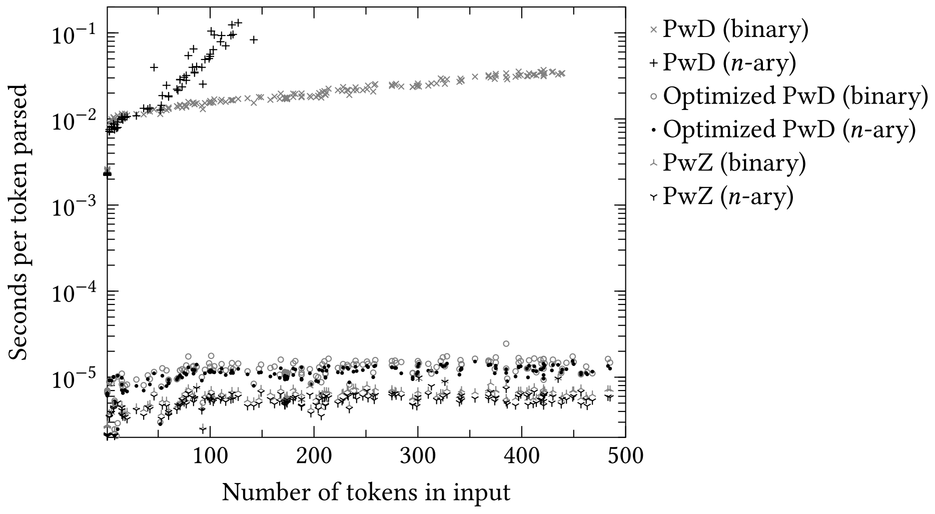


Fig. 27. Figure 26 zoomed out to show PwD

11 RELATED WORK

Parsing with Derivatives. Using derivatives to parse languages was first suggested by Brzozowski [1964] for regular expressions and by both Might et al. [2011] and Danielsson [2010] for context-free grammars (CFGs). For regular expressions, derivatives sometimes outperform traditional techniques [Owens et al. 2009]. However, they initially had poor performance for CFGs and were impractical to use [Might et al. 2011]. Later work [Adams et al. 2016] improved the performance of Parsing with Derivatives but requires optimizations and tunings that make the code less elegant. Our algorithm does not require these optimizations but outperforms even the optimized Parsing with Derivatives.

As explained in Section 9.2 and modulo the differences in Section 9.1, calling `plug` at any point in PwZ produces the same grammar that PwD would produce at the same point. Thus we consider our algorithm to be a variant of PwD.

Zippy LL(1) Parsing with Derivatives. Edelmann et al. [2020], a publication concurrent with ours, describes and formally verifies an LL(1) parser that is based on zippers and PwD. The main difference from our work is that our algorithm does full CFG while Edelmann et al. [2020] is restricted to LL(1) grammars. As a consequence, our algorithm has to deal with cycles and ambiguities in a way that Edelmann et al. [2020] does not. This also means Edelmann et al. [2020] can use a traditional zipper, whereas we have to generalize the zipper to handle cycles and ambiguities. On the other hand, the algorithm in Edelmann et al. [2020] is formally verified, while ours is not. We speculate that the proof techniques used in Edelmann et al. [2020] might provide insights about how to formally verify our algorithm.

GLL Parsing. Our algorithm has some similarities to GLL parsing [Scott and Johnstone 2010, 2013]. Both are recursive-decent parsers that handle arbitrary CFGs, and GLL's shared-packed parse forest (SPPF) and graph-structured stack (GSS) are similar in structure to our `exp` and `cxt`, respectively. However, our algorithm is a parser, and GLL is a parser *generator*. The difference is similar to the one between an interpreter (our algorithm) and a compiler (GLL). Nevertheless, a sufficiently smart partial evaluator might be able to transform our algorithm into a parser generator. We suspect that such a partially evaluated algorithm would closely correspond to GLL.

Derivatives of Types. McBride [2001] shows that the context type (e.g., `cxt`) used in a zipper type can be mechanically generated from the type over which the zipper traverses (e.g., `exp`). McBride [2001] calls this taking the derivative of a type. That both Brzozowski [1964] and McBride [2001] call their techniques a "derivative" is not a complete coincidence. They both have the same structure and laws as the partial derivative from calculus. What is surprising is that these two derivatives, which use two different interpretations of `exp` (i.e., as a grammar in Brzozowski [1964] and as a data structure in McBride [2001]), combine so well. A deeper understanding of the connection between them might provide further insights about combining them as we do in this paper.

Clowns and Jokers. McBride [2008] also describes a variant of the one-hole context that uses different types for the values to the left and right of the hole. We could take advantage of this by noting that an `exp` to the left of the current hole never contains a token expression (`Tok`). The technique in McBride [2008] would let the types reflect such a distinction and have one type for the left siblings that does not contain `Tok` and one type for the right siblings that contains `Tok`. We chose not to implement this in favor of presentational simplicity.

Continuations and parsing. Johnson [1995] describes a top-down parsing technique that uses memoized continuations. Like our algorithm, it does before-the-call memoization, and the process of memoizing continuations has a very similar structure to how we manipulate the `mems` table and `mem` records in Figure 17. The `entry-continuations` and `entry-results` in Johnson [1995] correspond, respectively, to the `parents` and `result` fields of `mem` in Figure 16.

However, there are also some differences. The algorithm in Johnson [1995] expects a list of all input tokens from the outset, but our parser parses one token at a time. In addition, the algorithm in Johnson [1995] is merely a recognizer and does not produce parse trees whereas our algorithm does. Johnson [1995] alludes to the possibility of extracting sparse packed parse forests from the memoization tables but says that "a straightforward implementation attempt would probably be very complicated". Finally, Johnson [1995] does not address performance and does not eliminate the memoization tables as we describe in Section 8.

Item Sets. A number of parsing algorithms use item sets to represent their parsing state. These item sets contain items of the form $\langle X \to \alpha \bullet \beta, i\rangle$, where X is a non-terminal, α and β are strings of terminals and non-terminals, and i is an input position. For example, Earley [1970] uses one item set per input position. The item $\langle X \rightarrow \alpha \bullet \beta, i\rangle$ being in the item set for position j means that a parse for the production $X \rightarrow \alpha\beta$ was started at position j, the input between positions j and i is parsed by α, and the production $X \rightarrow \alpha\beta$ is looking for a string that starts at position j and is parsed by β.

These item sets have some similarity to `SeqC` contexts, where α and β are the left and right siblings, i is the `start` field of the `mem` stored in the `SeqC` context, and j is the input position in scope when the `SeqC` context is being processed (i.e., the `p` argument to `derive`).

However, there are a number of differences in how these algorithm operate. PwZ operates by traversing over the grammar, while parsers based on item sets operate by finding the fixed points of item sets for successive input tokens. Also, PwZ represents choices between alternatives explicitly using `Alt` and `AltC` and explicitly represents where a sub-parse is called from using `parent` pointers in its contexts. Parsers based on item sets represent choices between alternatives implicitly by items being in the same item set and implicitly represent where a sub-parse is called from by looking up items in item sets at particular positions.

12 CONCLUSION

Parsing with Derivatives has long suffered from impractically slow performance. Recent improvements [Adams et al. 2016] have mitigated this, but require several optimizations and tunings that make the code less elegant. By introducing a zipper, we achieve good performance without needing such tweaks. The resulting parser is also concise: the algorithm in Figure 22 lacks only a driver loop, yet takes only 31 lines of code. Despite its brevity, our algorithm is 6,500 times faster than PwD [Might et al. 2011] and 3.24 faster than optimized PwD [Adams et al. 2016].

This generalization of zippers is specific to CFGs but suggests that it may be possible to generalize the zipper to other data structures that have shared regions, cycles or choices between alternates. We encourage future work that explores these possibilities.

A ARTIFACT

The artifact for this paper is both an embedded file in this document and available on the web at `https://github.com/pdarragh/parsing-with-zippers-paper-artifact`. The version embedded in this document is commit `c5c0830`, which has tag `icfp-2020-artifact`. It includes a full implementation, several example grammars used as stress tests, and the benchmarks used in Section 10. To save a copy right click the following filename and select "Save Attachment As...".

Artifact File: `parsing-with-zippers-paper-artifact.zip`

B SOURCE CODE

This appendix contains code for the algorithm described by this paper in an ASCII form suitable for copying and pasting into a source file. Due to limitations of LaTeX and PDF, such a copy will not preserve indentation. Thus, as an alternative, you can right click a file name and select "Save Attachment As..." to save a properly indented version of the code. This code is split into separate files for presentation purposes, but it can all be placed in the same source file if the `open` statements at the top of each file are removed. Also, the types in `pwz_abstract_types.ml` can easily be changed as the rest of the code treats them as abstract.

B.1 `pwz_abstract_types.ml`

```
type pos = int ref (* Using ref makes it easy to create values that are not pointer equal *)
let p_bottom = ref (-1)

type sym = string
```

```
let s_bottom = "<s_bottom>"

type tok = string
let t_eof = "<t_eof>"
```

B.2 pwz_types.ml

```
open Pwz_abstract_types

type exp = { mutable m : mem; e' : exp' }
and exp' = Tok of tok
         | Seq of sym * exp list
         | Alt of (exp list) ref
and cxt  = TopC
         | SeqC of mem * sym * exp list * exp list
         | AltC of mem
and mem  = { start_pos : pos;
             mutable parents : cxt list;
             mutable end_pos : pos;
             mutable result : exp }

type zipper = exp' * mem

let rec e_bottom = { m = m_bottom; e' = Alt (ref []) }
    and m_bottom = { start_pos = p_bottom; parents = []; end_pos = p_bottom; result = e_bottom }
```

B.3 pwz_derive.ml

```
open Pwz_abstract_types
open Pwz_types

let derive (p : pos) (t : tok) ((e', m) : zipper) : zipper list =
  let rec d_d (c : cxt) (e : exp) : zipper list =
    if p == e.m.start_pos
    then (e.m.parents <- c :: e.m.parents;
          if p == e.m.end_pos then d_u' e.m.result c else [])
    else (let m = { start_pos = p; parents = [c]; end_pos = p_bottom; result = e_bottom } in
          e.m <- m;
          d_d' m e.e')

  and d_d' (m : mem) (e' : exp') : zipper list =
    match e' with
    | Tok (t')         -> if t = t' then [(Seq (t, []), m)] else []
    | Seq (s, [])      -> d_u (Seq (s, [])) m
    | Seq (s, e :: es) -> let m' = { start_pos = m.start_pos; parents = [AltC m];
                                     end_pos = p_bottom; result = e_bottom } in
                          d_d (SeqC (m', s, [], es)) e
    | Alt (es)         -> List.concat (List.map (d_d (AltC m)) !es)

  and d_u (e' : exp') (m : mem) : zipper list =
    let e = { m = m_bottom; e' = e' } in
    m.end_pos <- p;
    m.result <- e;
    List.concat (List.map (d_u' e) m.parents)

  and d_u' (e : exp) (c : cxt) : zipper list =
    match c with
    | TopC                          -> []
    | SeqC (m, s, es, [])           -> d_u (Seq (s, List.rev (e :: es))) m
    | SeqC (m, s, es_L, e_R :: es_R) -> d_d (SeqC (m, s, e :: es_L, es_R)) e_R
    | AltC (m)                      -> if p == m.end_pos
                                       then match m.result.e' with
                                            | Alt (es) -> es := e :: !es; []
                                            | _        -> failwith "Not an Alt."
                                       else d_u (Alt (ref [e])) m
  in d_u e' m
```

B.4 pwz_driver.ml

```
open Pwz_abstract_types
open Pwz_types
open Pwz_derive

let init_zipper (e : exp) : zipper =
  let e'    = Seq (s_bottom, []) in
  let m_top = { start_pos = p_bottom; parents = [TopC]; end_pos = p_bottom; result = e_bottom } in
  let c     = SeqC (m_top, s_bottom, [], [e; { m = m_bottom; e' = Tok t_eof }]) in
  let m_seq = { start_pos = p_bottom; parents = [c]; end_pos = p_bottom; result = e_bottom } in
  (e', m_seq)

let unwrap_top_zipper ((e', m) : zipper) : exp =
  match m.parents with
  | [SeqC ({ parents = [TopC] }, s_bottom, [e; _], [])] -> e
  | _ -> failwith "Invalid top zipper."

let parse (ts : tok list) (e : exp) : exp list =
  let rec parse' (p : pos) (ts : tok list) (z : zipper) : zipper list =
    match ts with
    | []       -> derive p t_eof z
    | t :: ts' -> List.concat (List.map (fun z' -> parse' (ref (!p + 1)) ts' z') (derive p t z)) in
  List.map unwrap_top_zipper (parse' (ref 0) ts (init_zipper e))

type ast = Ast of sym * ast list

let list_product (l1 : 'a list) (l2 : ('a list) list) : ('a list) list =
  List.concat (List.map (fun l -> List.map (List.cons l) l2) l1)

let rec ast_list_of_exp (e : exp) : ast list =
  match e.e' with
  | Tok _       -> []
  | Seq (l, es) -> List.map (fun es' -> Ast (l, es'))
                     (List.fold_right list_product (List.map ast_list_of_exp es) [[]])
  | Alt (es)    -> List.concat (List.map ast_list_of_exp !es)

let ast_list_of_exp_list (es : exp list) : ast list =
  List.concat (List.map ast_list_of_exp es)
```

B.5 pwz_plug.ml

```
open Pwz_abstract_types
open Pwz_types

let plug (p : pos) (zs : zipper list) : exp list =
  let rec pl (e' : exp') (m : mem) : exp list =
    let e = { m = m_bottom; e' = e' } in
    m.end_pos <- p;
    m.result <- e;
    List.concat (List.map (pl' e) m.parents)

  and pl' (e : exp) (c : cxt) : exp list =
    match c with
    | TopC                    -> [e]
    | SeqC (m, s, es_L, es_R) -> pl (Seq (s, (List.rev es_L) @ (e :: es_R))) m
    | AltC (m)                -> if p == m.end_pos
                                 then match m.result.e' with
                                      | Alt (es) -> es := e :: !es; []
                                      | _        -> failwith "Not an Alt."
                                 else pl (Alt (ref [e])) m

  in List.concat (List.map (fun (e', m) -> pl e' m) zs)
```

REFERENCES

Michael D. Adams, Celeste Hollenbeck, and Matthew Might. 2016. On the complexity and performance of parsing with derivatives. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Santa Barbara, CA, USA) *(PLDI '16)*. ACM, New York, NY, USA, 224–236. https://doi.org/10.1145/2908080.2908128

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

Nils Anders Danielsson. 2010. Total parser combinators. In *Proceedings of the 15th ACM SIGPLAN international conference on Functional programming* (Baltimore, Maryland, USA) *(ICFP '10)*. ACM, New York, NY, USA, 285–296. https://doi.org/10.1145/1863543.1863585

Jay Earley. 1970. An efficient context-free parsing algorithm. *Communications of the ACM (CACM)* 13, 2 (Feb. 1970), 94–102. https://doi.org/10.1145/362007.362035

Romain Edelmann, Jad Hamza, and Viktor Kunčak. 2020. Zippy LL(1) Parsing with Derivatives. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation* (London, UK) *(PLDI '20)*. ACM, New York, NY, USA, 1036–1051. https://doi.org/10.1145/3385412.3385992

Gérard Huet. 1997. The Zipper. *Journal of Functional Programming* 7, 05 (Sept. 1997), 549–554. https://doi.org/10.1017/S0956796897002864

Jane Street. 2014. core_bench. https://github.com/janestreet/core_bench version 109.58.01.

Mark Johnson. 1995. Memoization in top-down parsing. *Computational Linguistics* 21, 3 (Sept. 1995), 405–417. http://dl.acm.org/citation.cfm?id=216261.216269

Xavier Leroy, Damien Doligez, Alain Frisch, Jacques Garrigue, Didier Rémy, and Jérôme Vouillon. 2020. The OCaml system: release 4.10. https://ocaml.org/releases/4.10/htmlman/

Conor McBride. 2001. The Derivative of a Regular Type is its Type of One-Hole Contexts. strictlypositive.org/diff.pdf

Conor McBride. 2008. Clowns to the left of me, jokers to the right (pearl): dissecting data structures. In *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (San Francisco, California, USA) *(POPL '08)*. ACM, New York, NY, USA, 287–295. https://doi.org/10.1145/1328438.1328474

Matthew Might, David Darais, and Daniel Spiewak. 2011. Parsing with derivatives: a functional pearl. In *Proceedings of the 16th ACM SIGPLAN International Conference on Functional Programming* (Tokyo, Japan) *(ICFP '11)*. ACM, New York, NY, USA, 189–195. https://doi.org/10.1145/2034773.2034801

Emmanuel Onzon. 2012. dypgen: Self-extensible parsers and lexers for OCaml. http://dypgen.free.fr/ version 20120619.

Scott Owens, John Reppy, and Aaron Turon. 2009. Regular-expression derivatives re-examined. *Journal of Functional Programming* 19, 02 (March 2009), 173–190. https://doi.org/10.1017/S0956796808007090

François Pottier and Yann Régis-Gianas. 2019. Menhir. http://gallium.inria.fr/~fpottier/menhir/ version 20190626.

Python Software Foundation. 2015a. Python 3.4.3. https://www.python.org/downloads/release/python-343/

Python Software Foundation. 2015b. The Python Language Reference: Full Grammar specification. https://docs.python.org/3/reference/grammar.html

Elizabeth Scott and Adrian Johnstone. 2010. GLL Parsing. *Electronic Notes in Theoretical Computer Science* 253, 7 (Sept. 2010), 177–189. https://doi.org/10.1016/j.entcs.2010.08.041

Elizabeth Scott and Adrian Johnstone. 2013. GLL parse-tree generation. *Science of Computer Programming* 78, 10 (Oct. 2013), 1828–1844. https://doi.org/10.1016/j.scico.2012.03.005