ESCOLA POLITÉCNICA DA UNIVERSIDADE DE SÃO PAULO

Departamento de Engenharia de Computação e Sistemas Digitais

Lógica Computacional – 4MA 2007

EP 2

Fabio Correia Kung - 4957439
Nataniel Simon - 4061458

Linguagem específica para definição de autômatos

Com base em [1], foi criada uma linguagem específica (*DSL – Domain Specific Language*) para a definição dos autômatos. Tal funcionalidade é permitida através de macros; recurso disponível em Scheme.

Macros permitem um modelo de desenvolvimento conhecido como meta-programação, onde programas são escritos para gerarem outros programas ("programas que escrevem programas").

A macro retirada de [1], com pequenas modificações, possibilita que autômatos sejam definidos da seguinte forma:

```
(define automato-simples
  (automaton init ; estado inicial
             ; estados
             [init :
                   (c -> more)]
             [more :
                   ; transicoes de cada estado
                   (a -> more)
                   (d -> more)
                   (r -> end)]
             ; estado de aceitacao
             [end :
                  accept]))
```

A nova linguagem permite a definição de autômatos finitos de forma simples e natural. Inicia-se definindo o nome do autômato e o estado inicial. A seguir são declarados os estados e cada uma das transições pertencentes a cada estado.

Há ainda a palavra-chave especial "accept" que indica um estado de aceitação.

Um programa para escrever outro programa

Cada um dos estados de um autômato, pode ser representado por uma função em scheme. Estas funções lêem um caracter da cadeia e chamam outra função, dependendo do caracter lido, representando as transições deste estado.

Uma possível representação para cada um dos estados pode ser como se segue:

```
(lambda (stream) ; um estado é uma função em scheme
  (cond
    [(empty? stream ) false] ; rejeita a cadeia se for vazia
    [else
     (case (first stream ) ; lê um símbolo da cadeia
       [(a) ( more (rest stream ))] ; transicoes
       [(d) ( more (rest stream ))] ; estados alvo são more e end
       [(r) ( end (rest stream ))]
       [else false])])) ; rejeita caso não haja transição possível
```

Um autômato pode ser definido como um conjunto de funções em scheme e deve indicar o estado inicial (a primeira função a ser chamada).

Porém, esta representação é pouco natural e intuitiva para a definição de autômatos, já que Scheme não é uma linguagem que foi criada para a definição de autômatos.

Felizmente há um recurso na linguagem conhecido como macros, que permite que criemos nossa própria "linguagem" dentro do ambiente Scheme. A macro deve ser responsável por ler um autômato como definido no item anterior e gerar a representação necessária em Scheme (para cada estado uma função como a definida acima).

Para tal tarefa, foi utilizada a macro disponível em [1]. Mais detalhes sobre a macro utilizada podem ser encontrados neste artigo.

Suportando autômatos finitos não determinísticos

Com base em [2], conclui-se que um simulador de autômatos finitos não determinísticos é suficiente para a simulação de autômatos finitos não determinísticos. Na ocasião de não-determinismo (estado que possibilite a transição para mais de um estado com o mesmo símbolo) basta considerar cada uma das transições possíveis como a execução de um novo autômato finito determinístico. É necessário que apenas um dos novos autômatos finitos determinísticos reconheça a linguagem.

Foram necessárias pequenas modificações na macro anterior para suportar a execução de autômatos finitos não determinísticos. O primeiro passo é suportar a definição de transições que levam a mais de um estado:

```
(define automato-nao-deterministico
  (nd-automaton init ; estado inicial
                [init :
                      ; o simbolo c pode levar tanto para
                      ; o estado more quanto para moref
                      (c -> more moref)]
                [more :
                      (a -> more)
                      (d -> more) ; transicoes simples
                      (r -> end)]
                [moref :
                       (c -> moref)
                       ; novamente o nao determinismo
                       ; (transicoes multiplas p/ mesma entrada)
                       (f -> moref end)]
                [end : ; estado de aceitacao
                     accept]))
```

O segundo passo foi modificar a macro para disparar todas as transições associadas de um estado dado um símbolo e verificar se algum dos caminhos reconhece a linguagem. Decidiu-se por avaliar todas os caminhos possíveis por profundidade (e não por largura) o que facilitou bastante a implementação, bastando o uso de um `(or caminho1 caminho2 ...)`.

Exemplos de simulação

Autômato finito determinístico

```
; Exemplo de definicao de um automato finito deterministico
; que reconhece cadeias da linguagem "c(ad)*r"
(define cadeia-tipo-cdar?
  (automaton init ; estado inicial
             ; estados
             [init :
                   (c -> more)]
             [more :
                   ; transicoes de cada estado
                   (a -> more)
                   (d -> more)
                   (r -> end)]
             ; estado de aceitacao
             [end :
                  accept]))

; Exemplos de chamada
> (cadeia-tipo-cdar? '(c a d a d r))
> #t
> (cadeia-tipo-cdar? '(c a d a r d r))
> #f
```

Autômato finito não determinístico

```
; Exemplo de definicao de um automato finito nao deterministico
; que reconhece cadeias da linguagem "c(ad)*r || c(cf)*f"
; O nao determinismo eh definido quando um mesmo simbolo
; leva a dois diferentes estados
(define reconhece?
  (nd-automaton init ; estado inicial
                [init :
                      ; o simbolo c pode levar tanto para o estado
                      ; more quanto para moref!
                      (c -> more moref)]
                [more :
                      (a -> more)
                      (d -> more) ; transicoes simples
                      (r -> end)]
                [moref :
                       (c -> moref)
                       ; novamente o nao determinismo
                       ; (transicoes multiplas p/ mesma entrada)
                       (f -> moref end)]
                [end : ; estado de aceitacao
                     accept]))

; Exemplos de chamada
(reconhece? '(c r)) ; #t
(reconhece? '(c a d a d r)) ; #t
(reconhece? '(c a d a r d r)) ; #f
(reconhece? '(c a f r)) ; #f
(reconhece? '(c f)) ; #t
(reconhece? '(c f f)) ; #t
(reconhece? '(c f f c)) ; #f
(reconhece? '(c f f f)) ; #t
```

obs.: o programa exibirá mais informações de simulação passo a passo, que aqui foram omitidas.

Bibliografia

1. S. Krishnamurti. Automata via Macros **Journal of Functional Programming**, Volume 16 , Issue 3 (May 2006), pp. 253-267. (*download* de dentro da Poli direto da página da **JFP**).
2. S. Krishnamurti, M Felleisen, B. F. Duba. From Macros to Reusable Generative Programming Lecture Notes in Computer Science, 1799, Springer-Verlag, 1999, pp. 105-120. (download de dentro da Poli direto da página da LNCS)
3. C Wagenknecht, PD Friedman. Teaching Nondeterministic and Universal Automata
4. Using SCHEME Computer Science Education, vol. 8, Issue 3, 1998, p.197-227. (download em: http://www.inf.hs-zigr.de/~wagenkn/TI/Automaten/paper-final.pdf)