Derivatives of Regular Expressions

Frank Pfenning

15-150, April 23, 2020

Largely Following:
S. Owens, J. Reppy, and A. Turon. Regular Expression Derivatives Reexamined. Journal of Functional Programming 19(2):173-190, March 2009

Learning Objectives

- Review
  - Type-directed programming
  - Inductive and equational reasoning
  - Representation invariants
  - Higher-order functions
  - Restaging
  - Type classes
  - Functors
- Regular expression and automata
  - Brzozowski derivatives
  - Regular expression matching revisited
  - Optimization via algebraic laws for regular expressions
  - Deterministic finite-state automata (DFAs)
  - Compiling regular expressions to DFAs

Recalling Regular Expressions

- In Backus-Naur Form (BNF)

$$
\begin{array}{llll}
\text{Alphabet} & \Sigma & ::= & \{a_1, \ldots, a_n\} \\
\text{Words} & w & ::= & a \mid \epsilon \mid w_1\, w_2 \\
\text{Reg.Exps.} & r & ::= & a \mid 1 \mid 0 \mid r_1 \cdot r_2 \mid r_1 + r_2 \mid r^*
\end{array}
$$

- $\mathcal{L}(r)$, the language of r is a set of words

$$
\begin{array}{lcl}
\mathcal{L}(a) & = & \{a\} \\
\mathcal{L}(1) & = & \{\epsilon\} \\
\mathcal{L}(0) & = & \{\,\} \\
\mathcal{L}(r_1 \cdot r_2) & = & \{w_1\, w_2 \mid w_1 \in \mathcal{L}(r_1) \text{ and } w_2 \in \mathcal{L}(r_2)\} \\
\mathcal{L}(r_1 + r_2) & = & \{w \mid w \in \mathcal{L}(r_1) \text{ or } w \in \mathcal{L}(r_2)\} \\
\mathcal{L}(r^*) & = & \{w_1 \cdots w_n \mid \text{each } w_i \in \mathcal{L}(r)\} \\
 & = & \mathcal{L}(1 + r \cdot r^*)
\end{array}
$$

Examples

- String contains two consecutive a's (over $\Sigma = \{a, b\}$)

$$(a + b)^* \cdot a \cdot a \cdot (a + b)^*$$

- String contains no two consecutive a's

$$((a \cdot b) + b)^* \cdot (1 + a)$$

Brzozowski Derivatives

- Define $\partial_a(r)$ and $\text{nullable}(r)$ such that

$$\begin{aligned} a\,w \in \mathcal{L}(r) \quad &\text{iff} \quad w \in \mathcal{L}(\partial_a(r)) \\ \epsilon \in \mathcal{L}(r) \quad &\text{iff} \quad \text{nullable}(r) \end{aligned}$$

- Key: $\partial_a(r)$ is again a regular expression!
  - Brzozowski also allows $r_1 \& r_2$ and $\neg r$
  - Efficiency depends on size of $\partial_a(r)$
- In code

```
(* match : regexp -> char list -> bool *)
fun match r (a::w) = match (deriv a r) w
  | match r (nil) = nullable r
```

Computing nullable

- Recall: $\epsilon \in \mathcal{L}(r)$ iff nullable(r)

$$
\begin{array}{rcl}
\text{nullable}(a) & \text{iff} & \text{false} \\
\text{nullable}(1) & \text{iff} & \text{true} \\
\text{nullable}(0) & \text{iff} & \text{false} \\
\text{nullable}(r_1 \cdot r_2) & \text{iff} & \text{nullable}(r_1) \text{ and nullable}(r_2) \\
\text{nullable}(r_1 + r_2) & \text{iff} & \text{nullable}(r_1) \text{ or nullable}(r_2) \\
\text{nullable}(r^*) & \text{iff} & \text{true}
\end{array}
$$

Computing the Brzozowski Derivative

- Recall: $a\,w \in \mathcal{L}(r)$ iff $w \in \mathcal{L}(\partial_a(r))$

$$
\begin{array}{rcll}
\partial_a(c) & = & 1 & \text{if } a = c \\
 & = & 0 & \text{if } a \neq c \\
\partial_a(1) & = & 0 & \\
\partial_a(0) & = & 0 & \\
\partial_a(r_1 \cdot r_2) & = & \partial_a(r_1) \cdot r_2 & \text{if not nullable}(r_1) \\
 & = & \partial_a(r_1) \cdot r_2 + \partial_a(r_2) & \text{if nullable}(r_1) \\
\partial_a(r_1 + r_2) & = & \partial_a(r_1) + \partial_a(r_2) & \\
\partial_a(r^*) & = & \partial_a(r) \cdot r^* &
\end{array}
$$

- Time proportional to size of r
- Note size increase for $r_1 \cdot r_2$ and r^*

Examples

Let's Code!

Correctness Proof

- How do we prove the correctness of `nullable`, `deriv`, and `match`?
- `nullable` $r \Longrightarrow$ `true` iff $\epsilon \in \mathcal{L}(r)$, otherwise `false`
  - By induction over the structure of r
- If `deriv` $a\ r \Longrightarrow s$ then $a\,w \in \mathcal{L}(r)$ iff $w \in \mathcal{L}(s)$
  - By induction over the structure of r
- `match` $r\ w \Longrightarrow$ `true` iff $w \in \mathcal{L}(r)$, otherwise `false`
  - By induction over the structure of w (left to right)

The Algebra of Regular Expressions

- How can we avoid size explosion of regular expressions?
  - In practice, if not in theory
- Key idea: exploit their algebraic properties!
  - Regular expressions form a Kleene algebra
  - Can be derived from the definition of $\mathcal{L}(r)$
  - We only use some of the laws
- $(+, 0)$ form a commutative idempotent monoid

$$
\begin{array}{rcll}
(r + s) + t & = & r + (s + t) & \text{associativity} \\
0 + r & = & r & \text{left identity} \\
r + 0 & = & r & \text{right identity} \\
r + s & = & s + r & \text{commutativity} \\
r + r & = & r & \text{idempotence!}
\end{array}
$$

- Would like to use idempotence as much as possible

The Algebra of Regular Expressions, Continued

- More laws are better (if fast to use for simplification)
- $(\cdot, 1)$ form a monoid with annihilation by 0

$$
\begin{array}{rcll}
(r \cdot s) \cdot t & = & r \cdot (s \cdot t) & \text{associativity} \\
1 \cdot r & = & r & \text{left identity} \\
r \cdot 1 & = & r & \text{right identity} \\
0 \cdot r & = & 0 & \text{left annihilation} \\
r \cdot 0 & = & 0 & \text{right annihilation}
\end{array}
$$

- Some laws for Kleene $*$

$$
\begin{array}{rcll}
(r^*)^* & = & r^* & \text{idempotence} \\
1^* & = & 1 & \text{identity} \\
0^* & = & 1 & \text{identity} + \text{annihilation}
\end{array}
$$

Let's Code!

Deterministic Finite Automata (DFAs)

- An exceedingly simple computational model
- Usually defined as $M = \langle \Sigma, \mathcal{Q}, q_0, \mathcal{F}, \delta \rangle$
  - Σ is the alphabet
  - $\mathcal{Q}$ is a set of states q
  - $q_0 \in \mathcal{Q}$ is the initial state
  - $\mathcal{F} \subseteq \mathcal{Q}$ are the final states
  - $\delta : \mathcal{Q} \times \Sigma \to \mathcal{Q}$ is the transition function
- The automaton accepts $w = a_1\, a_2\, \ldots\, a_n \in \Sigma^*$ if there is a sequence of states $s_0, s_1 \ldots, s_n$ such that
  1. $s_0 = q_0$
  2. $s_{i+1} = \delta(s_i, a_{i+1})$ for $0 \leq i < n$
  3. $s_n \in \mathcal{F}$
- $\mathcal{L}(M) = \{w \mid M \text{ accepts } w\}$

Examples

Regular Expressions and DFAs

- $L = \mathcal{L}(M)$ for some DFA M iff $L = \mathcal{L}(r)$ for some regular expression r (over the same alphabet Σ)
- Key idea: compile a regular expression r to a DFA M accepting $\mathcal{L}(r)$
  - M executes very efficiently
  - M might be large (in theory, hopefully not in practice)
- Observe that `deriv` a r does not depend on input string!
  - Precompute all necessary derivatives
  - If we normalize there will only be finitely many! [Brzozowski 1964]

Compiling Regular Expressions

- Each derivative will be a state in a DFA
- Top-level r is initial state q_0
- If nullable(r) then r is a final state
- If `deriv` $a\ r = s$ then $\delta(r, a) = s$

Example

Let's Code

Summary

- Brzozowski derivatives
- Regular expression matching revisited
- Optimization via algebraic laws for regular expressions
- Deterministic finite-state automata (DFAs)
- Compiling regular expressions to DFAs by restaging
- This algorithm is effective in practice