Bachelor thesis
Computer Science

Radboud University

Parsing with derivatives in Haskell

Author:
Timo Maarse

First supervisor:
prof. dr. Ralf Hinze
`ralf@cs.ru.nl`

Second assessor:
dr. Peter Achten
`p.achten@cs.ru.nl`

January 14, 2018

Abstract

We show how to implement a parser combinator library in Haskell capable of parsing arbitrary context-free grammars. The library is based on the work of Might et al. on parsing with derivatives, [15] of which we derive the required results from a single Galois connection. The library improves upon the previous Haskell implementation [5] with regards to completeness, safety and documentation; we can handle any grammar and use generalized tries for memoization instead of unsafe IO. However, our implementation shows severe performance issues. We describe these issues and possible solutions where applicable.

Contents

1 Introduction **2**

2 Parsing with derivatives **4**
- 2.1 Introduction 4
- 2.2 Recursive regular expressions 5
- 2.3 Calculation of derivatives 6
- 2.4 Empty word in a language 7
- 2.5 Example 8

3 Haskell Implementation **9**
- 3.1 Recognition 9
  - 3.1.1 Basics 9
  - 3.1.2 Tagging 11
  - 3.1.3 Empty word in a language 12
- 3.2 Parse forest 14
  - 3.2.1 Parser combinators 14
  - 3.2.2 Derivatives of parsers 15
  - 3.2.3 Parsing the empty word 16
- 3.3 Various improvements 18
  - 3.3.1 Memoization of the derivative 18
  - 3.3.2 Tagging nodes automatically 20
  - 3.3.3 Compaction 21
- 3.4 Performance 22
  - 3.4.1 Complexity 22
  - 3.4.2 Benchmark 23
  - 3.4.3 Analysis 24

4 Related Work **26**

5 Conclusions **27**

A Preliminaries **30**

B Code **31**

Chapter 1

Introduction

Parsing is a thoroughly researched subject in computer science; developments in parsing methods have, for example, allowed us to construct high-level programming languages. Many algorithms are known for parsing, all of which have different properties regarding the class of languages or grammars they can handle, and different complexities.

Partially orthogonal to the parsing algorithm used is the method of constructing the parser. Parsers are usually constructed either manually, using parser generators, or using parser combinators. Parser generators are especially useful to construct parsers using very complex algorithms that are hard to write by hand. The advantage of parser combinators compared to generators is that parser combinators can leverage language features; we can, for example, construct parsers programmatically or construct an abstract syntax tree native to the host language while parsing. In essence, they form an embedded domain specific language, in contrast to standalone ones like those used by generators.

For the purpose of parsing in functional programming languages, parser combinator libraries are popular and well-known. These libraries traditionally implement recursive descent parsing. However, recursive descent parsers cannot handle left-recursive grammars such as:

$$T \to T \text{ ``+'' } T \mid a$$

Nevertheless, left-recursive grammars often provide a natural way to express a language. It is therefore of interest to have methods that allow parsing such grammars, and preferably using simple algorithms. Parser combinator libraries can be implemented to handle left-recursion, [8] but these do not follow from a clear mathematical theory, and in turn their correctness is harder to prove.

Parsing based on parsing with derivatives [15] can be used to implement parser combinator libraries that can parse arbitrary context-free grammars. We are specifically interested in a Haskell implementation as Haskell is a relatively widely used language, and has different properties than Racket, the main language used by the authors of [15]. The authors also provide a Haskell implementation [5], but it has the following issues:

1. It cannot handle, for example, the following grammar:

   $$A \to AA \mid a$$

   However, the grammar is perfectly valid and describes the language denoted by the regular expression aa^*. We will show how it fails due to Haskell's referential transparency and show a solution using tagging.

2. It uses unsafe features of Haskell for memoization; unsafe features of Haskell should be avoided where possible. We show how to implement memoization in a safe manner for arbitrary data structures using generalized tries.

3. It lacks documentation as to how it works; it is not a straight-forward implementation of their paper.

Chapter 2

Parsing with derivatives

2.1 Introduction

This chapter will describe most of the required theory from the original paper about parsing with derivatives. [15]

Let us start by reiterating the notion of recognition: we read a word and want to determine whether it is contained in a certain language. Consider checking whether the word *abba* is contained in the language A. Suppose now that we have read a, so that we are left with *bba*. Does a language $\partial_a\, A$ exist, such that $bba \in \partial_a\, A$ if and only if $abba \in A$? If we can calculate such a language, we can recursively calculate a language L such that it contains the empty word if and only if $abba \in A$. We can then check whether $\varepsilon \in L$ to determine whether $abba \in A$. By asking the question, we have practically answered it already:

$$w \in \partial_c\, A \Longleftrightarrow cw \in A$$

defines the language $\partial_c A$ uniquely. To see why, we consider the following well-known Galois connection [2]:

Definition 1. Let A and L be languages over an alphabet Σ, then we can define $L \mapsto A \setminus L$ as follows:

$$L_1 \subseteq A \setminus L_2 \Longleftrightarrow A \circ L_1 \subseteq L_2 \text{ for all } L_1, L_2 \subseteq \Sigma^*$$

The expression $A \setminus L$ is called a right factor of the language L.[1]

Then with $L_1 = \{\varepsilon\}$ and $A = \{c\}$ for some letter $c \in \Sigma$, we have exactly the desired result called Brzozowski's derivative [4]:

$$w \in \{c\} \setminus L = \partial_c\, L \Longleftrightarrow cw \in L$$

[1]It is well-defined as each set of the form $\{L_1 \subseteq \Sigma^* \mid A \circ L_1 \subseteq L_2\}$ has a greatest element (the union of all its elements is again an element).

To express whether a language contains the empty word, we use the nullability map δ given by:

$$\delta(L) = \begin{cases} \varepsilon & \text{if } \varepsilon \in L \\ \varnothing & \text{if } \varepsilon \notin L \end{cases}$$

As an example, from these definitions we see that $aa \in \{a, aa, ba, ab\}$:

$$\delta(\partial_a(\partial_a \{a, aa, ba, ab\})) = \delta(\partial_a \{\varepsilon, a\}) = \delta(\{\varepsilon\}) = \varepsilon$$

However, it is not immediately clear how to calculate the derivative of an arbitrary context-free language; manipulating grammars does not seem particularly attractive. The following section will introduce a better alternative.

2.2 Recursive regular expressions

While context-free languages are traditionally defined using a context-free grammar, we will define a context-free grammar as a system of recursive regular expressions, for example:

$$\begin{aligned} A &= aBa + \varepsilon \\ B &= bAb \end{aligned}$$

We need to define how to interpret the languages defined by this system of equations; we define them as the least fixed point of the equations seen as a vector-valued function:

$$f \begin{pmatrix} A \\ B \end{pmatrix} = \begin{pmatrix} aBa + \varepsilon \\ bAb \end{pmatrix}$$

It is easily proven that each individual component preserves directed suprema using induction, and therefore the existence of the fixed point follows from corollary 3 in appendix A. The languages defined by such least fixed points are exactly the context-free languages. [2]

We will restrict the regular expressions allowed in the recursive equations to regular expressions without the Kleene star operation, as the result of this operation can always be expressed using a recursive equation; we can express $L = A^*$ as follows:

$$L = AL \mid \varepsilon$$

2.3 Calculation of derivatives

Now that we have a more compact definition of context-free grammars, we can work out how to calculate the derivative of an arbitrary context-free language L. As our languages are defined using union and concatenation, we only need to recursively calculate the derivative for a few basic structures. We will first show that deriving to whole words can be done like we suggested earlier; derive with respect to a word w by deriving with respect to its head, and then recursively to its tail:

$$\begin{aligned} a \in \partial_{cw}\, L &\Leftrightarrow (cw)a \in L \\ &\Leftrightarrow c(wa) \in L \\ &\Leftrightarrow wa \in \partial_c\, L \\ &\Leftrightarrow a \in \partial_w\, (\partial_c\, L) \end{aligned}$$

So we have $\partial_{cw}\, L = \partial_w\, (\partial_c\, L)$ like expected.

To calculate the derivative of L with respect to a single letter, we will recursively calculate the derivative with respect to c based on the form of L:

- Let $L = \varnothing$, and suppose $w \in \partial_c\, L$, then $cw \in L$ which is false, so $\partial_c\, L = \varnothing$.
- Let $L = \varepsilon$, and suppose $w \in \partial_c\, L$, then $cw \in L$ which is again false, so $\partial_c\, L = \varnothing$.
- Let $L = \{c\}$, and $w \in \partial_c\, L$, then $cw \in L \Rightarrow w = \varepsilon$, so $\partial_c\, L = \{\varepsilon\}$.
- Let $L = \{c'\}$, with $c \neq c'$, and $w \in \partial_c\, L$. Then $cw \in L$ which is false, so $\partial_c\, L = \varnothing$.
- Let $L = L_1 \cup L_2$, then we have for all words w:

  $$\begin{aligned} w \in \partial_c\, (L_1 \cup L_2) &\Leftrightarrow cw \in L_1 \cup L_2 \\ &\Leftrightarrow cw \in L_1 \text{ or } cw \in L_2 \\ &\Leftrightarrow w \in \partial_c\, L_1 \text{ or } w \in \partial_c\, L_2 \\ &\Leftrightarrow w \in \partial_c\, L_1 \cup \partial_c\, L_2 \end{aligned}$$

  Therefore $\partial_c\, (L_1 \cup L_2) = \partial_c\, L_1 \cup \partial_c\, L_2$, in other words, the derivative of the union is the union of the derivatives.
- Let $L = L_1 \circ L_2$, then we have for all words w:

$$
\begin{aligned}
&w \in \partial_c\,(L_1 \circ L_2) \\
\Leftrightarrow\ & cw \in L_1 \circ L_2 \\
\Leftrightarrow\ & \begin{cases} cw_1 \in L_1 \text{ and } w_2 \in L_2 \text{ for } w_1 \text{ and } w_2 \text{ such that } w = w_1w_2\text{, or} \\ \varepsilon \in L_1 \text{ and } cw \in L_2 \end{cases} \\
\Leftrightarrow\ & \begin{cases} w_1 \in \partial_c\, L_1 \text{ and } w_2 \in L_2 \text{ for } w_1 \text{ and } w_2 \text{ such that } w = w_1w_2\text{, or} \\ \varepsilon \in L_1 \text{ and } w \in \partial_c\, L_2 \end{cases} \\
\Leftrightarrow\ & \begin{cases} w \in \partial_c\, L_1 \circ L_2\text{, or} \\ w \in \delta(L_1) \circ \partial_c\, L_2 \end{cases} \\
\Leftrightarrow\ & w \in (\partial_c\, L_1 \circ L_2) \cup (\delta(L_1) \circ \partial_c\, L_2)
\end{aligned}
$$

Therefore we have $\partial_c\,(L_1 \circ L_2) = (\partial_c\, L_1 \circ L_2) \cup (\delta(L_1) \circ \partial_c\, L_2)$.

Now that we can derive with respect to a letter (and therefore a word), we should determine how to check whether a languages contains the empty word.

2.4 Empty word in a language

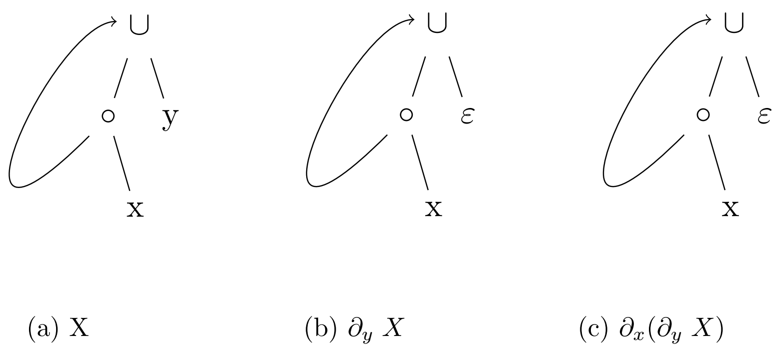
The question of how to calculate $\delta(L)$ arises naturally. It is clear that the following holds from the definition:

$$
\begin{aligned}
\delta(\varnothing) &= \varnothing \\
\delta(\varepsilon) &= \varepsilon \\
\delta(w) &= \varnothing \\
\delta(L_1 \cup L_2) &= \delta(L_1) \cup \delta(L_2) \\
\delta(L_1 \circ L_2) &= \delta(L_1) \cap \delta(L_2)
\end{aligned}
$$

However, consider $L = L \cup L$, then $\delta(L) = \delta(L) \cup \delta(L) = \delta(L)$; so the nullability of L depends on its own nullability. We define δ as the least fixed point of the nullability equations: the equations that the nullability map should satisfy,[2] in this case only $\delta(L) = \delta(L)$. This definition results in the desired value. [3]

[2]Again, the existence of the least fixed point follows from corollary 3 in appendix A; we can see the system of equations as a vector-valued function, and proving each individual component preserves directed suprema is straight-forward using induction.

Figure 2.1: Visualization of recognizing yx



2.5 Example

We have now developed the theory required for recognizing a word. Consider the language defined by the following equation:

$$X = Xx + y$$

We will show that $yx \in X$ by deriving to y, then to x, and finally calculating the nullability:

$$\begin{aligned}
\partial_x(\partial_y X) &= \partial_x(\partial_y(Xx + y)) \\
&= \partial_x(\delta(X)\varepsilon + (\partial_y X)x + \varepsilon) \\
&= \partial_x((\partial_y X)x + \varepsilon) \text{ where } \partial_y X = (\partial_y X)x + \varepsilon \\
&= \partial_x(\partial_y X)x + \delta(\partial_y X)\varepsilon + \varnothing \text{ where } \partial_y X = (\partial_y X)x + \varepsilon \\
&= \partial_x(\partial_y X)x + \varepsilon \\
&\Rightarrow \delta(\partial_x(\partial_y X)) = \varepsilon \\
&\Rightarrow yx \in X
\end{aligned}$$

A visualization is shown in figure 2.1.

Chapter 3

Haskell Implementation

3.1 Recognition

3.1.1 Basics

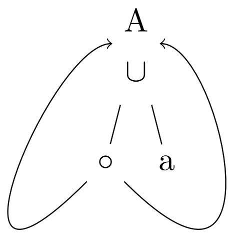
We start with recognizing whether a word is in a certain language. We will first show an approach that is, in a sense, similar to those taken by the authors of the original Haskell implementation; subsequently we will show why it cannot work, and how to modify it such that it does.

Instead of only allowing a language to be nothing, the empty word, a terminal, union or concatenation of languages, we will also allow $\delta(L)$. This is useful to defer the computation of the nullability of L; if a $\delta(L)$ node is derived, it is immediately clear that the result is $\varnothing$, without considering the nullability of L. We will also restrict the value of a terminal language node to a single letter from the alphabet, as it has the same expressive power as allowing words.

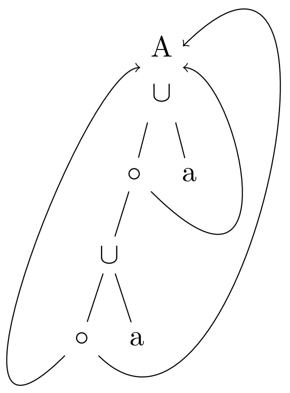
We start off by defining a language:

```
    -- The type ‘t‘’ is the type of token (mathematically an
    -- alphabet), for instance, ‘Char‘, ‘String‘ or more
    -- advanced types.
  data Language t = Empty
    | Epsilon
    | Terminal t
    | Nullable (Language w)
    | Union (Language w) (Language w)
    | Concatenation (Language w) (Language w)
```

Figure 3.1: Illustration of a problem of referential transparency



(a) Definition of A



(b) Definition of A with a self-reference replaced with definition of A

We can then concisely define the derivative using the results from the previous section:

$$
\begin{array}{l}
\textit{derive} :: (\textit{Eq } t) \Rightarrow t \rightarrow \textit{Language } t \rightarrow \textit{Language } t \\
\textit{derive } t \textit{ Empty} = \textit{Empty} \\
\textit{derive } t \textit{ Epsilon} = \textit{Empty} \\
\textit{derive } t \ (\textit{Terminal } a) = \textbf{if } t \equiv a \textbf{ then } \textit{Epsilon} \textbf{ else } \textit{Empty} \\
\textit{derive } t \ (\textit{Nullable } a) = \textit{Empty} \\
\textit{derive } t \ (\textit{Union } a \ b) = \textit{Union } (\textit{derive } t \ a) \ (\textit{derive } t \ b) \\
\textit{derive } t \ (\textit{Concatenation } a \ b) = \textit{Union} \\
\quad (\textit{Concatenation } (\textit{derive } t \ a) \ b) \\
\quad (\textit{Concatenation } (\textit{Nullable } a) \ (\textit{derive } t \ b))
\end{array}
$$

However, we cannot check whether an instance of *Language t* contains the empty word. Consider the grammar from the introduction:

$$A \rightarrow AA \mid a$$

Due to Haskell's referential transparency[1] it cannot distinguish between the two languages in figure 3.1. This may not seem like a problem as the

[1]Referential transparency means replacing names with their values does not alter the meaning of an expression.

two languages are identical. However, for Haskell to prove the A does not contain the empty word, we need to detect self-references, otherwise we will recurse indefinitely in an effort to prove A does not contain ε! We cannot currently verify whether two recursive languages are equal, so we will need to introduce a kind of tag for language nodes.

3.1.2 Tagging

We will introduce a tag for all possibly recursive language nodes, used to uniquely tag them.[2] For now, the library user will have to give a unique tag to each node manually. The reason for tagging every node instead of a subset has to do with an optimization called compaction detailled in section 3.3.3. However, the derivative creates new nodes that should get assigned unique tags too. Preferably, and to allow memoization of the derivative described in section 3.3.1, we prefer to not make the derivative computation stateful: we only want the derivative to depend on the letter being derived to and the language itself. We can accomplish this by allowing a tag to be extended by a marker for derivatives. We can then tag nodes created by the derivative based on the tag of the node being derived:

```
  -- A tag is a string with values tagged on.
data Tag t = Base String | Extend (TagValue t) (Tag t)
  deriving (Show, Eq, Ord)
  -- We need four different ways to extend a tag because the
  -- rule for concatenation creates four different recursive
  -- nodes which should all be tagged uniquely based on the
  -- tag of the node being derived.
data TagValue t = D1 t | D2 t | D3 t | D4 t
  deriving (Show, Eq, Ord)
data Language t = Empty
    | Epsilon
    | Terminal t
    | Nullable (Tag t) (Language w)
    | Union (Tag t) (Language w) (Language w)
    | Concatenation (Tag t) (Language w) (Language w)
derive :: (Eq t) ⇒ t → Language t → Language t
derive t Empty = Empty
derive t Epsilon = Empty
derive t (Terminal a) = if t ≡ a then Epsilon
  else Empty
derive t (Nullable _ a) = Empty
```

[2]Uniquely up to parser equality; if two parser nodes behave identical they are allowed to have the same tag.

```
derive t (Union name a b) = Union (Extend (D1 t) name)
  (derive t a) (derive t b)
derive t (Concatenation a b) = Union (Extend (D1 t) name)
  (Concatenation (Extend (D2 t) name)
    (derive t a)
    b)
  (Concatenation (Extend (D3 t) name)
    (Nullable (Extend (D4 t) name) a)
    (derive t b))
```

It is clear that every node in the derived tree will have a unique tag given that it is the case for the original tree. Suppose two nodes have the same tag; the tag of a node actually encodes uniquely how it was constructed, and therefore they must be equal.

3.1.3 Empty word in a language

We now want to determine the least fixed point of the nullability equations to determine whether a language contains the empty word, the last step required for recognition! To this end, we will use corollary 3 from appendix A; more concretely, we assume all nodes are not nullable, and recompute the nullability of all nodes until no changes occur. This process terminates because the number of nodes is finite, all functions are monotonic and the partially ordered set $\{\varnothing, \{\varepsilon\}\}$ is finite. We will use the following monad to keep track of the state during the recursive computation:

```
type Fix t = State (FixState t)
data FixState t = FixState
  {changed :: Bool
  , visited :: Set (Tag t)
  , values :: StrictMap.Map (Tag t) Bool
  }
```

Specifically, we store whether our current computation has caused any changes, we store a set of tags of nodes we have already visited to prevent infinite recursion, and we store the current estimates for the nullability function. We can then determine the nullability as follows:

```
parserNullable :: Language t → Bool
parserNullable = fixState ∘ work
  where
    work :: Language t → Fix t Bool
    work Empty          = return False
    work Epsilon        = return True
    work (Terminal _)   = return False
work (Union name a b) = do
  visited ← fixVisited name
  if visited then
    Maybe.fromMaybe False <$> fixValue name
  else do
    fixMarkVisited name
    current ← fixeValue name
    value ← (∨) <$> work a <*> work b
    when (current ≢ Just value)
      fixMarkChanged
    fixInsert name value
    return value
  -- Other clauses omitted.
```

The *work* function does most of the work. It is run repeatedly by the *fixState* function until no changes occur. The *fixState* function clears the changed and visited nodes each iteration. If the nullability is evaluated on an empty, epsilon or terminal language node, the *work* function returns immediately. If the node has already been visited, it returns the previously known result; otherwise, the function marks the node as visited, and calculates its own nullability. It subsequently marks a change if its nullability has changed, inserts the new value and returns.

The attentive reader may notice we do not precisely follow corollary 3 from appendix A, as each invocation of *work* may use newly calculated estimates instead of only using the old values. This still achieves the desired result, but more efficiently as we may require fewer iterations. Consider the following example:

$$\begin{aligned}\delta(L_1) &= \epsilon\\ \delta(L_2) &= \delta(L_1)\\ \delta(L_3) &= \delta(L_2)\end{aligned}$$

Formally, we want to compute the least fixed point of the following vector-valued function:

$$f\begin{pmatrix}L_1\\L_2\\L_3\end{pmatrix} = \begin{pmatrix}\{\varepsilon\}\\ \delta(L_1)\\ \delta(L_2)\end{pmatrix}$$

It is clear that:

$$f^3(\bot) = f^2(f(\bot)) = f^2\begin{pmatrix}\{\varepsilon\}\\ \varnothing\\ \varnothing\end{pmatrix} = f\left(f\begin{pmatrix}\{\varepsilon\}\\ \varnothing\\ \varnothing\end{pmatrix}\right) = f\begin{pmatrix}\{\varepsilon\}\\ \{\varepsilon\}\\ \varnothing\end{pmatrix} = \begin{pmatrix}\{\varepsilon\}\\ \{\varepsilon\}\\ \{\varepsilon\}\end{pmatrix}$$

and that it is a fixed point, so we would require three iterations when applying the corollary directly to determine the nullability of L_3. However, if run our code on L_3 it will immediately update the estimate of a single component using the results of the other components, resulting in only a single iteration! Formally, our implementation calculates the results of the individual components of f sequentially in a certain order. This still leads to the correct result because our estimates a satisfy $f^m(\bot) \leqslant a \leqslant f^n(\bot)$ for certain $m \leqslant n$, as all components of f are monotonic. Our estimate also increases each iteration as long as it has not yet reached a fixed point, since updating each component sequentially results in a value (not necessarily strictly) greater than applying f as a whole. Now we have all that is required to check whether a word is contained in a language:

```
contains :: Eq t ⇒ Language t → [t] → Bool
contains node []       = parserNullable node
contains node (x : xs) = contains (derive node x) xs
```

3.2 Parse forest

In this section we investigate how we can not only verify whether a word is contained in a language, but also construct a parse forest if it is. We will first introduce parser and parser combinators in nearly the same way as the authors of the original paper on parsing with derivatives [15] did.

3.2.1 Parser combinators

A partial parser takes input, and returns zero or more parse results, each of which consists of the value parsed and the remaining input. More formally, let T be the alphabet over which the parser is run, then we define the set of all parsers parsing words over T resulting in values of type A as

$$\mathbb{P}(T, A) = \{p : T^* \to \mathcal{P}(T^* \times A) \mid p(x) = (y, b) \Rightarrow \ y \text{ is a suffix of } x\}$$

For a parser p we write $\lfloor p \rfloor$ for the full parser based on p, it only returns parse results that consume the whole input: $\lfloor p \rfloor(x) = \{a \mid (\varepsilon, a) \in p(x)\}$. We will define a parser for each type of language node, and an additional reduce parser node used to construct more complicated parse trees. Parsers and languages are often denoted the same, but to which type of value is being referred to is always clear from context:

- The empty parser $\varnothing \in \mathbb{P}(T, A)$ is defined by $\varnothing(x) = \varnothing \in \mathcal{P}(T^* \times A)$, for any sets T and A.

- The epsilon parser consumes no input and returns fixed values, given as parameter, as parse results: $\varepsilon_R(x) = (x, R) = \{(x, r) \mid r \in R\}$, for any set T and $R \subseteq A$.
- The terminal parser parametrized by a given token either consumes the token and returns it as the parse result, or returns no results:

$$t(x) = \begin{cases} (w, t) \in \mathbb{P}(T, T) & \text{for } x = tw \\ \varnothing \in \mathbb{P}(T, T) & \text{otherwise} \end{cases}$$

  for any $t \in T$.
- For $p \in \mathbb{P}(T, A)$, the nullable parser $\delta(p) \in \mathbb{P}(T, A)$ equals the set of full parses for ε of p: $\delta(p)(x) = \{(x, a) \mid a \in \lfloor p \rfloor(\varepsilon)\}$.
- For $p \in \mathbb{P}(T, A)$ and map $f : A \to B$, we can run parser p and map the result using f, this results in a new parser reducing the old parser: $(p \hookrightarrow f)(x) = \{(y, f(a)) \mid (y, a) \in p(x)\}$.
- For $p_1, p_2 \in \mathbb{P}(T, A)$, we can take the union $p_1 \cup p_2 \in \mathbb{P}(T, A)$ of the parsers: $(p_1 \cup p_2)(x) = p_1(x) \cup p_2(x)$.
- For $p_1 \in \mathbb{P}(T, A_1)$ and $p_2 \in \mathbb{P}(T, A_2)$ we can concatenate the parsers into a parser $p_1 \circ p_2$:

$$(p_1 \circ p_2)(x) = \{(x'', (a, b)) \mid (x', a) = p(x) \text{ and } (x'', b) = p(x')\} \\ \in \mathbb{P}(T, A_1 \times A_2)$$

3.2.2 Derivatives of parsers

We will now need to define the derivative of parsers, and we will do so as done by the authors of the original paper [15]. The derivative of a parser p with respect to a letter t should behave as though it has already parsed t:

$$\partial_t\, p(x) = p(tx) \setminus (\lfloor p \rfloor(\varepsilon) \times \{tx\})$$

Note that we need to remove the null parses (parses that did not consume input), as we require the parser to have parsed t already. This definition is similar to the definition of the derivative for languages. In particular, we can now derive to all letters of the input, and apply it as full parser to ε to get the result of applying the original parser to the input. We have the following rules for the derivative of parsers:

- $\partial_t\, \varnothing = \varnothing$
- $\partial_t\, \varepsilon = \varnothing$

- $\partial_t\ t' = \begin{cases} \varepsilon_{\{t\}} & \text{if } t = t' \\ \varnothing & \text{otherwise} \end{cases}$
- $\partial_t\ \delta(p) = \varnothing$
- $\partial_t(p \hookrightarrow f)(x) = \partial_t\ p \hookrightarrow f$
- $\partial_t(p_1 \cup p_2) = \partial_t\ p_1 \cup \partial_t\ p_2$
- $\partial_t(p_1 \circ p_2) = (\partial_t\ p_1 \circ p_2) \cup (\delta(p_1) \circ \partial_t\ p_2)$

Again, the implementation tags all newly created parsers with names derived from the tag of the parser on the left-hand side. Implementation-wise, barely anything has changed except for the fact that parser nodes are parametrized over the value they parse, and the rule for deriving a terminal has changed a bit:

```
  -- The epsilon node returns the terminal's value.
derive c (Terminal a) = if a ≡ c then ε c else ∅
```

3.2.3 Parsing the empty word

We have now reduced the problem of parsing to parsing the empty word (null parsing). Analogous to checking nullability by calculating a least fixed point, we will calculate the null parses using a least fixed point. Again, we start with assuming each parser produces no parse results on the empty word, and then iteratively calculate the parse results for the empty word of all nodes. Note that this only works for finite parse forests![3]

As for the implementation, it is exactly analogous to the implementation for the least fixed point calculation for the nullability except for the type of values. The cached result for a parser *Parser t a* is of type *Set a*. In fact, we parametrize the monad given earlier over the type of value to cache. Unfortunately, the types of parse forests generated by each node may differ, so we have to use a *Dynamic* value to store them in a dictionary. The fixed point monad looks as follows:

```
data FixState t v = FixState
  { changed :: Bool
  , visited :: Set (Tag t)
  , values :: StrictMap.Map (Tag t) v
  }
```

[3]The existence of the least fixed point for finite parse forests follows from corollary 3 in appendix A only if the iteration converges in a finite number of steps; we cannot calculate the suprema over an infinite set using basic iteration.

```
type $Fix\ t\ v = State\ (FixState\ t\ v)$
```

And then we can calculate the null parses:

```
$parseNull :: (Token\ t, ParseTree\ a) \Rightarrow Parser\ t\ a \rightarrow Set\ a$
$parseNull = fixState \circ work$
  where
    $work\ Empty = return\ Set.\varnothing$
    $work\ (Reduce\ name\ f\ a) =$ do
      $visited \leftarrow fixStateVisited\ name$
      if $visited$ then
        $flip\ fromDyn\ \bot$
            $<\$>\ Maybe.fromMaybe\ (toDyn\ (Set.\varnothing :: Set\ a))$
            $<\$>\ fixStateValue\ name$
      else do
        $fixStateMarkVisited\ name$
        $current \leftarrow fmap\ (flip\ fromDyn\ \bot) <\$> fixStateValue\ name$
        $other \leftarrow work\ a$
        $value \leftarrow Set.map\ f <\$> work\ a$
        $when\ (current \not\equiv Just\ value)\ fixStateMarkChanged$
        $fixStateInsert\ name\ (toDyn\ value)$
        $return\ value$
      -- Other clauses omitted.
```

Finally, we can parse:

```
$parse :: (Token\ t, ParseTree\ a) \Rightarrow Parser\ t\ a \rightarrow [t] \rightarrow Set\ a$
$parse\ parser\ [\,]$         $= parseNull\ parser$
      $parser\ (x : xs) = parse\ (derive\ parser\ x)\ xs$
```

The *Token* and *ParseTree* classes simply shorten the constraints on the token and parse tree types respectively, for the derivative and null parsing:

```
  -- Different from ‘ParseTree‘ as another requirement will be
  -- added needed for memoization of the derivative in the next
  -- section.
class $(Ord\ a, Typeable\ a) \Rightarrow Token\ a$
class $(Ord\ a, Typeable\ a) \Rightarrow ParseTree\ a$
```

3.3 Various improvements

3.3.1 Memoization of the derivative

The original paper about parsing with derivatives requires memoization of the derivative for termination. In essence, it requires a function:

```
  -- Returns a memoized version of the argument.
memoize :: (a → b) → (a → b)
```

However, from the perspective of Haskell, the input and output are identical; as the result of a function depends only on its input, we cannot detect any difference between the results of multiple evaluations of the function with the same input, memoized or not. Therefore, memoization in our Haskell implementation is only an optimization that may reduce the number of nodes that need to be actually constructed in memory, but is not required in any way.

Currently the signature of the derivative is similar to:

```
derive :: (Token t, ParseTree a) ⇒ t → Parser t a → Parser t a
```

Therefore we need to memoize on both the tokens and the parser. We can achieve memoization on the parser by storing the derivative as field of the parser data type, which will only calculated when it is required due to Haskell's laziness. Memoization on the token is less straight-forward; the Haskell implementation by the authors uses unsafe IO for this purpose, but we demand a safer method.

We implement memoization using generalized tries as described in [9] and base our implementation partially[4] on [7]. The fundamental idea is that given a map of type $a \to b$, we create a data structure that maps each value of type a to the value of type b given by the map. We can then, instead of evaluating the map, retrieve the value from the data structure so that it will only be calculated the first time we access it. We can construct such a data structure for algebraic data types by using tries. Tries exploit the structure of a to implement the data structure, for example:

```
data BoolTrie b = BoolTrie b b
trie :: (Bool → b) → BoolTrie b
trie f = BoolTrie (f False) (f True)
untrie :: BoolTrie b → (Bool → b)
untrie (BoolTrie false _    ) False = false
untrie (BoolTrie _     true) True  = true
```

[4]In particular, we use their type class and a very similar implementation for bounded enumeration types.

```
memoize :: (Bool → b) → (Bool → b)
memoize = untrie ∘ trie
```

As seen above, memoization is simply converting to a trie and reading from it! We use the following type class for any value that can be converted to and from a trie:

```
class HasTrie a where
  data Trie a :: * → *
  trie         :: (a → b) → Trie a b
  untrie       :: Trie a b → a → b
```

The tries contain values for all possible values of the domain. To implement tries for any data structure, we can use the following isomorphisms:

$$\begin{aligned} 1 \to v &\cong v \\ a + b \to v &\cong (a \to v) \times (b \to v) \\ a \times b \to v &\cong a \to (b \to v) \end{aligned}$$

The isomorphism maps themselves are obvious. We can now implement instances of *HasTrie* for the basic data types using the above isomorphisms:

```
instance HasTrie () where
  newtype Trie () a = UnitTrie a
  trie f = UnitTrie (f ())
  untrie (UnitTrie value) () = value
instance (HasTrie a, HasTrie b) ⇒ HasTrie (Either a b) where
  data Trie (Either a b) c = EitherTrie (Trie a c) (Trie b c)
  trie f = EitherTrie (trie $ f ∘ Left) (trie $ f ∘ Right)
  untrie (EitherTrie a _) (Left value  ) = untrie a value
  untrie (EitherTrie _ b) (Right value) = untrie b value
instance (HasTrie a, HasTrie b) ⇒ HasTrie (a, b) where
  newtype Trie (a, b) c = TupleTrie (Trie a (Trie b c))
  trie f = TupleTrie $ trie $ λx → trie (f ∘ (, ) x)
  untrie (TupleTrie trie) (a, b) = untrie (untrie trie a) b
```

We can now use these basic instances to implement instances for any algebraic data type using basic type isomorphisms; consider, for example, the following type:

```
data Token = Empty | A Token | B Token Token
  deriving Show
```

It can easily be seen that:

$$\text{Token} \cong \text{Empty} + (\text{Token} + \text{Token} \times \text{Token})$$

And therefore we can effortlessly use our above instances to define *HasTrie* for *Token*, by simply converting a value to and from its elementary form.

```
instance HasTrie Token where
  newtype Trie Token b = TokenTrie (
    Trie (Either () (
          Either Token
                 (Token, Token))) b)
  trie f = TokenTrie $ trie $ f ∘ from
    where
      from ((Left _))                   = Empty
      from ((Right (Left t)))           = A t
      from ((Right (Right (t, t′))))    = B t t′
  untrie (TokenTrie trie) = untrie trie ∘ to
    where
      to Empty    = Left ()
      to (A t)    = Right $ Left t
      to (B t t′) = Right $ Right (t, t′)
```

Similarly, we can define instances for primitives like lists, as a list can be seen as a normal algebraic type with constructors [ ] and (:). The *HasTrie* implementation for *Char* in our implementation is not a typical trie; it exploits the properties of the key's *Bounded* and *Enum* instances, but is implemented using a binary search tree for reasons of efficiency. In fact, we can automatically generate such a binary search tree for any object that is an instance of *Bounded* and *Enum*.

As an additional note, testing that memoization does in fact work requires unsafe Haskell code. We can, for example, do the following:

```
f = memoize (λx → trace "function evaluated" $ show x)
```

3.3.2 Tagging nodes automatically

Tagging all nodes is obviously not very ergonomic. We implement a *tagAll* function that takes a parser where only a subset of nodes is tagged (but enough to prevent cycles of untagged nodes), and tags all remaining nodes. To allow construction of parsers without tag, we add a *None* variant. The implementation of the tagging function uses the state monad to save the next tag to assign and a map from tags to parsers so that we can "tie the knot". The type of tags assigned by *tagAll* is a new variant of *Tag*: *BaseAuto Int*. With this addition we can write, for example, a parser for zero or more `x`'s as follows:

```
  -- Right recursive
xsR :: Parser Char String
```

$$\begin{aligned}
&\mathit{xsR} = \texttt{"xsR"} <\#> \varepsilon\,[\,]\\
&\qquad\qquad <|> \mathit{terminal}\ \texttt{'x'} \circ \mathit{xsR} \hookrightarrow \mathit{uncurry}\ (:)
\end{aligned}$$

The *parse* function will then invoke *tagAll* before parsing. We have used the following operators for briefness:

- `<#>` for tagging a taggable node with a string.
- `<|>` for the union of two parsers.
- $\circ$ for the concatenation of two parsers.
- $\hookrightarrow$ for reducing a parser with a function.

3.3.3 Compaction

Although parsing already works, it is extremely slow. Consider the following grammar:

$$X \to xX \mid \varepsilon$$

We can implement a parser in our library as in the previous section:

$$\begin{aligned}
&\mathit{xsR} :: \mathit{Parser\ Char\ String}\\
&\mathit{xsR} = \texttt{"xsR"} <\#> \varepsilon\,[\,]\\
&\qquad\qquad <|> \mathit{terminal}\ \texttt{'x'} \circ \mathit{xsR} \hookrightarrow \mathit{uncurry}\ (:)
\end{aligned}$$

However, parsing a sequence of fifteen `x`'s takes little over a minute! This is caused by generating a lot of useless parser nodes. We fix this with a process called compaction [15], which essentially simplifies parsers via a number of identities. We have implemented the following simplifications:

$$\begin{aligned}
\delta(\varnothing) &\Rightarrow \varnothing\\
\delta(\varepsilon_R) &\Rightarrow \varepsilon_R\\
\delta(t) &\Rightarrow \varnothing\\
\varnothing \hookrightarrow f &\Rightarrow \varnothing\\
\varepsilon_R \hookrightarrow f &\Rightarrow \varepsilon_{R'} \text{ where } R' = \{f(r) \mid r \in R\}\\
(\varepsilon_{\{a\}} \circ p) \hookrightarrow f &\Rightarrow p \hookrightarrow g \text{ where } g(x) = f(a, x)\\
(p \circ \varepsilon_{\{a\}}) \hookrightarrow f &\Rightarrow p \hookrightarrow g \text{ where } g(x) = f(x, a)\\
(p \hookrightarrow f) \hookrightarrow g &\Rightarrow p \hookrightarrow (g \circ f)\\
p \circ \varepsilon_{\{a\}} &\Rightarrow p \hookrightarrow f \text{ where } f(x) = (x, a)\\
\varepsilon_{\{a\}} \circ p &\Rightarrow p \hookrightarrow f \text{ where } f(x) = (a, x)\\
p \circ \varnothing &\Rightarrow \varnothing\\
\varnothing \circ p &\Rightarrow \varnothing\\
p \cup \varnothing &\Rightarrow p\\
\varnothing \cup p &\Rightarrow p
\end{aligned}$$

We want to apply these simplifications recursively without infinite recursion or repeatedly optimizing the same parser. We achieve this by storing the compacted version of the parser alongside the parser itself, so that we can simply access the lazily-computed compacted version of a parser for its compacted version. This prevents infinite recursion by effectively memoizing the compaction function. Haskell's pattern matching allows a succinct implementation, see for example the first rule:

```
  -- The additional parser fields are its derivative and
  -- compacted version.
runCompact (Parser (Nullable _ (Parser Empty _ _)) _ _) = ∅
```

Implementing compaction results in parsing the sequence of fifteen `x`'s instantaneously, as should be the case. Notably, the simplifications for the δ nodes not described in [15] are required: without them the δ nodes are opaque to the compaction function and parsing stays extremely slow. Depending on the grammar, it might be preferential to repeatedly apply compaction; our library allows the user to declare the number of times compaction should be iterated, but by default does ten iterations like the original Haskell implementation.

As the parser on the right-hand side behaves the same as the corresponding parser on the left-hand side, we can reuse the left-hand parser's tag.

Tagging nodes in the compaction function is the reason why we need to tag *all* nodes. In principle, we could also only tag enough nodes to make sure every cycle in the grammar has at least one tagged node... However, consider the following instances of the above simplifications:

$$(p \hookrightarrow f) \hookrightarrow g \Rightarrow p \hookrightarrow g \circ f$$
$$(p \hookrightarrow f) \hookrightarrow h \Rightarrow p \hookrightarrow h \circ f$$

where the only tagged node is the node that reduces by f. In this case, we cannot distinguish the expressions on the left side, so giving unique tags to the right-hand side based on the left-hand side is impossible.

3.4 Performance

3.4.1 Complexity

The laziness of Haskell makes time complexity harder to analyze, but we will summarize the analysis done in a follow-up paper [1] of the original paper on parsing with derivatives. We also assume lookup and insertion into dictionaries to be constant time. We do not consider compaction as it is only an optimization, normally improves performance, and can easily be disabled if that is not the case.

Let g denote the number of total parser nodes after taking all derivatives without compaction, then the sum of the running time of all invocations of *derive* is $\mathcal{O}(g)$: we only have to take into account calls that are not yet memoized, and they create at least one new node per call.

The authors subsequently show that the total number of nodes g is bounded by $\mathcal{O}(Gn^3)$ where G is the size of the initial parser in number of parser nodes. They achieve this by assigning a unique symbol to each node and showing there are at most $\mathcal{O}(Gn^3)$ such symbols after deriving.

Now the complexity $\mathcal{O}(f(g))$ of null parsing in g remains to be calculated, as then the total complexity is $\mathcal{O}(f(Gn^3))$. The authors of the follow-up paper improve upon iteration using an improved algorithm which uses mutable state to achieve a linear complexity in the number of nodes. It is not clear how such an algorithm could be implemented efficiently in Haskell, so we have used straight-forward iteration. We suspect only a single iteration to be required for unambiguous grammars under basic conditions, resulting in cubic complexity, but we can currently only prove this for regular languages.

Let us consider a variant of parsing with derivatives where we do not allow recursive parsers and add a Kleene star parser p^*, so that we can still parse any regular language. It is clear that only a single iteration is required to find the fixed point of the nullability equation in this case, as the null parsing function becomes a simple memoized function on the parser nodes. The final parse result is a set with at most a single element, therefore only a single element needs to be evaluated per node resulting in linear complexity in the number of nodes.

3.4.2 Benchmark

We benchmarked our implementation against Parsec [12], a popular Haskell parser combinator library, using the Criterion [16] benchmarking library. We only parsed unambiguous grammars without left-recursive productions as Parsec cannot handle any other grammars. It is important to note that constructing the parsers using parsing with derivatives was more straightforward than using Parsec: it is not required to keep in mind e.g. the evaluation order of `<|>` and other operators. We parsed the following expressions:

Name	Expression
xs/3	*replicate* 3 `'x'`
xs/100	*replicate* 100 `'x'`
xs/1000	*replicate* 1000 `'x'`
parenthesis/small	`"((()))"`
parenthesis/large	(*replicate* 100 `'('`) ++ (*replicate* 100 `')'`)
expression/basic	`"1010*10101+101+(101+1)+0*1+1*(1+1)"`

Criterion calculated the following estimates using OLS regression for the running times:

Name	Lower bound	Estimate	Upper bound
xs/3/parser	3328 ns	3397 ns	3486 ns
xs/3/parsec	453,0 ns	474,2 ns	503,0 ns
xs/100/parser	55,96 μs	57,20 μs	58,84 μs
xs/100/parsec	7,701 μs	7,839 μs	8,072 μs
xs/1000/parser	1182 μs	1192 μs	1204 μs
xs/1000/parsec	83,12 μs	83,71 μs	84,58 μs
parenthesis/small/parser	10,51 μs	10,66 μs	10,83 μs
parenthesis/small/parsec	1,219 μs	1,236 μs	1,257 μs
parenthesis/large/parser	6202 μs	7844 μs	8920 μs
parenthesis/large/parsec	91,42 μs	91,76 μs	92,16 μs
expression/basic/parser	23350 μs	23840 μs	24550 μs
expression/basic/parsec	30,37 μs	30,53 μs	30,71 μs

Without additional diagrams, it is clear that our implementation performs very slowly compared to Parsec; around a factor of 10 to 1000 times slower for our test cases. In the next section we analyze the results. The raw benchmark and profiling results can be found in appendix B.

3.4.3 Analysis

Our implementation has a lot of overhead compared to the much more straight-forward recursive descent parsing done by Parsec. Unfortunately, profiling shows that parsing the parenthesis was not slowed down significantly by unexpected cost centres: more than 80% of time was spent in the *runCompact* and *parserFromNode* functions, responsible for compaction and creating new nodes respectively. We do not see how we can significantly improve this; the optimized Racket implementation by the original authors applies compaction once, locally, using smart constructors. However, this is impossible in Haskell without giving the constructors additional arguments as the tree of parsers would be evaluated completely to determine the shape of the root, recursing infinitely.

However, profiling while parsing the very basic expression shows a worrisome result: around 25% of the running time was spent comparing *Tag* values. The size of tags will only increase as the number of derivatives and compactions done increases. A more complex implementation might only use e.g. integers for tagging, and keep track of which integer to use next throughout derivation. However, this would be more error-prone and memoizing the derivative could not be done so easily. Derivation would be stateful, and memoization would require manually mapping nodes to their derivatives.

Another approach that would be much more efficient, *and* increase the ergonomics of the library substantially, involves generating unique tags for nodes using unsafe IO. Then we can use, for example, a global counter to keep track of the next unused tag. We could then tag parsers automatically when constructing them, so it would lift the requirement of tagging *any* node.

The excessive number of allocations done by the compaction algorithm could be greatly reduced using mutable state, and therefore improve performance. This is presumably the reason why the Racket implementation performs much better. [1]

Chapter 4

Related Work

In 2010, Might et al. published online the first version of their paper about parsing with derivatives [14]. After feedback and refinements they published their paper accompanied by a Racket, Haskell and Scala implementation. The work applies Brzozowski's work on derivatives for regular languages [4] to context-free languages as a new method of parsing. Parsing with derivatives can be used to construct parser combinator libraries that can parse arbitrary context-free grammars, which is usually complicated (see for example [8]).

Parsing arbitrary context-free grammars with reasonable efficiency has long been possible; the CYK [10], GLR [11] and Earley parsing [6] are the well-known algorithms for parsing arbitrary context-free grammars. All three have cubic worst-case time complexity, but there are significant differences. CYK operates only on grammars given in Chomsky normal form, and is usually slower than an Earley parser, but is reasonably simple [13]. An Earley parser, though more efficient, is more complicated and its correctness does not follow directly from a mathematical theory. Parsing with derivatives is an additional parsing methods of which the correctness is easier to prove, but its performance in practice may be worse.

The Haskell implementation by the authors does not work on all grammars and uses unsafe memoization. The implementation terminates on most common grammars by first deriving as usual, and then repeatedly compacting; the compaction function is modified to propagate δ nodes downward in hopes of reducing the parser to an empty or epsilon parser. Memoization is done by unsafely creating a mutable data structure for storing results.

In a follow-up on parsing with derivatives, they proof a modified version of the algorithm can achieve cubic complexity under basic assumptions. [1] Our implementation is slower due to the data structures used and the requirement of tagging, as well as a slower fixed point algorithm. The follow-up paper uses a more advanced fixed point algorithm, but it is unclear how to efficiently implement it without mutable state.

Chapter 5

Conclusions

We have shown how the equations for the derivative of languages follow succinctly from the definition of a right factor of a language. Next we have shown how to implement parsing by tagging nodes and straight-forward iteration to calculate fixed points. We implemented memoization based on generalized tries; [9] an elegant method that allows any user of the library to quickly allow safe memoization for any token type. Tagging nodes to prevent recursion is neither very ergonomic nor elegant, but the usability is satisfactory for most cases. If acceptable, tagging can be implemented unsafely for increased performance and usability.

The result is a working parser combinator library in Haskell of which the correctness is clearer than many other algorithms for parsing arbitrary context-free grammars. We show the time complexity is cubic for parsing regular expressions under basic assumptions, and cubic for any context-free language if the fixed point computation takes linear time; profiling showed null parsing taking up only a fraction of the running time, so we suspect the complexity to be cubic for all grammars.

The performance in practice is bad compared to Parsec. Many of the performance issues originate from Haskell's lack of mutable state to implement certain optimizations to reduce the number of allocations. Usage of unsafe and imperative style Haskell could improve performance, but this would increase the complexity of the implementation and make its correctness harder to verify.

We would like to conclude by remarking that even though parsing with derivatives resulted in a much slower implementation than Parsec, constructing parsers feels more natural since their structure matches the grammar. The algorithm is also online: it can start parsing with incomplete input.

Bibliography

[1] Michael D. Adams, Celeste Hollenbeck, and Matthew Might. On the complexity and performance of parsing with derivatives. *CoRR*, abs/1604.04695, 2016.

[2] Roland Backhouse. Galois connections and fixed point calculus. In *Algebraic and Coalgebraic Methods in the Mathematics of Program Construction*, pages 89–150. Springer, 2002.

[3] Roland Carl Backhouse. Fusion on languages. In *Proceedings of the 10th European Symposium on Programming Languages and Systems*, ESOP '01, pages 107–121, London, UK, UK, 2001. Springer-Verlag.

[4] Janusz A. Brzozowski. Derivatives of regular expressions. *J. ACM*, 11(4):481–494, October 1964.

[5] David Darais. The derp package. `https://hackage.haskell.org/package/derp-0.1.6`, 2012.

[6] Jay Earley. An efficient context-free parsing algorithm. *Communications of the ACM*, 13(2):94–102, 1970.

[7] Conal Elliott. The memotrie package. `https://hackage.haskell.org/package/MemoTrie`, 2008.

[8] Richard A. Frost and Rahmatullah Hafiz. A new top-down parsing algorithm to accommodate ambiguity and left recursion in polynomial time. *SIGPLAN Not.*, 41(5):46–54, May 2006.

[9] Ralf Hinze. Generalizing generalized tries. *J. Funct. Program.*, 10(4):327–351, July 2000.

[10] Tadao Kasami. An efficient recognition and syntaxanalysis algorithm for context-free languages. page 40, 07 1965.

[11] Bernard Lang. Deterministic techniques for efficient non-deterministic parsers, 07 1974.

[12] Daan Leijen and Paolo Martini. The parsec package. `https://hackage.haskell.org/package/parsec`, 2008.

[13] Te Li and Devi Alagappan. A comparison of cyk and earley parsing algorithms.

[14] Matthew Might and David Darais. Yacc is dead. *CoRR*, abs/1010.5023, 2010.

[15] Matthew Might, David Darais, and Daniel Spiewak. Parsing with derivatives: A functional pearl. *SIGPLAN Not.*, 46(9):189–195, September 2011.

[16] Bryan O'Sullivan. The criterion package. `https://hackage.haskell.org/package/criterion`, 2009.

Appendix A

Preliminaries

This appendix details a few required and well-known results from mathematics.

Theorem 2. Let L be a complete lattice, and let $f : L \to L$ be a map preserving directed suprema. Then f has a least fixed point given by $\bigvee_{n\geqslant 0} f^n(\bot)$.

Proof. We first show that $\alpha = \bigvee_{n\geqslant 0} f^n(\bot)$ is a fixed point. Since f preserves directed suprema we have:

$$\begin{aligned} f(\alpha) &= f\left(\bigvee_{n\geqslant 0} f^n(\bot)\right) \\ &= \bigvee_{n\geqslant 0} f(f^n(\bot)) \\ &= \bigvee_{n\geqslant 0} f^{n+1}(\bot) \\ &= \alpha \end{aligned}$$

Next we show that α is the least fixed point. Suppose $\beta \in L$ is any other fixed point. We have $\bot \leqslant \beta$, so by induction and monotonicity of f it follows that $f^n(\bot) \leqslant \beta$ for any $n \geqslant 0$. Then it follows from the definition of the supremum that $\alpha \leqslant \beta$. □

Corollary 3. Let $A_1, \ldots, A_n$ be arbitrary sets, such that $L = \mathcal{P}(A_1) \times \cdots \times \mathcal{P}(A_n)$, and let $f : L \to L$ be a monotonic function that preserves directed suprema. Then f has a fixed point given by $\bigvee_{n\geqslant 0} f^n(\varnothing, \ldots, \varnothing)$ over the product lattice L.

Proof. Clearly L forms a complete product lattice with bottom element $\varnothing \times \cdots \times \varnothing$ and top element $A_1 \times \cdots \times A_n$, as $\mathcal{P}(A_i)$ is a complete lattice for $1 \leqslant i \leqslant n$. Therefore the result follows from the previous theorem. □

Appendix B

Code

Our Haskell implementation, including examples, benchmark, benchmark results and profiling results, can be found at:

`https://gitlab.science.ru.nl/tmaarse/parsing-with-derivatives`