Parsing with Derivatives

Cole Margerum and Isaac Zinman

Overview

- Paper by Matthew Might, David Darais, Daniel Spiewak
- Presented at ICFP, 2011

Background: Definition of Formal Languages

- Atomic languages:
  - $\varnothing = \{\}$
  - $\varepsilon = \{\varepsilon\}$
  - $c \in A = \{c\}$, over some alphabet A
- Regular languages: Atomic languages combined with union, concatenation, Kleene star
- Context-free languages: Regular with mutual recursion

Brzozowski Derivative - Regular Expressions

Definition:

- $D_c(L) = \{w : cw \in L\}$

Examples:

- $D_b\{foo, bar, baz\} = \{ar, az\}$
- $D_f\{foo, bar, baz\} = \{oo\}$
- $D_a\{foo, bar, baz\} = \emptyset$

Membership with the Derivative

- $cw \in L$ iff $w \in D_c(L)$
- Repeat with every character using the previous derivative
- Check if resultant language contains the empty string: if so, the original string is part of L

Derivatives on the Atomic Languages

- $D_c(\emptyset) = \emptyset$
- $D_c(\epsilon) = \emptyset$
- $D_c(c) = \epsilon$
- $D_c(c') = \emptyset$ if $c \neq c'$

Closures

$$D_{\mathtt{f}}\,\{\mathtt{foo},\mathtt{bar}\}^\star = \{\mathtt{oo}\} \circ \{\mathtt{foo},\mathtt{bar}\}^\star$$
$$D_{\mathtt{f}}\,\{\mathtt{foo},\mathtt{bar}\}^\star \circ \{\mathtt{frak}\} = \{\mathtt{oo}\} \circ \{\mathtt{foo},\mathtt{bar}\}^\star \circ \{\mathtt{frak}\} \cup \{\mathtt{rak}\}$$

Union

- $D_c(L_1 \cup L_2) = D_c(L_1) \cup D_c(L_2)$

Kleene Star

- $D_c(L^*) = D_c(L) \circ L^*$

Concatenation

- $D_c(L_1 \circ L_2) = D_c(L_1) \circ L_2$, if $\epsilon \notin L_1$
- $D_c(L_1 \circ L_2) = (D_c(L_1) \circ L_2) \cup D_c(L_2)$, if $\epsilon \in L_1$

Simplification of Concatenation

Nullability Function

- $\delta(L) = \emptyset$, if $\epsilon \notin L$
- $\delta(L) = \epsilon$, if $\epsilon \in L$

Revised Concatenation:

- $D_c(L_1 \circ L_2) = (D_c(L_1) \circ L_2) \cup (\delta(L_1) \circ D_c(L_2))$

Nullability

- $\delta(\emptyset) = \emptyset$
- $\delta(\epsilon) = \epsilon$
- $\delta(c) = \emptyset$
- $\delta(L_1 \cup L_2) = \delta(L_1) \cup \delta(L_2)$
- $\delta(L_1 \circ L_2) = \delta(L_1) \circ \delta(L_2)$
- $\delta(L^*) = \epsilon$

Derivatives of Context-Free Languages

- Derivative code for RL doesn't work with CFG's
- Recursive implementation of the derivative and the recursive nature of CFG's leads to non-termination

Example:

$L = L \circ \{x\} \cup \epsilon$

$D_x L = D_x L \circ \{x\} \cup \epsilon$

Solutions to Non-Termination

- Laziness
  - Concatenation, Union, and Repetition done by need-only
- Memoization
  - Use derivatives of languages already seen
- Least Fixed Points
  - Expand only as much as necessary…?

Least Fixed Points

If we allow mutually recursive definitions, then the set of describable languages is exactly the set of context-free languages. (Even without Kleene star, the resulting set of languages is context-free.) We assume, of course, a least-fixed-point interpretation of such recursive structure. For instance, given the language L:

$$L = (\{\mathtt{x}\} \circ L) \cup \epsilon.$$

The least-fixed-point interpretation of L is a set containing a finite string of every length (plus the null string). Every string contains only the character $\mathtt{x}$. [The greatest-fixed-point interpretation of L adds an infinite string of $\mathtt{x}$'s.]

From Recognition to Parsing

- Partial parser:
  - $\mathbb{P}(A, T) \subseteq A^* \rightarrow \mathscr{P}(T \times A^*)$ for alphabet A, parse tree T
- Full parser:
  - $\lfloor \mathbb{P} \rfloor(A, T) \subseteq A^* \rightarrow \mathscr{P}(T)$
- Atomic languages easily translate to parsers
  - Single character -> partial parser for exactly itself
  - Empty set -> reject-everything
  - Empty string -> consume-nothing, accept-everything

Parser Combinators

Union: The union of two parsers, $p, q \in \mathbb{P}(A, X)$, combines all parse trees together, so that $p \cup q \in \mathbb{P}(A, X)$:

$$p \cup q = \lambda w.p(w) \cup q(w).$$

Concatenation: The concatenation of two parsers, $p \in \mathbb{P}(A, X)$ and $q \in \mathbb{Q}(A, Y)$, produces a parser that pairs the parse trees of the individual parsers together, so that $p \circ q \in \mathbb{P}(A, X \times Y)$:

$$p \circ q = \lambda w.\{((x, y), w'') : (x, w') \in p(w), (y, w'') \in q(w')\}$$

Function reduction: A reduction by function $f : X \to Y$ over a parser $p \in \mathbb{P}(A, X)$ creates a new partial parser, $p \to f \in \mathbb{P}(A, Y)$:

$$p \to f = \lambda w.\{((f(x), w') : (x, w') \in p(w)\}$$

Parser Combinators

Nullability:

A special nullability combinator, δ, simplifies the definition of the derivative over parsers. It becomes a reject-everything parser if the language cannot parse empty, and the null parser if it can:

$$\delta(p) = \lambda w. \{(t, w) : t \in \lfloor p \rfloor(\epsilon)\}.$$

Null reduction:

To implement the derivative of parsers for single characters: the null reduction partial parser, $\epsilon \downarrow S$, is handy. This parser can only parse the null string; it returns a set of parse trees stored within:

$$\epsilon \downarrow S \equiv \lambda w. \{(t, w) : t \in S\}.$$

Kleene star:

It is easiest to define the Kleene star of a partial parser $p \in \mathbb{P}(A, T)$ in terms of concatenation, union and reduction, so that $p^\star \in \mathbb{P}(A, T^*)$:

$$\begin{aligned} p^\star = & (p \circ p^\star) \to \lambda(head, tail).head : tail \\ & \cup \epsilon \downarrow \{\langle\rangle\}. \end{aligned}$$

The colon operator (:) is the sequence constructor, and $\langle\rangle$ is the empty sequence.

Derivatives of Parser Combinators

- Intuitive definition: $D_c(P) = P'(A, T)$ if $P(A, T)$; that is, alphabet and parse tree types are the same
- Derivative strips the character and eliminates null parses (doesn’t make sense to expand input)
- Formally: $D_c(p) = \lambda w.p(cw) - (\lfloor p \rfloor(\epsilon) \times \{cw\}).$

Derivatives of Parser Combinators

Derivatives of atomic parsers:

The derivative of the empty parser is empty:

$$D_c(\emptyset) = \emptyset.$$

The derivative of the null parser is also empty:

$$D_c(\epsilon) = \emptyset.$$

The derivative of the nullability combinator must be empty, since it at most parses the empty string:

$$D_c(\delta(L)) = \emptyset.$$

The derivative of a single-character parser is either the null reduction parser or the empty parser:

$$D_c(c') = \begin{cases} \epsilon \downarrow \{c\} & c = c' \\ \emptyset & \text{otherwise.} \end{cases}$$

Derivatives of combined parsers:

The derivative of the union is the union of the derivative:

$$D_c(p \cup q) = D_c(p) \cup D_c(q).$$

The derivative of a reduction is the reduction of the derivative:

$$D_c(p \to f) = D_c(p) \to f.$$

The derivative of concatenation requires nullability, in case the first parser doesn't consume any input:

$$D_c(p \circ q) = (D_c(p) \circ q) \cup (\delta(p) \circ D_c(q)).$$

The derivative of Kleene star peels off a copy of the parser:

$$D_c(p^\star) = (D_c(p) \circ p^\star) \to \lambda(h,t).h : t$$

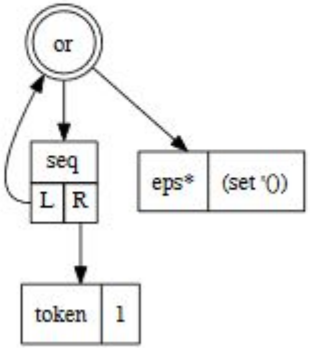
Parsing with Derivatives of Parser Combinators

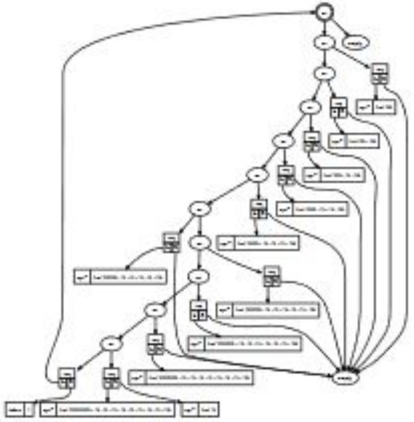
- Compute successive derivatives of the top-level parser with respect to each character in a string
- Supply null character to resultant parser and see if it matches
  - How to parse null?

Performance Analysis

- Due to null expansion of concatenation, derivatives grow exponentially, leading to worst-case $O(n^{2n}G^2)$ where n is input tokens and G is the size of the grammar

- Left: original grammar
- Right: after 10 derivatives





Solution: Compaction

$$\emptyset \circ p = p \circ \emptyset \Rightarrow \emptyset$$
$$\emptyset \cup p = p \cup \emptyset \Rightarrow p$$
$$(\epsilon \downarrow \{t_1\}) \circ p \Rightarrow p \to \lambda t_2.(t_1, t_2)$$
$$p \circ (\epsilon \downarrow \{t_2\}) \Rightarrow p \to \lambda t_1.(t_1, t_2)$$
$$(\epsilon \downarrow \{t_1, \ldots, t_n\}) \to f \Rightarrow \epsilon \downarrow \{f(t_1), \ldots, f(t_n)\}$$
$$((\epsilon \downarrow \{t_1\}) \circ p) \to f \Rightarrow p \to \lambda t_2.f(t_1, t_2)$$
$$(p \to f) \to g \Rightarrow p \to (g \circ f)$$
$$\emptyset^{\star} \Rightarrow \epsilon \downarrow \{\langle\rangle\}\,.$$

“We can implement these simplification rules in a memoized, recursive simplification function. When simplification is deeply recursive and memoized, we term it *compaction.*” [Note: must use recursive, not just top-level reduction, or it still expands exponentially]

Parsing with Compaction: Analysis

- Keeps approximately constant-size grammar while taking successive derivatives until last derivative collapses to parse forest
- Still worst-case exponential, but (conjectured) average-case O(nG) for parsing and recognition of unambiguous grammars - parsing for ambiguous grammars means returning parse forest, which is necessarily exponential, but *recognition* of ambiguous grammars also believed to be O(nG)