Brzozowski Derivatives

Sam Penny

Churchill Computer Science Talk Series

19th October 2022

Introduction and Motivation

- In part IA of the course we were taught a method to construct a DFA from a regular expression, by constructing an NFA-ϵ and then using subset construction.
  - Results in large, inefficient DFAs.
- In this talk I will introduce the concept of a Brzozowski derivative - the derivative of a regular expression.
  - Constructs more efficient, often optimal DFAs.
  - Supports extended regular expressions.
  - Concise definition in functional languages

Regular expressions abstract syntax

Over a given alphabet Σ:

$$
\begin{aligned}
r, s ::= \; & \emptyset && \text{empty set} \\
\mid \; & \epsilon && \text{empty string} \\
\mid \; & a && a \in \Sigma \\
\mid \; & r + s && \text{union (logical or)} \\
\mid \; & r \cdot s && \text{concatenation} \\
\mid \; & r^* && \text{Kleene-closure (zero-or-more)} \\
\mid \; & r \& s && \text{logical and} \\
\mid \; & \neg r && \text{negation}
\end{aligned}
$$

The last two expressions make this type represent *extended* regular expressions.

Languages of regular expressions

The language of a regular expression r is a set of strings $L(r) \subseteq \Sigma^*$ generated by the following rules:

$$\begin{aligned}
L(\emptyset) &= \{\} \\
L(\epsilon) &= \{\epsilon\} \\
L(a) &= \{a\} \\
L(r + s) &= L(r) \cup L(s) \\
L(r \cdot s) &= \{u \cdot v \mid u \in L(r) \text{ and } v \in L(s)\} \\
L(r^*) &= \{\epsilon\} \cup L(r \cdot r^*) \\
L(r \& s) &= L(r) \cap L(s) \\
L(\neg r) &= \Sigma^* \setminus L(r)
\end{aligned}$$

What is a derivative of a language?

- The derivative of a language $L \subseteq \Sigma^*$ with respect to a string $u \in \Sigma^*$ is the language generated by stripping the leading u from the strings in L that start with u
- That is, $\partial_u L = \{v \mid u \cdot v \in L\}$.
- For example: $L = \{\text{unhappy}, \text{unusual}, \text{un}, \text{cat}, \text{dog}\}$, then $\partial_{un} L = \{\text{happy}, \text{usual}, \epsilon\}$
- A string s is within a language $L \iff$ the language $\partial_s L$ contains ϵ.

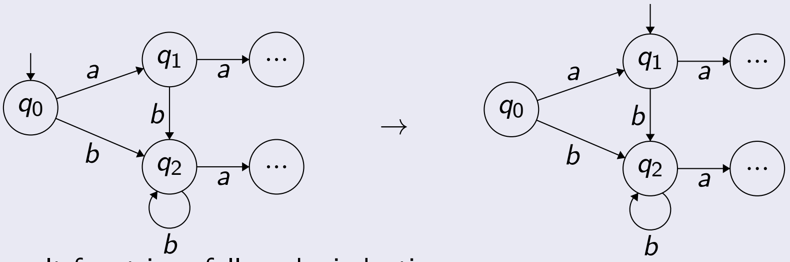
Derivatives of regular languages

Theorem

If $L \subseteq \Sigma^*$ is regular, then $\partial_u L$ is regular for all strings $u \in \Sigma^*$

Proof.

We start by showing that for any $a \in \Sigma$, the language $\partial_a L$ is regular.



The result for strings follows by induction.

Nullable languages

A language is *nullable* if it contains the empty string.

$$
\begin{aligned}
\text{nullable}(\emptyset) &= \text{false} \\
\text{nullable}(\epsilon) &= \text{true} \\
\text{nullable}(a) &= \text{false} \\
\text{nullable}(r + s) &= \text{nullable}(r) \text{ or } \text{nullable}(s) \\
\text{nullable}(r \cdot s) &= \text{nullable}(r) \text{ and } \text{nullable}(s) \\
\text{nullable}(r^*) &= \text{true} \\
\text{nullable}(r \& s) &= \text{nullable}(r) \text{ and } \text{nullable}(s) \\
\text{nullable}(\neg r) &= \neg\ \text{nullable}(r)
\end{aligned}
$$

Brzozowski Derivatives

Brzozowski defined the following rules to compute the derivative of a regular expresion with respect to a symbol a.

$$
\begin{aligned}
\partial_a \emptyset &= \emptyset \\
\partial_a \epsilon &= \emptyset \\
\partial_a b &= \begin{cases} \epsilon & \text{if } a = b \\ \emptyset & \text{if } a \neq b \end{cases} \\
\partial_a (r + s) &= \partial_a r + \partial_a s
\end{aligned}
$$

Brzozowski Derivatives - Concatenation

$$\partial_a(r \cdot s) = \begin{cases} \partial_a r \cdot s + \partial_a s & \text{if nullable}(r) \\ \partial_a r \cdot s & \text{otherwise} \end{cases}$$

Example:
let $t = (a + \epsilon) \cdot b$.
$L(t) = \{ab, b\}$
$\partial_a t = \epsilon \cdot b + \varnothing$
$\partial_b t = \varnothing \cdot b + \epsilon$

Brzozowski derivatives - Kleene-closure (Star)

$$\partial_a(r^*) = \partial_a r \cdot r^*$$

Example:
let $t = (a+b)^*$
$L(t) = \{a, b, ab, aa, bb, \dots\}$

$$\begin{aligned}\partial_a((a+b)^*) &= \partial_a(a+b) \cdot (a+b)^* \\ &= (\partial_a a + \partial_a b) \cdot (a+b)^* \\ &= (\epsilon + \emptyset) \cdot (a+b)^*\end{aligned}$$

Brzozowski derivatives - And & Not

$$\partial_a(r \mathbin{\&} s) = \partial_a r \mathbin{\&} \partial_a s$$
$$\partial_a(\neg r) = \neg(\partial_a r)$$

Brzozowski Derivatives (Cont.)

- These rules are extended to strings as follows:

$$\partial_{\epsilon} r = r$$
$$\partial_{ua} r = \partial_a(\partial_u r)$$

- For example, $\partial_{abc} r = \partial_c(\partial_b(\partial_a r))$
- This is analogous to a `fold_left` operation of the 'derive' function applied to a string.

Brzozowski Derivatives Implementation

```
type regex =
  |EmptySet
  |EmptyString
  |Character of char
  |Union of regex * regex
  |Concat of regex * regex
  |Star of regex
  |And of regex * regex
  |Not of regex

let rec nullable = function
  EmptySet -> false
  |EmptyString -> true
  |Character(_) -> false
  |Union(r, s) -> (nullable r) || (nullable s)
  |Concat(r, s) -> (nullable r) && (nullable s)
  |Star(_) -> true
  |And(r, s) -> (nullable r) && (nullable s)
  |Not(r) -> not (nullable r)
```

Brzozowski Derivatives Implementation (2)

```
let rec derive r c =
  match r with
    EmptySet -> EmptySet
    |EmptyString -> EmptySet
    |Character(c') -> if c = c' then EmptyString
                      else EmptySet
    |Union(r, s) -> Union (derive r c, derive s c)
    |Concat(r, s) ->  if nullable r then
                        Union (Concat (derive r c, s), derive s c)
                      else
                        Concat(derive r c, s)
    |Star(r) -> Concat (derive r c,  Star(r))
    |And(r, s) -> And(derive r c, derive r c)
    |Not(r) -> Not(derive r c)
```

Brzozowski Derivative Implementation (3)

```
let regexmatch r char_list =
  nullable (List.fold_left derive r char_list)
```

Problems With Current Implementation

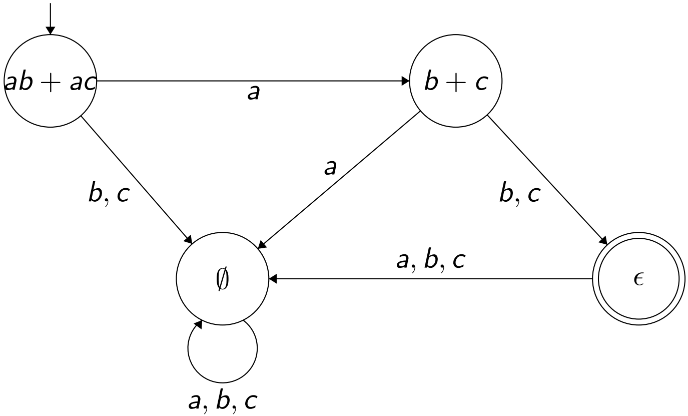
1. Testing against the same RE results in repeated work.
    - In this case, it makes more sense to build a DFA.
2. Huge growing complexity in the generated REs.
    - E.g $\epsilon \cdot r$.

Building a DFA

- We say that r and s are equivalent, written $r \equiv s$ whenever $L(r) = L(s)$.
- Each state of the constructed DFA is labelled with a RE r representing it's equivalence class - that is, $\{s \mid s \equiv r\}$.
- Construction:
  - Set the start state to be the initial RE.
  - Perform a depth-first search on the DFA.
  - Calculate transitions by taking the derivative of the current state.
  - Only introduce a new state when there are no equivalent states.

Example DFA Construction

$$\partial_b(b + c) = \epsilon + \emptyset$$
$$\partial_c(b + c) = \emptyset + \epsilon$$



Building a DFA (2)

- This algorithm is *guaranteed* to generate the minimal DFA.
- However, determining whether two REs are equivalent is too expensive to be practical.

Practical DFA construction

- For efficiency, instead we only introduce a new state when no *similar* state is present.
- *Similarity* is an approximation of RE equivalence.
- $\approx$ denotes the least relation on REs according to a set of rules, some of which are shown below.

$$
\begin{aligned}
r + r &\approx r \\
r + s &\approx s + r \\
(r + s) + t &\approx r + (s + t) \\
(r \cdot s) \cdot t &\approx r \cdot (s \cdot t) \\
\epsilon \cdot r &\approx r \\
\emptyset \cdot r &\approx \emptyset \\
(r^*)^* &\approx r^*
\end{aligned}
$$

Practical DFA Construction (2)

- We maintain the invariant that all REs are in $\approx$-canonical form.
- 'Smart'-constructor functions, e.g:

```
let concat r s =
  match r, s with
    EmptySet, _ -> EmptySet
    |_, EmptySet -> EmptySet
    |EmptyString, _ -> s
    |_, EmptyString -> r
    |Concat(_,_), _ -> Concat(s, r)
    |_, _ -> Concat(r, s)
```

Performance

- I will be comparing the size of state machines generated by ml-lex (a popular lexer generator for ML) and ml-ulex, a generator that uses Brzozowski derivatives.

Table 1. *Number of states (best results in **bold**)*

Lexer	`ml-lex`	`ml-ulex`	**Minimal**	**Description**
Burg	61	**58**	**58**	A tree-pattern match generator
CKit	122	**115**	**115**	ANSI C lexer
Calc	**12**	**12**	**12**	Simple calculator
CM	153	**146**	**146**	The SML/NJ compilation manager
Expression	**19**	**19**	**19**	A simple expression language
FIG	150	**144**	**144**	A foreign-interface generator
FOL	**41**	**41**	**41**	First-order logic
HTML	52	**49**	**49**	HTML 3.2
MDL	161	**158**	**158**	A machine-description language
ml-lex	121	**116**	**116**	The `ml-lex` lexer
Scheme	324	**194**	**194**	R^5RS Scheme
SML	251	**244**	**244**	Standard ML lexer
SML/NJ	169	**158**	**158**	SML/NJ lexer
Pascal	60	**55**	**55**	Pascal lexer
ml-yacc	100	**94**	**94**	The `ml-yacc` lexer
Russo	4803	3017	**2892**	System-log data mining
L_2	*n/a*	147	**106**	Monitoring stress-test

Figure: Example from *Regular expressions re-examined*

Further Reading I

- Brzozowski, Janusz A (1964). "Derivatives of regular expressions". In: *Journal of the ACM (JACM)* 11.4, pp. 481–494.
- Owens, Scott, John Reppy, and Aaron Turon (2009). "Regular-expression derivatives re-examined". In: *Journal of Functional Programming* 19.2, pp. 173–190.

Code examples: github.com/sam1penny/brzozowski-derivatives-talk