Reg.Ex → DFA

Originally:
e-mail to Bruce Litow
1992 January 16 16:12:31 -0600

The reason I’m interested in a good Reg.Ex. → DFA algorithm is that I’m trying to formalize a CFG parsing algorithm that I’ve been using by hand for a couple years.

It will accept a grammar with rules stated in the form

Non-Terminal = Regular-Expression(Terminals, Non-Terminals).

convert the regular expression on the right to a DFA thus resulting in something much like a recursive transition network.

Then it will eliminate arcs in the DFAs traversed by non-terminals, perform “stack symbol introduction”. There’s a whole body of rules for carrying out this transformation. The result is a PDA in a certain special form.

Then it will perform “lambda elimination” transformations for which, again, there’s a whole body of rules.

For example, the grammar

$$S \to \left(a|(S)\right)^{¿}.$$

When subjected to the first stage will yield the DFA given by the regular grammar

$$S = Q_0 \to aQ_0|(Q_1|1, \quad Q_1 \to SQ_2, \quad Q_2 \to)Q_0.$$

The non-terminal elimination phase will remove the arc involving the non-terminal S and replacing it as follows:

$$Q_1 \overset{S}{\to} Q_2 \Rightarrow \begin{cases} Q_1 \overset{\langle A|}{\to} S = Q_0 \\ Q_0 \overset{|A\rangle}{\to} Q_2 \end{cases}$$

The right end of the arcs for $\langle A|$ goes to the starting states associated with the non-terminal S, and the left end of the arcs for $|A\rangle$ go to the final states associated with the non-terminal. The meanings of these arcs are

(1) $Q_1 \overset{\langle A|}{\to} Q_0$: a λ -transition accompanied by pushing A onto the stack.

(2) $Q_0 \overset{|A\rangle}{\to} Q_2$: a λ -transition accompanied by popping A from the stack.

The pop action is blocked if the top of the stack does not match; hence it entails a test.

The corresponding “regular” grammar at the second stage thus becomes

$$S = Q_0 \to aQ_0|(Q_1||A\rangle Q_2|1, \quad Q_1 \to \langle A|Q_0, \quad Q_2 \to)Q_0.$$

As an example of a rule, one of many: if there were any other arcs anywhere in any of the other DFAs (assuming this example had more than one production) of the form

$$Q \overset{S}{\to} Q_2$$

it would perform the transformation above *using the same stack symbol* A .

The third stage eliminates some or all of the lambda transitions. First the $Q_1 \overset{\langle A|}{\to} Q_0$ transition

$$S = Q_0 \to aQ_0|(\langle A|Q_0||A\rangle Q_2|1, \quad Q_2 \to)Q_0.$$

Then the $Q_0 \overset{|A\rangle}{\to} Q_2$ transition

$$S = Q_0 \to aQ_0|(\langle A|Q_0||A\rangle)Q_0|1.$$

I can transform standard C syntax quite easily in this way into about a 60 state parser. But it takes a while to carry out the manipulations.

I tend to think of this automaton as an infinite DFA whose states are indexed by strings of stack symbols, so the automaton above would have the following arcs

$$(Q_0, w) \xrightarrow{\mathbf{a}} (Q_0, w), \quad (Q_0, w) \xrightarrow{\mathbf{(}} (Q_0, Aw), \quad (Q_0, Aw) \xrightarrow{\mathbf{)}} (Q_0, w) \quad ,$$

for $w \in A^{¿}$, with $(Q_0, 1)$ as both the start and final state.

So, for instance, for the word **(a(aa))** , one has the transition sequence

$$\rightarrow (Q_0, 1) \xrightarrow{\mathbf{(}} (Q_0, A) \xrightarrow{\mathbf{a}} (Q_0, A) \xrightarrow{\mathbf{(}} (Q_0, AA) \xrightarrow{\mathbf{a}} (Q_0, AA) \xrightarrow{\mathbf{a}} (Q_0, AA) \xrightarrow{\mathbf{)}} (Q_0, A) \xrightarrow{\mathbf{)}} (Q_0, 1) \rightarrow \quad .$$

All the transformation rules I use are motivated by this correspondence.

Converting Regular Expressions to DFA’s (subtle)

comp.theory
Part 1: **Re: Converting Regular Expressions to DFA’s (subtle)**
1992 January 14 14:45:06 GMT
Part 2: **Re: Converting Regular Expressions to DFA’s (subtle)**
1992 January 17 20:24:03 GMT
Part 3: **Re: An algorithm (was: Re: Converting Regular Expressions to DFA’s (subtle))**
1992 January 17 20:11:45 GMT

Part 1:

Recently, I was stumped when trying to come up with an efficient algorithm that directly converts regular expressions to minimal DFA’s without going through any intermediate stages. It sounds so easy, and it is in fact such an easy problem to work by hand, but an afternoon’s study reveals that it involved some fairly subtle concepts.

Somehow, these concepts seem central:

(a) Identities: 1 = empty string; 0 = error string (= empty set); (note: $a \mid 0 = a = 0 \mid a$, $a0 = 0 = 0a$, $a1 = a = 1a$)
(b) $A_0 = 1$ if A contains the empty string; $A_0 = 0$ else
(c) $x \backslash A = \{ a : xa \in A \}$
(d) Partition theorem: $A = A_0 \mid a(a \backslash A) \mid b(b \backslash A) \mid \ldots \mid z(z \backslash A)$ A; where $a, b, \ldots, z$ are the input symbols of the language.

The algorithm would iterate (b) and (c) on the regular expression in question and thus generate the minimal DFA, somehow keeping track of and eliminating duplicates on the fly.

Is there a detailed description or outline, or even software available on-line that carries out this conversion?

I have used this method to generate DFA’s directly from regular expressions, but not minimal DFA’s. The reason is that I know of no easier way of determining identity of regular expressions than to generate mininal DFA’s and compare these.

Thus, I used an approximate identity relation: if the relation holds, the regular expressions are equal, but the may be equal even if the relation doesn’t hold. You can make this relation as precise as you like, but adding precision adds complexity. The only important requirement is that you will get a finite number of states for any regular expression. I could not prove that this was the case, but I found no counter examples. I found that the DFA’s I obtained were in most cases identical to those generated by the method in section 3.9 of Aho,Sethi & Ullmans “Dragon book”. Sometimes I got better results, sometimes it got better results.

From Torben AEgidius Mogensen (torbenm@diku.dk):

The paper "minimal model generation" by Bouajjani, Fernandez & Halbwachs from the Workshop on Computer-Aided Verification, June 1990 presents a method for generating minimal DFA's directly. It involves manipulating boolean functions by binary decision diagrams.

Part 2:

From raoult@irisa.fr (Jean-Claude Raoult):

*Another similar method is to use a finer equivalence on regular expressions, essentially associativity, commutativity, idempotence of set union (+) and distributivity over concatenation. There you get Brozowski's algorithm. Berry & Sethi (Theor. Comp. Sci. **48** (1986) :*

Actually, as per the algorithm I described yesterday, you'll only need the following properties

$$\begin{array}{lll} a^* = 1 + aa^*, & (a+b)\,c = ac + bc, & 1a = a, \\ x = x1, & (ab)\,c = a(bc), & (a+b)+c = a+(b+c) \end{array}$$

(where x denotes an input symbol) to get minimal DFA's under the condition Berry and Sethi describe (distinct input symbols).

Part 3:

(Algorithm deleted)

The one problem with this method is that you still have to perform subset construction and minimilization.

Try finishing off the example above following this description...

I forgot to mention, despite the apparent complexity of the example used in the description of the algorithm, you'll only end up generating a few states (5 I think).

The following property also holds:

If every symbol in the regular expression E is distinct (call them $x_1, x_2, \ldots, x_n$), then the resulting finite automaton is minimal. In equation form, every state can be represented as a finite sum taken out of the set $\{1, x_1Q_1, x_2Q_2, \ldots, x_nQ_n\}$ where $Q_1, Q_2, \ldots, Q_n$ are (not necessarily distinct) states *uniquely* determined by the symbols $x_1, x_2, \ldots, x_n$. Compare with the method described in Berry and Sethi ("From Regular Expressions to Deterministic Automata" Theoretical Computer Science 1986).

Their example: $(a_1b_2 \mid b_3)^* b_4a_5$ with the algorithm I described previously will yield the automaton below (listed in equation form) with the starting state X_0:

$$\begin{array}{lllll} X_0 = & a_1X_2 \mid & & b_3X_0 \mid \quad b_4X_1, & \\ X_1 = & & & & a_5X_3, \\ X_2 = & & b_2X_0, & & \\ X_3 = & 1 & & & \end{array}$$

and finite sums taken out of the set $\{1, a_1X_2, b_2X_0, b_3X_0, b_4X_1, a_5X_3\}$.