CS 164 : Programming Languages and Compilers Lecture 3 - 1/25/17

Notes on NFA-to-DFA Translation

Professor: Koushik Sen *Guest: Alex Reinking*

We covered a lot of ground today, so I wanted to make sure everyone had a reference for the NFA to DFA material. I followed the exposition from "Modern Compiler Implementation in ML" [1]

The basic intuition is that when we're simulating (cf. executing) an NFA, we're tracking all the possible states we could be in, then when we process the next character, we follow the corresponding transitions for each state in our set, and then we follow all the ϵ-transitions from those.

The actual translation from NFA to DFA follows the same logic, just by repeating that process for every possible letter. This has to terminate because the automata are finite, and there are finitely (although exponentially) many subsets of NFA states. So, without further ado, here are the definitions and algorithms from today's lecture:

- s is an NFA state
- $c \in \Sigma$ is a character
- $edge(s, c) = \{\text{all NFA states reachable via } c \text{ from } s\}$
- S is a set of NFA states
- $closure(S) = T$ where T is the smallest set such that $T = S \cup \left(\bigcup_{s \in T} edge(s, \epsilon)\right)$

We can compute $closure(S)$ iteratively by starting with $T' = S$ and adding everything in $edge(t, \epsilon)$ to T' for every $t \in T'$. Once this process stops adding states to T', you have T.

Now, remember that the states in the equivalent DFA are sets of states in the NFA. We will use d to denote these. We can simulate an NFA by defining

$$DFAedge(d, c) = closure\left(\bigcup_{s \in d} edge(s, c)\right) \tag{1}$$

and running the following algorithm:

```
d = closure({q0})
for ch in input_string:
    d = DFAedge(d, ch)
accept if d contains an NFA final state
```

To actually produce a DFA – which could be exponentially large, but is typically linear in practice – we use this next algorithm:

```
S = [closure({q0})]
p = 0
j = 0
T = <dfa-table>
while j <= p:
```

```
    foreach c in alphabet:
        e = DFAedge(S[j], c)
        if e in S and its index is i:
            T[j][c] = i
        else:
            p += 1
            S[p] = e
            T[j][c] = p
    j += 1
output T
```

Running this by hand typically involves creating a table for the NFA, then building up a table for the DFA, as we did in lecture. Don't forget to take the closure all the way – you follow ϵ as many times as you need to, not just one hop.

References

[1] Andrew W. Appel. *Modern Compiler Implementation in ML: Basic Techniques*. Cambridge University Press, New York, NY, USA, 1997.