Applications of Subset Constructors in nondeterminism

By: Eun Il Kim and Andre Wong

What is nondeterminism? Pros and Cons

Pros

- Nondeterminism is in which a certain point of a program is branched between multiple choices/paths points and applies them all to obtain the result. Some choices will fail and some will succeed.
- It is easier to read/follow the path of an NFA due to less states used.

Cons

- Harder to construct due to the multiple branching paths.
- According to Robert W. Floyd's "Non-Determinsitic Algorithms," nondeterminism can cause inconsistencies as results can vary based on time and sequence known as race condition. (running a program can cause different results based on sequence and time.
- To avoid inconsistent results, we convert nondeterminism to determinism.

Why convert nondeterminism to determinism?

- Although it is easier to understand an NFA, DFA is easier to build.
- DFA is always consistent since it only branches to one unique path for every state.
- Because every set of NFA is a DFA, it is possible to convert NFA to DFA via subset algorithm.

- Problem: DFA uses many states to cover all the subset of the NFA, making it harder to read.

Subset algorithm

- As quoted in J. Howard Johnson

and Derick Wood's "Instruction Computation

In Subset Construction," for every possible

set of NFA states are the subset of DFA states

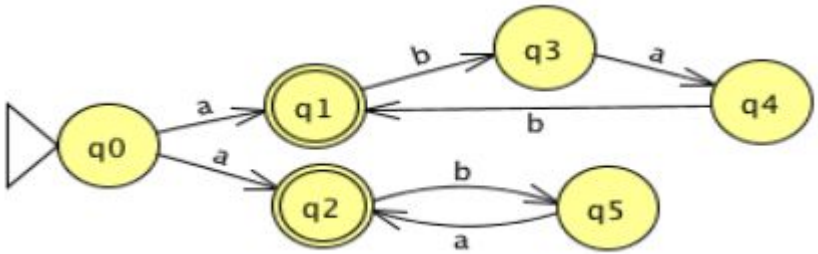
- Therefore, a DFA is a union of all

NFA transition sets.

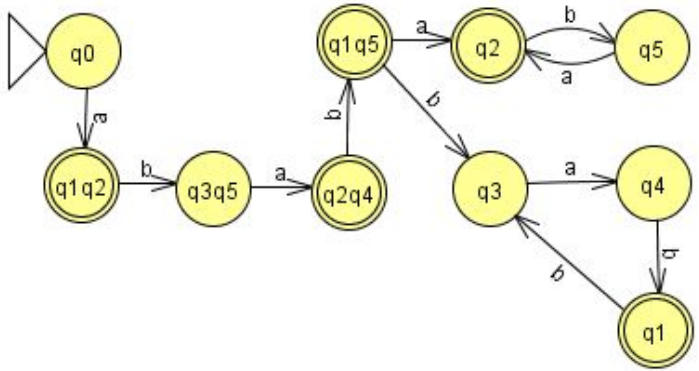
Algorithm Subset Construction
On entry: A nondeterministic finite-state machine, $M = (Q, \Sigma, \delta, s, F)$.
On exit: A deterministic finite-state machine, $M' = (Q', \Sigma, \delta', s', F')$
that satisfies $L(M) = L(M')$.
begin
Let $Q_0 = \{s\}$ be the zeroth subset of Q and 0 be the
corresponding state in Q', let i be 0, and let $last$ be 0;
while $i \leq last$
do begin
for all a in Σ
do if $\{(p, a, q) : p \in Q_i\} \neq Q_j$, for some j, $0 \leq j \leq last$
then begin $last := last + 1$;
$\delta' := \delta' \cup \{(i, a, last)\}$;
$Q_{last} := \{q : p \in Q_i \text{ and } (p, a, q) \in \delta\}$
end;
$i := i + 1$
end {while};
$Q' := \{i : 0 \leq i \leq last\}$;
$F' := \{i : Q_i \cap F \neq \emptyset \text{ and } 0 \leq i \leq last\}$
end

Our Scheme Code

- Before explaining the scheme program, the program will convert NFA to DFA

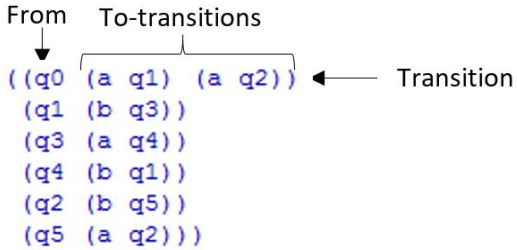


NFA



DFA

General Structure of NFA



- The general structure of the NFA is shown here. The NFA as well as DFA can be broken down into a group of sequences which will contain a list of transitions made up of a from-state, to-transition, and to-state which will be described during conversion process.

Conversion Process: Main Function

```
(define (construct-dfa nfa) (display "\nDFA Computations\n")
   (define (iter-dfa dfa stack)
     (let ((current-state (get-test stack)))
       (map display (list dfa "\n"))
       (cond((null? current-state) dfa)
            (else (let* ((a-list (find-to-states nfa current-state 'a))
                         (b-list (find-to-states nfa current-state 'b))
                         (new-stack (update-stack dfa stack a-list b-list))
                         (new-dfa (list(append-dfa current-state a-list b-list))))
                    (if (null? dfa)
                        (iter-dfa new-dfa new-stack)
                        (iter-dfa (append dfa new-dfa) new-stack)))))))
  (iter-dfa '() (list (list (caar nfa)))))
```

- The main function (construct-dfa) that will construct the DFA from NFA. ‘nfa’ will be converted to a newly sequenced ‘dfa’ and ‘stack’ holds new possible sequences. (a-list, b-list, new-stack, new-dfa are temporary variables to store the values from later functions)

Conversion Process: Simple Procedure

- Procedure (get-test) determines if there will be a state to test on found at the ‘stack’. This state is called temporary state called ‘current-state.’
- (get-test) will return first list in the given ‘stack,’ otherwise return null if there is no list and the value is returned

is placed in ‘current-state.’

- The main function (construct-dfa)

will halt depending if the ‘current-state’

is null, otherwise resume testing.

```
(define (get-test stack)
   (cond ((null? stack) '())
         (else (car stack))))

(define (pop stack)
  (cond ((null? stack)'())
        (else (cdr stack))))

(define (update-stack dfa stack a-list b-list)
  (let* ((current-stack (pop stack))
         (nonempty-stack? (not (null? current-stack)))
         (input-a? (not (exist-in-dfa? dfa a-list)))
         (input-b? (not (exist-in-dfa? dfa b-list))))
    (cond((and nonempty-stack? input-a? input-b?) (list current-stack a-list b-list))
         ((and input-a? input-b?) (list a-list  b-list))
         (nonempty-stack? (append current-stack
                                  (if input-a? (list a-list) '())
                                  (if input-b? (list b-list) '())))
         (else (list(append (if nonempty-stack? current-stack '())
          (if input-a? a-list '())
          (if input-b? b-list '()))))
        )))

(define (get-to-states nfa state input)
  (define (get-help mylist)
    (cond ((null? mylist)'())
          ((eq? (caar mylist) input) (cons (cadar mylist) (get-help (cdr mylist))))
          (else (get-help (cdr mylist)))))
  (get-help (find-to nfa state)))
```

Conversion Process: Complex Procedure

```
((q0 (a q1) (a q2))
 (q1 (b q3))
 (q3 (a q4))
 (q4 (b q1))
 (q2 (b q5))
 (q5 (a q2)))
```

The main function (construct-dfa) will make several temporary variables.

- a-list and b-list.
- Both lists will be a newly formed to-state, which will be added to ‘dfa’ using the procedure (find-to-states).

```
(else (let* ((a-list (find-to-states nfa current-state 'a))
             (b-list (find-to-states nfa current-state 'b))
```

```
(q1 q2 (b q3 q5))
```

(find-to-states) will test each atom of ‘current-state’ in (get-to-states), which will then be appended to temp-state variable.

```
(define (find-to-states nfa current-state input)
    (cond ((null? current-state) '())
          (else (let ((temp-state (get-to-states nfa (car current-state) input)))
                  (cond ((null? temp-state) (find-to-states nfa (cdr current-state) input))
                        (else (append temp-state (find-to-states nfa (cdr current-state) input)))
                  )))))
```

Conversion Process: Complex Procedure (Cont'd)

The (get-to-states) will use the procedure (find-to).
The (find-to) will return all to-transition that matches and 'state'.

- The 'mylist' parameter contains 'nfa.'

```
((q0 (a q1) (a q2))
 (q1 (b q3))
 (q3 (a q4))
 (q4 (b q1))
 (q2 (b q5))
 (q5 (a q2)))
```

```
(q0 (a q1 q2))
```

```
(define (get-to-states nfa state input)
  (define (get-help mylist)
    (cond ((null? mylist)'())
          ((eq? (caar mylist) input) (cons (cadar mylist) (get-help (cdr mylist))))
          (else (get-help (cdr mylist)))))
  (get-help (find-to nfa state)))
```

The (get-help), found in within (get-to-states) will then utilize (find-to).
'mylist' contains to-transition
Will return a list of to-state that matches 'state' and 'input'.

```
(define (find-to mylist state)
  (cond ((null? mylist) '())
        ((eq? (caar mylist) state) (cdar mylist))
        (else (find-to (cdr mylist) state))))
```

Conversion Process: Complex Procedure (Cont'd)

In (construct-dfa), 'a-list' and 'b-list will now hold to-transition for respective input.

```
(else (let* ((a-list (find-to-states nfa current-state 'a))
             (b-list (find-to-states nfa current-state 'b))
```

```
(q0 (a q1 q2))
```

```
(define (find-to-states nfa current-state input)
   (cond ((null? current-state) '())
         (else (let ((temp-state (get-to-states nfa (car current-state) input)))
                 (cond ((null? temp-state) (find-to-states nfa (cdr current-state) input))
                       (else (append temp-state (find-to-states nfa (cdr current-state) input)))
                       )))))

(define (get-to-states nfa state input)
  (define (get-help mylist)
    (cond ((null? mylist)'())
          ((eq? (caar mylist) input) (cons (cadar mylist) (get-help (cdr mylist))))
          (else (get-help (cdr mylist)))))
  (get-help (find-to nfa state)))


(define (find-to mylist state)
  (cond ((null? mylist) '())
        ((eq? (caar mylist) state) (cdar mylist))
        (else (find-to (cdr mylist) state))))
```

Conversion Process: Helper functions

Then stack will be updated with (update-stack) to push ‘a-list’ and ‘b-list’ into the ‘stack’

```
(else (let* ((a-list (find-to-states nfa current-state 'a))
             (b-list (find-to-states nfa current-state 'b))
             (new-stack (update-stack dfa stack a-list b-list
```

(update-stack) will pop ‘stack’ and check if ‘a-list’ or ‘b-list’ exists in ‘stack’ using the procedure (exist-in-dfa?)

```
(define (exist-in-dfa? dfa state)
  (cond((null? state) #t)
       (else (find-from? dfa state))))
```

(exist-in-dfa?) then calls (find-from?) to break down each transition in ‘dfa’
If it is not null, check if ‘state’ is an atom or a list of from-state. Then call (get-from)

```
(define (update-stack dfa stack a-list b-list)
  (let* ((current-stack (pop stack))
         (nonempty-stack? (not (null? current-stack)))
         (input-a? (not (exist-in-dfa? dfa a-list)))
         (input-b? (not (exist-in-dfa? dfa b-list))))
    (cond((and nonempty-stack? input-a? input-b?) (list current-stack a-list b-list))
         ((and input-a? input-b?) (list a-list  b-list))
         (nonempty-stack? (append current-stack
                                  (if input-a? (list a-list) '())
                                  (if input-b? (list b-list) '())))
         (else (list(append (if nonempty-stack? current-stack '())
                            (if input-a? a-list '())
                            (if input-b? b-list '()))))
         )))
```

```
(q0 (a q1 q2))
```

```
(define (find-from? mylist state)
  (cond ((null? mylist) #f)
        ((list? state)
         (cond((equal? (get-from (car mylist)) state) #t)
              (else (find-from? (cdr mylist) state))))
        (else (cond((eq? (caar mylist) state) #t)
                   (else (find-from? (cdr mylist) state))))
        ))
```

Conversion Process: Helper functions (Cont'd)

(State list?)
If 'state' is a list, (find-from?) will compare to another list of from-state in 'dfa.'
(get-from) will extract a list of from-state from a single list of transition in 'my-list' from 'dfa.'
(get-from) will then cons each atom until it sees a list.

```
(q0 (a q1 q2)) (q1 q2 (b q3 q5))
```

Or state is atom?
If the list of from-state returned from (get-from) matches the list of from-state or 'state' in (find-from?), return #t. Else, cdr down 'my-list' or 'dfa.'

```
(define (get-from mylist)
  (cond ((list? (car mylist)) '())
        (else (cons (car mylist) (get-from (cdr mylist))))))

(define (find-from? mylist state)
  (cond ((null? mylist) #f)
        ((list? state)
         (cond((equal? (get-from (car mylist)) state) #t)
              (else (find-from? (cdr mylist) state))))
        (else (cond((eq? (caar mylist) state) #t)
                   (else (find-from? (cdr mylist) state))))
        ))
```

Conversion Process: Helper functions (Cont'd)

The (update-stack) will return an update stack

```
(else (let* ((a-list (find-to-states nfa current-state 'a))
             (b-list (find-to-states nfa current-state 'b))
             (new-stack (update-stack dfa stack a-list b-list
```

```
(define (exist-in-dfa? dfa state)
  (cond((null? state) #t)
       (else (find-from? dfa state))))
```

```
(define (find-from? mylist state)
  (cond ((null? mylist) #f)
        ((list? state)
         (cond((equal? (get-from (car mylist)) state) #t)
              (else (find-from? (cdr mylist) state))))
        (else (cond((eq? (caar mylist) state) #t)
                   (else (find-from? (cdr mylist) state))))
        ))
```

```
(q0 (a q1 q2)) (q1 q2 (b q3 q5))
```

```
(define (get-from mylist)
  (cond ((list? (car mylist)) '())
        (else (cons (car mylist) (get-from (cdr mylist))))))

(define (find-from? mylist state)
  (cond ((null? mylist) #f)
        ((list? state)
         (cond((equal? (get-from (car mylist)) state) #t)
              (else (find-from? (cdr mylist) state))))
        (else (cond((eq? (caar mylist) state) #t)
                   (else (find-from? (cdr mylist) state))))
        ))
```

Conversion Process: Helper functions (Cont'd)

- (append-dfa) requires the *from-state*'s 'current-state' and *to-transition*'s 'a-list' and 'b-list.'
- 'a-list' and 'b-list' can both be null, but one must exist, otherwise it will return null.
- The multiple conditions in (append-dfa) assesses how the list should append depending if one of the list is null.

```
(q0 (a q1 q2)) (q1 q2 (b q3 q5))
```

'new-stack' to the next iteration of main.

```
(else (let* ((a-list (find-to-states nfa current-state 'a))
             (b-list (find-to-states nfa current-state 'b))
             (new-stack (update-stack dfa stack a-list b-list))
             (new-dfa (list(append-dfa current-state a-list b-list))))

(define (append-dfa current-state a-list b-list)
  (cond ((not (or (null? a-list) (null? b-list)))
         (append current-state (list (flatten(list 'a a-list))) (list (flatten(list 'b b-list)))))
        ((not (null? a-list))
         (append current-state (list (flatten(list 'a a-list)))))
        ((not (null? b-list))
         (append current-state (list (flatten(list 'b b-list)))))
        (else '())))
```

Conversion Process: New Accept States

- After the DFA is built, the new states must also have new recognizable accepting states using the procedure (new-accept-output).
- (new-accept-output) will cdr the ‘dfa’ and find all *from-state* that contains any of the atom within ‘accept-states.’
- (get-from) is then called to extract each *from-state* from a transition in ‘dfa’ and input it in the ‘from-states’ variable.

```
(q0 (a q1 q2)) (q1 q2 (b q3 q5))
```

```
(define (new-accept-output dfa accept-states)
  (cond ((null? dfa) '())
        (else (let ((from-states (get-from (car dfa))))
                ;ormap returns the (or (map func list))
                (cond ((ormap (lambda (e) (member? e from-states)) accept-states)
                       (cons from-states (new-accept-output (cdr dfa) accept-states)))
                      (else (new-accept-output (cdr dfa) accept-states)))))))
```

Output of Scheme Code

```
NFA
((q0 (a q1) (a q2)) (q1 (b q3)) (q3 (a q4)) (q4 (b q1)) (q2 (b q5)) (q5 (a q2)))

Accept State:(q1 q2)

DFA Computations
()
((q0 (a q1 q2)))
((q0 (a q1 q2)) (q1 q2 (b q3 q5)))
((q0 (a q1 q2)) (q1 q2 (b q3 q5)) (q3 q5 (a q4 q2)))
((q0 (a q1 q2)) (q1 q2 (b q3 q5)) (q3 q5 (a q4 q2)) (q4 q2 (b q1 q5)))
((q0 (a q1 q2)) (q1 q2 (b q3 q5)) (q3 q5 (a q4 q2)) (q4 q2 (b q1 q5)) (q1 q5 (a q2) (b q3)))
((q0 (a q1 q2)) (q1 q2 (b q3 q5)) (q3 q5 (a q4 q2)) (q4 q2 (b q1 q5)) (q1 q5 (a q2) (b q3)) (q2 (b q5)))
((q0 (a q1 q2)) (q1 q2 (b q3 q5)) (q3 q5 (a q4 q2)) (q4 q2 (b q1 q5)) (q1 q5 (a q2) (b q3)) (q2 (b q5)) (q3 (a q4)))
((q0 (a q1 q2)) (q1 q2 (b q3 q5)) (q3 q5 (a q4 q2)) (q4 q2 (b q1 q5)) (q1 q5 (a q2) (b q3)) (q2 (b q5)) (q3 (a q4)) (q5 (a q2)))
((q0 (a q1 q2)) (q1 q2 (b q3 q5)) (q3 q5 (a q4 q2)) (q4 q2 (b q1 q5)) (q1 q5 (a q2) (b q3)) (q2 (b q5)) (q3 (a q4)) (q5 (a q2)) (q4 (b q1)))
((q0 (a q1 q2)) (q1 q2 (b q3 q5)) (q3 q5 (a q4 q2)) (q4 q2 (b q1 q5)) (q1 q5 (a q2) (b q3)) (q2 (b q5)) (q3 (a q4)) (q5 (a q2)) (q4 (b q1)) (q1 (b q3)))


DFA
((q0 (a q1 q2)) (q1 q2 (b q3 q5)) (q3 q5 (a q4 q2)) (q4 q2 (b q1 q5)) (q1 q5 (a q2) (b q3)) (q2 (b q5)) (q3 (a q4)) (q5 (a q2)) (q4 (b q1)) (q1 (b q3)))

New Accept States:((q1 q2) (q4 q2) (q1 q5) (q2) (q1))
```

- After running the scheme code with our desired NFA, it will go through the process of conversion and result the DFA and it’s new accepting states.

Time Complexity of the code

- The time complexity of the code is polynomial time of O(n^3) due to the many procedures being either recursive or iterative which will result between O(n) or O(nlogn) in the worst case complexity.

Conclusion

- The nondeterministic finite automata goes through multiple paths at the same time, causing randomness and we do not want that.
- To remove the randomness, convert it to deterministic finite automata that goes through one path at a time which will always return the same result every time.
- The subset construction helps us with that and it is an effective algorithm that removes the inconsistencies of a nondeterministic machine into a singular solidified resultant of a deterministic machine.
- Functional programming allowed us to create a powerful interpreter that can analyze the structure of a collection of lists.
- Ultimately, the subset algorithm was able to assist us in creating a solid NFA to DFA conversion programming in Scheme.

What we could have improved.

- Amb-Eval: Amb-Evaluator would return several paths that could either succeed or fail, but it would not be practical as there are infinitely repeating sequences that could still lead to the correct path.
- Improve Time Complexity: The time complexity could also be improved by enhancing the complexity of several procedures (find-to-state) and (update-stack) as they have polynomial time of O(n^2) by themselves.

Citations

Floyd, Robert W. “Non-Deterministic Algorithms.” *Http://Repository.cmu.edu/*, Carnegie Institute of Technology, Nov. 1966, http://repository.cmu.edu/cgi/viewcontent.cgi?article=2787&context=compsci.

Johnson, J Howard, and Derick Wood. “Instruction Computation in Subset Construction.”*CiteSeerX*, Natural Sciences and Engineering Research Council of Canada, from the Information Technology Research Centre of Ontario, and from the Research Grants Committee of Hong Kong., June 1996, http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.49.9630&rep=rep1&type=pdf.

Rabin, Michael O, and Dana Scott. “Finite Automata and Their Decision Problem's.” *Finite Automata and Their Decision Problem's*, Apr. 1959, pp. 114–125. www.cse.chalmers.se/~coquand/AUTOMATA/rs.pdf.