CÉSAR EDUARDO CAVANI GARANHANI

UM AMBIENTE PARALELO PARA IMPLEMENTAÇÃO DE MODELOS ADAPTATIVOS

São Paulo
2008

CÉSAR EDUARDO CAVANI GARANHANI

UM AMBIENTE PARALELO PARA IMPLEMENTAÇÃO DE MODELOS ADAPTATIVOS

Dissertação apresentada ao Departamento de Engenharia Elétrica da Universidade de São Paulo para obtenção do título de mestre em Engenharia de Computação

Área de concentração: Tecnologias Adaptativas
Orientador: Prof. Dr. Ricardo Luis de Azevedo Rocha

São Paulo
2008

FICHA CATALOGRÁFICA

Garanhani, César Eduardo Cavani
Um ambiente paralelo para implementação de modelos adaptativos / C.E.C. Garanhani. -- São Paulo, 2008.
74p.

Dissertação (Mestrado) – Escola Politécnica da Universidade de São Paulo. Departamento de Engenharia de Computação e Sistemas Digitais.

1.Teoria dos autômatos I.Universidade de São Paulo. Escola Politécnica. Departamento de Engenharia de Computação e Sistemas Digitais II.t.

RESUMO

Neste trabalho, é proposto um modelo de execução em paralelo dos autômatos finitos adaptativos, visando melhor eficiência destes dispositivos. Desta forma, é apresentada uma comparação de tempo de execução do modelo seqüencial e o modelo proposto.

Além disso, é apresentado um ambiente de desenvolvimento de autômatos finitos adaptativos paralelos. Com o objetivo de padronizar e facilitar a implementação destes dispositivos, optou-se por adotar uma linguagem de alto nível para a implementação, o *Scheme*. Esta é uma linguagem funcional que permite a utilização de mecanismos de paralelização, o que torna a escrita de programas paralelos mais direta.

Palavras-chave: autômatos adaptativos, computação paralela e linguagem funcional

ABSTRACT

It is presented in this paper a model for parallel execution of the finite-state adaptive automaton, arguing that this kind of execution could be more efficient than the traditional sequential one. For this discussion, a comparison between the two models of execution is shown in this paper.

Also, a framework for parallel adaptive finite-state automata development is presented. This framework was designed to facilitate the creation and execution of the device for researchers. For this purpose, the language Scheme was used, for it is a functional language, with high level of abstraction, that includes features for parallelizing computation

Keywords: adaptive automaton, parallel computing and functional language

À minha esposa Regiane

Sumário

1 Introdução

Muito se tem estudado sobre os autômatos adaptativos e suas aplicações. Esse poderoso modelo tem se mostrado eficaz em muitas áreas da ciência, como, por exemplo, na área de controle de navegação de robôs autônomos (ALMEIDA, 2000), na inteligência artificial na aprendizagem e reconhecimento de padrões (ROCHA, 2000a), na construção de modelos de algoritmos genéticos (PISTORI, 2005), na construção de compiladores (LUZ, 2003) ou ainda no reconhecimento sintático da gramática da língua portuguesa. (MENEZES, 2002, NETO; MORAES, 2003).

Além de seu poder de computação, que tem equivalência expressiva com a máquina de Turing (ROCHA, 2000a), esse modelo de computação apresenta algumas vantagens sobre os modelos existentes, sendo importante citar:

- É eficiente e de fácil visualização, pois é baseado em modelos de máquinas de estados, autômatos finitos, já há muito utilizados (NETO, 1993 – pp. 112-116).
- É capaz de aprender como lidar com novas construções sintáticas (NETO, 1993).
- Pode lidar com a dependência de contexto de forma estritamente sintática (NETO, 1993).

A falta de um padrão de implementação desses modelos dificulta sua utilização. Os modelos adaptativos construídos e referenciados nesta pesquisa foram todos desenvolvidos *ad hoc*, evidenciando que a ausência de uma metodologia de desenvolvimento pode atrapalhar o uso da adaptatividade em larga escala.

Em relação à questão de desempenho computacional dos programas desenvolvidos, incluindo aplicativos para usuários finais, este é um fator de grande relevância, que pode ser ilustrado pela crescente construção de arquiteturas *multicore* para usuários domésticos. Portanto, uma das formas de se obter melhor desempenho é a construção de programas que utilizem arquiteturas paralelas (SUTTER, 2005).

A busca por paralelização de programas vem sendo muito estudada ultimamente, pois a necessidade de aumento da eficiência já supera a capacidade computacional de sistemas monoprocessados. Além disso, o avanço tecnológico na área de hardware tem feito com que pesquisadores da área de software busquem, cada vez mais, ferramentas de implementação e execução de programas em paralelo, aproveitando, assim, todo o poder computacional oferecido por tais sistemas (SUTTER, 2005).

Dentre os paradigmas de linguagens de programação existentes, o mais utilizado é o paradigma imperativo, baseado na arquitetura de von Neuman. Outro paradigma bastante utilizado, ao menos no ambiente acadêmico, é o paradigma funcional, que se baseia no formalismo do cálculo lambda (SEBESTA, 2003).

Assim, a proposta desse trabalho é mostrar uma possível implementação de autômatos adaptativos aproveitando os recursos disponíveis para o aumento de eficiência dos autômatos, utilizando uma linguagem de alto nível e discutindo a dificuldade de implementação desse modelo computacional no paradigma funcional.

1.1 Justificativa

No meio acadêmico, muitas discussões dão enfoque ao formalismo dos dispositivos adaptativos gerais desde que foram apresentados em (NETO, 2001). Diversos modelos e dispositivos foram criados e descritos formalmente, de forma que atualmente existem dispositivos adaptativos baseados em autômatos, tabelas, árvores de decisão. Embora algumas implementações tenham surgido, até o momento desta discussão, não foi ainda definido formalmente o processo de implementação adequado aos dispositivos adaptativos. Há um trabalho (CAMOLESI, 2007) que procurou cobrir essa lacuna desenvolvendo um método associado a um dispositivo para definição de dispositivos adaptativos, seguindo a definição formulada em (NETO, 2001). Para utilização da ferramenta, é necessário conhecimento aprofundado em adaptatividade.

Dessa forma, uma ferramenta de apoio ao desenvolvimento de modelos adaptativos que prescinda de conhecimentos técnicos aprofundados em tecnologia adaptativa

ajudaria esses grupos de estudo, que passariam a reduzir parte do tempo despendido na implementação desses modelos.

Outra motivação para este trabalho é o estudo da utilização do modelo de autômatos finitos adaptativos executando em paralelo. Isto porque, no caso dos autômatos finitos não determinísticos, o modelo paralelo pode ser, em teoria, mais eficiente que o modelo seqüencial, pois os diversos caminhos de execução podem ser processados paralelamente, se aproximando do modelo teórico de autômatos.

1.2 Objetivos

O autômato adaptativo foi idealizado com o objetivo de se construir um dispositivo baseado em uma estrutura simples, como o autômato de estados finitos, mas com capacidade de resolver problemas mais complexos. Um exemplo dessa característica pode ser a realização de todos os passos sintáticos da compilação de um código, incluindo sintaxe dependente de contexto, usando o modelo de autômatos (NETO, 1993).

Apesar de ser um modelo de fácil compreensão e de uso abrangente, a implementação dos autômatos adaptativos, na prática, pode apresentar algumas dificuldades. Por ser um modelo de autômatos, sua implementação exige o uso de alguns mecanismos, como técnicas para tratamento de não-determinismo.

Além disso, o modelo de autômato adaptativo possui mecanismos que alteram a estrutura do dispositivo em tempo de execução: as funções adaptativas. Essas funções são executadas através de chamadas anexadas nas transições do dispositivo, alterando a estrutura deste durante sua execução, criando estados e adicionando ou removendo transições, segundo sua base teórica em Neto (2001). A implementação deste mecanismo pode ser complexa, pois não há um padrão de implementação definido para tal, apesar de suas regras teóricas serem bem definidas.

O objetivo principal deste trabalho é discutir e propor um modelo de execução em paralelo do modelo de autômatos adaptativos. Tal proposta leva em conta estudos na

área de computação paralela e padrões de execução em paralelo de modelos simbólicos da computação.

É proposto também, como objetivo secundário deste trabalho, um *framework* de implementação para criação e execução dos autômatos finitos adaptativos. O *framework* proposto foi desenvolvido utilizando uma linguagem de alto nível, cujo *design* facilita a implementação de programas baseados em computação simbólica, a fim de agilizar a construção de programas que fazem uso deste modelo.

1.3 Estrutura do texto

Os sete capítulos que compõem este texto visam argumentar quanto à escolha das técnicas e ferramentas utilizadas para evolução do trabalho, bem como sua relevância.

Os primeiros capítulos fazem uma revisão bibliográfica nas áreas de dispositivos adaptativos (cap. 2), programação funcional (cap. 3) e computação paralela e programação funcional paralela (cap. 4).

O capítulo 5 apresenta a proposta deste trabalho: um modelo de execução de autômatos adaptativos em paralelo. Discute-se, também, as características da implementação de tal modelo.

O capítulo 6 descreve o desenvolvimento de um programa utilizando o framework construído.

Finalmente o capítulo 7 apresenta as considerações finais do trabalho.

2 Dispositivos adaptativos

Dispositivos baseados em regras são dispositivos formais usados para descrever e modelar sistemas encontrados na vida real. Esses dispositivos servem de base para os dispositivos adaptativos, porque estes usam os dispositivos baseados em regras como estruturas subjacentes para seu funcionamento.

Em cada estágio dos dispositivos baseado em regras, estes se apresentam em uma determinada configuração, a qual representa a situação da máquina em relação à cadeia de entrada e ao seu controle (estado corrente, por exemplo). O comportamento futuro da máquina é determinado a partir de sua configuração atual, independentemente de como ela foi alcançada, e dos estímulos de entrada ainda não processados. Ou seja, o comportamento de um dispositivo baseado em regras nada mais é que a movimentação de uma configuração a outra, sucessivamente, como resposta aos estímulos de entrada.

Dessa forma, os *dispositivos adaptativos* são também dispositivos baseados em regras, que funcionam como seu dispositivo subjacente, e que possuem a capacidade de alterar sua estrutura durante a execução, sem interferência de agentes externos. Essas alterações são feitas através de *ações adaptativas*, definidas no próprio dispositivo (PISTORI, 2003).

2.1 Autômatos finitos adaptativos

O *autômato finito adaptativo* (AFA) é um dispositivo formal auto-modificável que tem como mecanismo subjacente o *autômato de estados finitos* (ou *autômato finito* - AF). Além disso, esse dispositivo possui em sua definição uma camada adaptativa, na qual são definidas as regras de como o dispositivo subjacente será alterado durante a execução do autômato finito adaptativo, como definido em Pistori (2003). Nessa camada adaptativa do autômato finito adaptativo estão definidas as *funções adaptativas*, através das quais o autômato finito subjacente sofrerá as alterações em sua estrutura.

Por ser um dispositivo com o poder computacional de uma Máquina de Turing (PISTORI, 2003) e por ser construído sobre um dispositivo subjacente simples e de grande importância no meio científico, os autômatos finitos adaptativos foram escolhidos como objeto de estudo deste trabalho.

Um autômato finito adaptativo é descrito como um dispositivo formal que evolui a partir de um autômato finito inicial. Baseando-se em Pistori (2003), pode-se representar um autômato finito adaptativo como $AFA = (AF, \Phi)$, onde Φ é a camada adaptativa de AFA e $AF = (Q, \Sigma, q_0, F, T)$ um autômato finito, no qual:

- Q é o conjunto de estados do modelo inicial do autômato. $q_0 \in Q$ é o estado inicial do autômato;
- Σ é o conjunto de símbolos de entrada;
- $\mathrm{F} \subseteq \mathrm{Q}$ é o conjunto de estados de aceitação do autômato; e
- $\mathrm{T} \subseteq (\mathrm{Q} \times \Sigma \times \mathrm{Q})$ é o conjunto de transições do autômato finito.

Dessa maneira, a definição do autômato finito adaptativo se dá por $AFA = (Q, \Sigma, q_0, F, T, Q_\infty, \Gamma)$, onde

- $\mathrm{Q}, \Sigma, \mathrm{q}_0$ e F definem autômato finito inicial;
- Q_∞ é o conjunto de todos os possíveis estados que o dispositivo pode assumir;
- Γ é o conjunto das funções adaptativas do dispositivo; e
- $\mathrm{T} \subseteq (\Gamma \times Q_\infty \times \Sigma \times Q_\infty \times \Gamma)$ é o conjunto de transições do autômato finito adaptativo inicial.

O funcionamento dos autômatos finitos adaptativos é feito por mudanças de configuração do autômato finito em função de suas transições. Ou seja, Dado uma configuração atual, o dispositivo irá consumir o símbolo mais a esquerda da cadeia de entrada e executar as transições que preenchem os seguintes requisitos: (1) o estado

inicial da transição deve ser o mesmo que o estado corrente do autômato finito; e (2) o símbolo da transição deve ser o mesmo símbolo consumido da cadeia de entrada pelo dispositivo.

Na execução de uma transição $t_i = (a, c_i, s, c_j, b)$, se houver alguma ação adaptativa a ser executada, isto é, se a e/ou b for uma chamada para uma função adaptativa (e não vazia – ε), então esta função será executada, alterando o conjunto de regras (transições) T do dispositivo.

2.2 Função adaptativa

O mecanismo através do qual o autômato finito adaptativo altera o conjunto de regras do dispositivo subjacente é a chamada de função adaptativa. O funcionamento deste mecanismo é mostrado em Neto (1993), e formalizado por Pistori (2003). Na formalização feita por Pistori, no entanto, as funções adaptativas são definidas como um conjunto de ações elementares apenas, sendo que na descrição deste mecanismo em Neto, além das ações elementares, é descrito também um mecanismo para chamada de funções dentro das funções adaptativas.

Para os objetivos deste trabalho, faz-se necessário definir a execução das funções adaptativas de forma diferente daquela utilizado por Pistori (2003), ou seja, retomando a definição original de Neto (1993). Para isso, utiliza-se os algoritmos 2.1, 2.2 e 2.3 descrito a seguir.

Além disso, as ações elementares apresentadas em Pistori são mostradas como uma tripla, que representa a transição do dispositivo subjacente do autômato finito adaptativo, desprezando as ações adaptativas anterior e posterior. Neste trabalho, as ações elementares de busca, remoção e adição de transições serão representadas por uma quíntupla, a mesma que representa uma transição $t \in T$, mostrada acima. O motivo principal para tal diferenciação é que, dadas as transições $t_1 = (f(\ ), c_1, s, c_2, g(\ ))$ e $t_2 = (\varepsilon, c_1, s, c_2, \varepsilon)$, a ação elementar

$$?\,[\varepsilon, ?\,q_1, s, c_2, \varepsilon]$$

onde $?q_1$ é uma variável da função adaptativa, resultará apenas na transição t_2. Ou seja, é possível diferenciar transições também através das chamadas das funções adaptativas, anterior e/ou posterior.

Na função adaptativa, a declaração de variáveis é implícita. Por isso, para identificarmos uma variável nestas funções, usamos um ponto de interrogação precedendo o nome da variável.

Além disso, o *bind* delas é feito nas ações elementares de pesquisa e ocorre na primeira ocorrência dela numa dessas ações elementares. Uma vez instanciada, o valor da variável é substituído nas outras ações elementares (adição e remoção).

É possível que, numa ação elementar de pesquisa que contenha uma variável não instanciada, haja mais de uma transição que satisfaça as condições da pesquisa. Neste caso, a variável é instanciada com cada um dos possíveis valores, e as ações de remoção e inserção que contém essa variável executa para cada um desses valores. Dessa forma, é possível executar um laço de código.inserção e remoção.

Caso não haja nenhuma ocorrência de uma variável numa ação elementar de busca, ou se não houver nenhuma transição que satisfaz as condições da pesquisa, as ações elementares de remoção ou inserção que fazem referência a essa variável não são executadas.

Como as variáveis, também os parâmetros da função adaptativa são implícitos. Neste caso, a representação deve ser feita pelo símbolo % seguido da posição do parâmetro passado na chamada da função (%1, %2, ..., %n).

Os algoritmos 2.1, 2.2 e 2.3 mostram em detalhe a execução das funções adaptativas. É importante notar neste algoritmo a característica descrita acima.

O primeiro algoritmo (2.1) mostra a execução das funções adaptativas, especificando a ordem de execução das ações elementares (primeiramente, são executadas as ações de busca, depois de remoção e por fim, de adição de transições), e o momento de chamada das funções adaptativas (linhas 02, 03 e 04, e 33, 34 e 35)

Algoritmo 2.1 Execução da função adaptativa

```
entrada:
   Transições do autômato T = t_1, ..., t_k
   Ações elementares de consulta    A_C = c_1, ..., c_l
   Ações elementares de remoção     A_R = r_1, ..., r_m
   Ações elementares de inserção    A_I = i_1, ..., i_n
   Funções adaptativas anteriores   F_a = a_1, ..., a_o
   Funções adaptativas posteriores  F_d = b_1, ..., b_p
   Lista de parâmetros              P = p_1, ..., p_q
saída:
   Lista T' com as transições do autômato após modificações
01: T' ← T
02: para x = 1 até o faça
03:   T' ← a_x(q_1, ..., q_r) {Executa as funções adaptativas anteriores, passando os
      parâmetros necessários, atribui o valor da execução a T'}
04: fim para
05: Substitui parâmetros em A_C, A_R e A_I usando P
06: se l > 0 então
07:   S ← ExecutaConsultas(T, A_C, ∅)
08:   se S ≠ ∅ então
09:     para j = 1 até m faça
10:       para todo s ∈ S faça
11:         Aplica substituição s sobre r_j
12:         T' ← T' − r_j
13:       fim para
14:     fim para
15:     para j = 1 até n faça
16:       para todo s ∈ S faça
17:         Aplica substituição s sobre i_j
18:         Gera um novo símbolo para cada diferente gerador de i_j e efetua
            substituições correspondentes
19:         T' ← T' + i_j
20:       fim para
21:     fim para
22:   fim se
23: senão {m = 0}
24:   Remover de A_R e A_i todas as ações contendo variáveis
25:   para j = 1 até m faça
26:       T' ← T' − r_j
27:   fim para
28:   para j = 1 até n faça
29:       Gera um novo símbolo para cada diferente gerador de i_j e efetua
          substituições correspondentes
30:       T' ← T' + i_j
31:   fim para
32: fim se
33: para x = 1 até p faça
34:   T' ← b_x(q_1, ..., q_r) {Executa as funções adaptativas posteriores, passando
      os parâmetros necessários, atribui o valor da execução a T'}
35: fim para
```

Nos algoritmos 2.2 e 2.3, é mostrado o comportamento da função de busca das transições e instanciação das variáveis da função adaptativa.

Algoritmo 2.2 Função $ExecutaConsultas(T, A_C, s)$

```
entrada:
   Transições do autômato          T = t_1, ..., t_k
   Ações elementares de consulta   A_C = c_1, ..., l
   Lista s de variáveis istanciadas até o momesno
saída:
   Lista S de substituições {S tem escopo global e é inicalizado com vazio}
1: S ← ∅
2: se n_C = 0 então {Foi encontrada uma substituição que satisfaz a consulta}
3:    S ← S ∪ {s}
4: senão
5:    para i = 1 até n faça
6:       se s' ← Unifica(c_1, t_i, s) então
7:          ExecutaConsultas(T, A_C − c_1, s')
8:       fim se
9:    fim para
10:fim se
```

Algoritmo 2.3 Função $Unifica(c, t, s)$

```
entrada:
   Lista c com os termos da consulta   c = c_1, ..., c_m
   Lista t com os termos da transição  t = t_1, ..., t_m
   Lista s com as variáveis instanciadas (binded) até o momento
saída:
   Valor verdade indicando se c e t são unificáveis
   Lista s0 com variáveis instanciadas após unificação
01: s' ← s
02: Retorno ← VERDADEIRO
03: para j = 1 até m faça
04:    se c_j é uma variável então
05:       se c_j aparece instanciada em s' então
06:          c_j ← valor de c_j em s' {c_j deixa de ser uma variável}
07:       senão
08:          s' ← s ∪ {c_j/t_j}
09:          c_j ← t_j
10:       fim se
11:    fim se
12:    se c_j ≠ t_j então
13:       Retorno ← FALSO
14:    fim se
15: fim para
```

2.3 Exemplo de funcionamento

Um exemplo clássico de autômato adaptativo é o autômato que resolve a linguagem $L(AA) = \{a^n b^n c^n \mid n \geq 1\}$. Essa linguagem é classificada como sensível a contexto, o que significa que os modelos clássicos de autômatos não conseguem representá-lo.

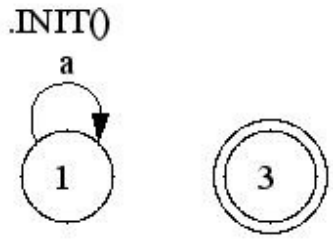

Figura 2.1 Autômato adaptativo que resolve a linguagem $a^n b^n c^n$. Configuração inicial.

Função adaptativa:

```
F :
{
  g*  // geradores. Cada gerador
  h*  // instancia um novo estado.
  j*:
  −[F(%1,%2,%3),%1,a,%1,ε];
  −[ε,%1,b,%2,ε];
  −[ε,%2,b,3,ε];
  −[ε,%3,c,4,ε];
  +[ε,%1,a,g*,ε];
  +[ε,g*,a,%2,ε];
  +[ε,%2,b,*h,ε];
  +[ε,h*,b,3,ε];
  +[ε,%3,c,j*,ε];
  +[ε,j*,c,4,ε];
  +[F(g*,h*,j*),g*,a,g*,ε];
}
```

Figura 2.2 Função adaptativa *F*.

A representação formal desse modelo se dá por $AFA = (Q, \Sigma, q_0, F, T, Q_\infty, \Gamma)$, onde:

- $\mathrm{Q} = \{1,2,3,4\}$
- $q_0 = 4$
- $\Sigma = \{a, b, c\}$
- $F = \{4\}$
- $\Gamma = \{\mathrm{F}\}$

Por ser um modelo simples, baseado no modelo de autômatos finitos, os autômatos finitos adaptativos têm aplicações em várias áreas da computação. Suas aplicações iniciaram-se no campo da implementação de linguagens de programação, e evoluíram gradativamente, hoje compreendendo muitas outras áreas, tais como processamento de linguagens naturais, robótica, representação do conhecimento, resolução automática de problemas, inferência gramatical, e outros.

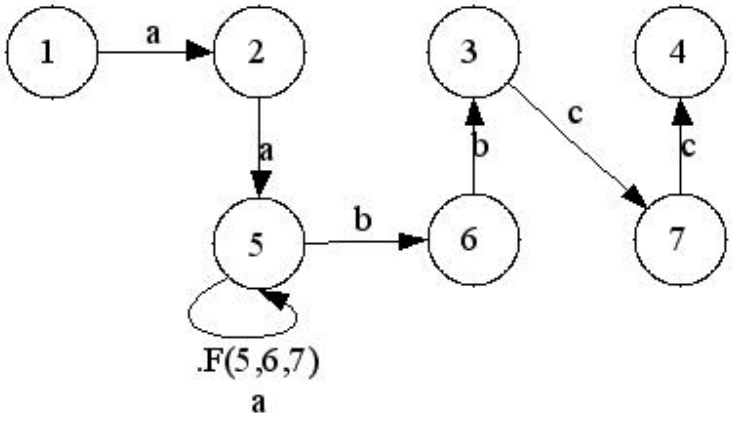

Figura 2.3 Autômato adaptativo que resolve a linguagem $a^n b^n c^n$.
Configuração após a cadeia entrada aabbcc

Em uma dessas aplicações, o modelo adaptativo é usado no campo da inteligência artificial como um dispositivo para inferência, aplicado tanto a linguagens quanto a

outras áreas do conhecimento, como por exemplo, problemas de decisão (PISTORI, 2003).

3 Programação funcional

Por volta das décadas de 60 e 70, as linguagens de programação imperativas recebiam uma série de críticas por seu formato e desempenho dos programas objeto gerados pelos compiladores da época. De fato, essas linguagens surgiram para suprir a necessidade da época, de desenvolver sistemas para computadores baseadas na arquitetura de *von Neumann*, que foi concebida por volta dos anos 40 e era uma solução elegante e prática que simplificou um grande número de problemas de engenharia e de programação (BACKUS, 1978). Atualmente, apesar de as condições que produziram tal arquitetura tenham mudado, a noção de computador ainda é baseada nessa idéia.

O modelo de arquitetura de computador de *von Neumann*, em sua forma mais simples, é composto de três partes: uma unidade de processamento central (CPU), uma unidade de armazenamento (ou memória) e um "canal" que liga os dois, fazendo a comunicação, palavra por palavra, entre eles. Essa comunicação entre a CPU e a memória pode ser dita um gargalo, pois, mesmo que a unidade de processamento seja extremamente eficiente, essa comunicação sempre limitará a eficiência do sistema como um todo (BACKUS, 1978).

As linguagens de programação convencionais são, basicamente, versões complexas de alto nível do modelo de computador de *von Neumann* (BACKUS, 1978). Apesar de novos paradigmas de linguagens de programação terem sido propostos, as linguagens de programação continuam utilizando a idéia básica das linguagens imperativas de uma instrução por vez. Em sua concepção, essas linguagens utilizam variáveis para simular as células de armazenamento da máquina; estruturas de controle transcrevem suas instruções de *jump* e de teste; operações de cópia imitam o *bind*, armazenamento e aritmética dela (SEBESTA, 2003). Assim, as instruções de atribuição seriam o gargalo de *von Neumann* das linguagens imperativas, o que limita, também, nossa estrutura lógica de construção de programas em uma instrução a ser executada por vez.

Além disso, a atribuição nas linguagens imperativas divide a programação em dois mundos. Um deles é o mundo ordenado das expressões, onde existem propriedades algébricas úteis (apesar de essas propriedades serem, geralmente, destruídas por efeitos colaterais). É o mundo onde acontece a computação mais relevante (BACKUS, 1978).

O outro mundo é o mundo das instruções, o qual tem como principal instrução a atribuição. Este é um mundo desordenado, com poucas propriedades matemáticas úteis, onde todas as instruções da linguagem existem apenas para se poder executar uma computação que deve ser baseada na primitiva de atribuição (BACKUS, 1978).

Em contraste às linguagens imperativas, o paradigma funcional é voltado para a programação de sistemas baseada numa linguagem matemática formal para manipulação simbólica, de caráter funcional. Isto é, qualquer computação neste paradigma corresponde a uma avaliação de uma ou mais funções, que formam a estrutura da programação funcional. Assim, um programa funcional nada mais é que um conjunto de funções.

Essa característica da programação funcional libera a lógica de programação do modelo imperativo, sendo que a construção de um programa naquele paradigma não se dá por *como* executar a computação, mas *o que* e de que forma matemática deve ser computado.

No paradigma funcional, as funções são os valores de primeira importância. Elas podem possuir parâmetros como valores de entrada, e sempre possuem um valor de saída (retorno). Assim, a execução de um programa é o mapeamento dos valores de saída de funções nos valores de entrada de uma função.

Por essas características, em teoria, as linguagens funcionais são mais concisas, contém nível mais alto de abstração, estão em concordância com raciocínio e análise formais e são mais fáceis de implementar para serem executados em arquiteturas paralelas. Apesar de algumas dessas características serem subjetivas e, por isso,

discutíveis, outras fazem dessas linguagens ótimas opções como ferramentas de desenvolvimento (HUDAK, 1989).

3.1 Cálculo lambda

O cálculo lambda é uma sintaxe para expressões e conjunto de regras de reescrita para transformação de expressões (BARENDREGT, 1997), desenvolvida por Church entre as décadas de 30 e 40 (GORDON, 1988). Este modelo matemático formal define aspectos computacionais das funções. Dessa forma, importantes conceitos presentes em linguagens de programação, como escopo e ordem de avaliação, por exemplo, estão presentes em sua sintaxe (HUDAK, 1989).

O cálculo lambda generaliza as funções ao seu máximo: funções podem ser aplicadas sobre elas mesmas. Dessa forma, ele permite recursão, sem ter que escrever uma definição formal para tal. Além disso, essa característica não tira a consistência do cálculo lambda como um modelo matemático, ou seja, não há contradições ou paradoxos no modelo (BARENDREGT, 1984), (HUDAK, 1989).

Esse modelo matemático consiste de definições de funções e regras de substituição de variáveis, chamadas de redução (GORDON, 1988). Sua sintaxe abstrata é mínima, porém poderosa, sendo definida por expressões lambda. A expressão lambda é construída recursivamente a partir de um conjunto de variáveis, podendo assumir uma das três formas:

- x
- $(\lambda x.M)$
- (MN)

A expressão $(\lambda x.M)$ é uma abstração onde x é a variável e M é o corpo da expressão, enquanto que (MN) é uma aplicação da expressão M sobre N. Assim, toda ocorrência de x em M na abstração será substituída por N na aplicação (MN). Por convenção, uma aplicação é associativa à esquerda. Portanto, uma expressão $(MN\,P)$ é equivalente a $\big((MN)P\big)$.

Na ocorrência de uma variável x, dizemos que x é uma variável livre se ela não se encontra no alcance (escopo) de λ. Caso contrário, dizemos que ela é uma variável ligada. Para ilustrar melhor a definição de variáveis livre, digamos que $vl(M)$ são as variáveis livres da expressão M. Neste caso:

- $vl(x) = \{x\}$
- $vl(M\ N) = vl(M) \cup vl(N)$
- $vl(\lambda x.M) = vl(M) - \{x\}$

Representamos uma substituição da forma $[N/x]M$, que representa a substituição variável x da expressão M pela expressão lambda N. Assim, é possível representar as regras de redução do cálculo lambda através de substituições. Pode-se definir que as substituições nas expressões lambda ocorrem da seguinte forma:

$$[M/x_i]x_j = \begin{cases} M, & se\ i = j \\ x_j, & se\ i \neq j \end{cases}$$

$$[M/x](N\ P) = ([M/x]N)([M/x]P)$$

$$[M/x_i](\lambda x_j.N) = \begin{cases} \lambda x_j N, & se\ i = j \\ \lambda x_j.[M/x_i]N, & se\ i \neq j\ e\ x_j \notin vl(M) \\ \lambda x_k.[M/x_i]([x_k/x_j]N), & caso\ contrário, \\ & onde\ k \neq i, k \neq j\ e\ x_k \notin vl(M) \cup vl(N) \end{cases}$$

A última regra resolve qualquer problema de conflito de nome trocando o nome dos identificadores conflitantes.

1. Conversão-α: permite a mudança de nome das variáveis ligadas. Diz-se que, termos que diferem apenas de uma conversão alfa, são equivalentes entre si.

$$\lambda x_j.M \Leftrightarrow \lambda x_i.[x_j/x_i]M, \qquad onde\ x_j \notin vl(M)$$

2. Conversão-β: exprime a idéia de uma aplicação de função.

$$(\lambda x.M)N \Leftrightarrow [N/x]M$$

3. Conversão-η: exprime a idéia de que duas expressões são a mesma se, e somente se, o resultado delas for igual para todos os argumentos dados a elas.

$$\lambda x.(M\ x) \Leftrightarrow M, \qquad se\ x \notin vl(M)$$

Por ser um formalismo simples e poderoso, o cálculo lambda serve como base para a maioria das linguagens funcionais. Uma dessas linguagens foi escolhida como ferramenta para este trabalho: *Scheme*.

3.2 Scheme

Scheme é um dialeto recursivo e com escopo estático da linguagem de programação LISP desenvolvida por um grupo do M.I.T. (Massachusetts Institute of Technology). Ela foi projetada para ter uma semântica desobstruída e simples, com poucas maneiras diferentes de se formar expressões, tornando a programação mais rápida e simples (FELLEISEN, 2003).

Scheme apresenta algumas características específicas do cálculo lambda, das quais vale ressaltar (ABELSON, 2007):

- As funções são tratadas como **objetos de primeira classe**, ou seja, uma função pode ser definida a qualquer momento durante a execução do programa, passada como argumento para outras funções, ou ser retornada como valor de outra função.

- Por conseqüência das funções serem objetos de primeira classe, elas são ditas **funções de ordem superior**. Funções de ordem superior são funções que manipulam funções. Para tal, elas tomam outras funções como valores de entrada, ou seu valor de saída é uma função.

- A linguagem possui **escopo estático** (ou **escopo léxico**), na qual o escopo da variável é calculado em tempo de compilação.

- As funções podem ser polimórficas, ou seja, o valor de retorno da função pode ser qualquer tipo suportado pela linguagem. Por exemplo, uma mesma função

pode retornar um valor numérico ou outra função, dependendo de uma condição.

Funções de ordem superior são poderosos mecanismos para formulação abstrações. Por exemplo, considere as definições a seguir, para ordenar listas:

```
(define split
  (lambda (lst)
    (letrec ([split-h
              (lambda (lst ls1 ls2)
                (cond
                  [(or (null? lst) (null? (cdr lst)))
                   (cons (reverse ls2) ls1)]
                  [else
                   (split-h
                    (cddr lst)
                    (cdr ls1) (cons (car ls1) ls2))]))])
      (split-h lst lst '()))))
```

Figura 3.1 Função auxiliar para o algoritmo de merge sort.

```
(define merge-string
  (lambda (ls1 ls2)
    (cond
      [(null? ls1) ls2]
      [(null? ls2) ls1]
      [(string<=? (car ls1) (car ls2))
       (cons (car ls1) (merge-string (cdr ls1) ls2))]
      [else
       (cons (car ls2) (merge-string ls1 (cdr ls2)))])))


(define merge-sort-string
  (lambda (lst)
    (cond
      [(null? lst) lst]
      [(null? (cdr lst)) lst]
      [else (let ([splits (split lst)])
              (merge-string
               (merge-sort-string (car splits))
               (merge-sort-string (cdr splits))))])))
```

Figura 3.2 Ordena uma lista de palavras usando o algoritmo merge sort.

```
(define merge-number
  (lambda (ls1 ls2)
    (cond
      [(null? ls1) ls2]
      [(null? ls2) ls1]
      [(<= (car ls1) (car ls2))
       (cons (car ls1) (merge-number (cdr ls1) ls2))]
      [else
       (cons (car ls2) (merge-number ls1 (cdr ls2)))])))

(define merge-sort-number
  (lambda (lst)
    (cond
      [(null? lst) lst]
      [(null? (cdr lst)) lst]
      [else (let ([splits (split lst)])
              (merge-number
               (merge-sort-number (car splits))
               (merge-sort-number (cdr splits))))])))
```

Figura 3.3 - Ordena uma lista de números inteiros usando o algoritmo merge sort.

Esses procedimentos claramente realizam o mesmo trabalho. A diferença é que para uma lista de números inteiros, ele usa um a função `<=` como operador de antecessor e para lista cadeia de caracteres, a função `string<=?`. Ou seja, podemos dizer que o algoritmo de ordenação é o mesmo, porém a implementação deve ser diferente para satisfazer à linguagem.

Em *Scheme* é possível construir esse algoritmo como ele é descrito. Por ser uma linguagem funcional com funções de ordem superior, é possível passar como parâmetro de uma função uma outra função.

Dessa forma, basta informar à função de ordenação qual operador utilizar como operador de comparação, passando o operador como parâmetro da função, como na figura 3.4.

Assim, aumentamos ainda mais o nível de abstração do programa, construindo uma função de ordenação por *merge sort* para qualquer lista homogênea.

```
(define merge
  (lambda (pred ls1 ls2)
    (cond
      [(null? ls1) ls2]
      [(null? ls2) ls1]
      [(pred (car ls1) (car ls2))
       (cons (car ls1) (merge pred (cdr ls1) ls2))]
      [else
       (cons (car ls2) (merge pred ls1 (cdr ls2)))])))


(define merge-sort
  (lambda (pred lst)
    (cond
      [(null? lst) lst]
      [(null? (cdr lst)) lst]
      [else (let ([splits (split lst)])
              (merge pred
                     (merge-sort pred (car splits))
                     (merge-sort pred (cdr splits))))])))
```

Figura 3.4 Ordena qualquer tipo de lista usando o algoritmo *merge sort*, apenas informando qual função utilizar como operador de predecessor.

Além dos aspectos inerentes ao paradigma funcional, a linguagem *Scheme* apresenta algumas características importantes. Uma delas é a facilidade de se implementar macros. Macros são transformações código-para-código definidas em tempo de desenvolvimento; são regras em nível de sintaxe da linguagem. Isto é, quando definimos um macro, estamos definindo uma “palavra reservada” do programa na linguagem.

Em *Scheme*, a implementação de tais regras é simples e funciona bem, independente do contexto em que executam. Elas são definidas e invocadas como qualquer outra função da linguagem, podendo conter blocos de código (CLINGER, 1991).

```
 (define-syntax discriminante
  (syntax-rules ()
    ((a b c)
     (let ((temp b))
       (- (* temp temp) (* 4 a c))))))
```

Figura 3.5 – Macro Discriminante

Além disso, algumas estruturas são definidas para execução de iterações, como laços de execução, que se aproximam do modelo imperativo. Apesar de essas construções serem parecidas com as das linguagens imperativas, sua execução mantém o comportamento recursivo previsto na linguagem para execução de repetição de código (FELLEISEN, 2003).

Em computadores baseados na arquitetura de von Neuman (seja uma arquitetura seqüencial ou paralela), isto pode acarretar numa perda significativa de eficiência, pois a recursividade nesse modelo é mais custosa em relação a laços de execução de código (SEBESTA, 2003).

4 Programação paralela

Muitos problemas físicos e matemáticos e modelos abstratos enfrentam sérios problemas computacionais, pois a tecnologia da computação seqüencial enfrenta limitações físicas. É amplamente aceito que o caminho mais imediato para o aumento de desempenho é através do uso de sistemas não restritos à computação seqüencial, sendo a computação paralela a principal aposta para esse fim. Esse é um argumento já um tanto antigo e um grande avanço já foi alcançado em relação a computadores com alto grau de paralelismo. (HUDAK, 1986)

Um dos mais simples modelos de arquitetura paralela consiste em vários processadores trabalhando em conjunto e com uma memória compartilhada ou distribuída e interconectada por uma via comum ou uma rede comunicando-se através de envio de mensagens. Esse tipo de arquitetura, além de minimizar os efeitos do gargalo de *von Neumann*, é extensível e geralmente fácil de construir (HUDAK, 1986).

Apesar dos avanços tecnológicos para a construção de microprocessadores paralelos, o desenvolvimento de linguagens e algoritmos para a programação de tais máquinas não tem evoluído com a mesma velocidade. Por isso, o ponto mais crítico no desenvolvimento de um sistema desses não é a modelagem do hardware, e sim o desenvolvimento do software. O futuro da computação paralela depende da criação de modelos simples e ao mesmo tempo poderosos de programação paralela que deixem transparente para o usuário os detalhes da arquitetura do sistema. Muitos pesquisadores afirmam que as linguagens imperativas convencionais são inadequadas para tais modelos, já que essas linguagens são intrinsecamente amarradas ao modelo de von Neuman de “uma operação por vez” (HUDAK, 1986).

Por razões históricas e comerciais, as linguagens dominantes na computação paralela são as linguagens imperativas, como C (utilizando-se extensões como **openmp**), C++ e HPF (High Performance FORTRAN). Como o paradigma imperativo prevê, essas linguagens utilizam-se de estruturas, definições ou comandos para realizar o paralelismo no programa. O HPF e algumas extensões do C, como o Cpar, por exemplo, utilizam as formas `forall` e `dopar` para laços a serem realizados em

paralelo. As informações do sistema no qual o programa será executado, como o número de processadores, por exemplo, ainda devem ser informadas ao programa.

Cada linguagem de programação contém técnicas específicas para uso do paralelismo. As técnicas utilizadas pelas linguagens imperativas são de controle de fluxo do programa. Assim, laços paralelos e blocos declarados explicitamente paralelos são os principais tipos de paralelismo encontrados nessas linguagens. Portanto, é preciso saber de antemão como o programa deve se comportar para se fazer sua programação (GEHANI; MCGETTRICK, 1988).

Para controle exato do fluxo de execução do programa faz-se uso de mecanismos de sincronismo (como filas), sistema de travamento em atribuições e leitura de variáveis compartilhadas (semáforos) e análise de dependência de dados em cada laço ou bloco paralelo. Isso torna a programação paralela de sistemas grandes e complexos ainda mais difícil, pois além de garantir que a lógica do programa está correta, o programador deve garantir que a ordem de execução de tal lógica está também correta.

Linguagens declarativas, ou funcionais, tentam eliminar alguns dos problemas que as linguagens imperativas apresentam. Essas linguagens evitam paralelismo explícito em favor do paralelismo implícito extraído pelo compilador, ou o sistema de execução, ou ainda o hardware.

Uma alternativa para a resolução desse problema é a utilização da programação funcional paralela (ou para-funcional), uma metodologia de programação paralela baseada no paradigma funcional de programação de computadores. O aspecto mais significante da metodologia é que o multiprocessador é tratado como um sistema autônomo único onde o programa é mapeado, e não um conjunto de processadores independentes que fazem comunicações complexas e requer sincronizações complexas. Com a computação para-funcional, um único modelo de execução e uma única linguagem de programação podem ser usados para resolução de problemas em um sistema uniprocessado e um sistema com múltiplos processadores em paralelo (HAMMOND, 1999).

4.1 Programação funcional paralela

O uso de linguagens imperativas para a programação de sistemas paralelos não é natural (HUDAK, 1986). De fato, o paradigma imperativo cria confusão no modo como pensamos para desenvolvermos programas.

Já em 1978, Backus discutia que o uso de técnicas de programação complexas, construções para invocar, controlar e coordenar a execução e a análise exata do que está ocorrendo no programa durante toda sua execução são fatores que dificultam muito a programação de tais sistemas. Faz mais sentido o uso de linguagens não seqüenciais para a programação de sistema paralelos. Um dos melhores candidatos para computação paralela é o paradigma de linguagens de programação funcionais.

A programação funcional paralela tem uma história relativamente longa. Desde 1975, sugere-se a execução em paralelo de argumentos das funções de ordem maior (funções que agem em outras funções). Desde então, várias implementações de compiladores e até mesmo máquinas de redução foram construídos (LOIDL, 2002).

O desenvolvimento de um programa paralelo é muito mais complexo que de um programa seqüencial, pois, além de se preocupar com a lógica do programa, é necessário pensar também em como coordenar a computação das partes paralelas do programa. Neste ponto, a programação funcional tem atributos que a deixa menos complexa que o tradicional paradigma imperativo (LOIDL, 2002). Por possuírem alto grau de abstração, é mais fácil atingir o paralelismo em vários níveis do programa (LOIDL, 2002). Além disso, as linguagens funcionais existentes são, em sua maioria, independentes de arquitetura (LOIDL, 2002), o que as tornam ainda mais atrativas para a programação paralela.

A semântica de um programa funcional puro define um único resultado para um conjunto de entradas. Essa característica é denominada de determinismo (HUDAK, 1986). Assim, programas funcionais são determinísticos no sentido de que qualquer programa que roda seqüencialmente irá produzir exatamente os mesmos resultados

quando rodar em paralelo, caso suas entradas sejam idênticas. Além disso, o programa irá terminar exatamente sob as mesmas condições (HAMMOND, 1999).

O determinismo de tais programas se dá muito pelo fato de linguagens funcionais puras não produzirem efeitos colaterais, tal como atribuição. Essa propriedade agrega grande valor para o desenvolvimento de sistemas computacionais paralelos, pois dessa forma o desenvolvimento e a depuração de um programa podem ocorrer em uma máquina seqüencial (onde a depuração é mais fácil) e esse mesmo programa pode ser posto para rodar em máquinas multiprocessadas para aumento de desempenho. Ainda, garante-se que é impossível um programa funcional puro paralelo entrar em *deadlock*, a não ser que o programa correspondente seqüencial não termine por causa de dependências cíclicas. (HAMMOND, 1999)

Porém, apesar de linguagens funcionais puras apresentarem esse alto grau de determinismo, elas representam as linguagens de programação funcionais menos populares. Na realidade, as linguagens funcionais mais usadas nas comunidades, tais como LISP e *Scheme*, fazem uso de estruturas de atribuições, que produzem o indesejado – para sistemas de computação paralela – e necessário – para sistemas de tempo real e controladores de dispositivos – efeito colateral. Afinal, apesar de as estruturas de atribuições serem uma herança do modelo de computador de *von Neuman* e não serem essenciais para um modelo abstrato de computação, uma linguagem sem nenhum tipo de atribuição é de certa forma um tanto radical (HAMMOND, 1999).

Isso não significa que o paralelismo seja difícil de ser atingido, pois programas funcionais geram muito pouco efeito colateral, afinal, o não-determinismo não está implícito na semântica da linguagem e, sim, explícito no programa (HUDAK, 1989). O ponto chave é que o paralelismo está implícito nas linguagens funcionais e sua semântica dá suporte a ele. Não há necessidade de construções de passagem de mensagens ou outro tipo de comunicação e não há necessidade de construções sintáticas ou diretivas para o paralelismo, tais como *parbegin... parend,* ou *forall*. (HAMMOND, 1999)

Existem três classes de linguagens funcionais (HAMMOND, 1999):

- **Linguagens estritas**: os argumentos de funções são avaliados antes da avaliação da função;
- **Linguagens não estritas**: os argumentos são avaliados somente se forem necessários (*lazy binding*);
- **Linguagens híbridas**: argumentos simples, como inteiros, são resolvidos antes da avaliação da função; no entanto, argumentos complexos, como listas, seguem a proposta das linguagens não estritas.

Nesse contexto, o paralelismo em linguagens funcionais pode ser explorado de forma explícita ou implícita.

A exploração do paralelismo implícito em linguagens estritas é bastante simples, pois todos os argumentos devem ser avaliados antes da chamada da função. Considerando-se que os argumentos podem ser expressões, surge a oportunidade para a exploração do paralelismo sempre que uma função for avaliada, ou seja, a avaliação de seus argumentos em paralelo antes da chamada da própria função.

Porém, nas linguagens não estritas o paralelismo implícito é obtido na avaliação paralela apenas dos argumentos necessários para a avaliação de uma função. Nesta classe de linguagem não há uma previsão da natureza dos argumentos antes de passá-los para as funções e isto dificulta a correta paralelização dos programas.

O *lazy binding* (ou *lazy evaluation*) (passagem de parâmetro por demanda) assegura que os argumentos das funções somente serão avaliados se forem necessários. Esta característica permite que se trabalhe com estruturas infinitas. A expressão lambda (GORDON, 1988).

$$w = ((x.xx)\ (x.xx))$$

não possui forma normal. Isto mostra que independentemente da ordem das reduções, a validação desta expressão não chegará ao fim. Já a expressão

$$w = (y.z)\ ((x.xx)\ (x.xx))$$

tem a forma normal z.

Este exemplo mostra que se fosse aplicada a ordem normal de redução (avaliação dos parâmetros antes da própria função), w seria reduzido antes da saída, o que amarra x a w, tornando o processo de validação infinito. O *lazy binding* permite que somente o necessário seja avaliado, possibilitando a redução de x à forma normal z. A exploração do paralelismo nessas avaliações tem como principal problema a consistência dos resultados, ou seja, a possibilidade de duas reduções aplicadas à mesma expressão encontrarem formas normais diferentes. Além disso, a ordem de redução é importante para se certificar que um programa termina (GORDON, 1988).

O *lazy binding* baseia-se na ordem normal para realizar a validação das expressões necessárias à computação do resultado. Ele avalia sempre a redução mais à esquerda e mais para fora, ou seja, avalia os argumentos somente quando necessário, evitando assim computações infinitas. Esta característica dificulta a redução de expressões em paralelo, isto porque a ordem normal de avaliações restringe a somente uma redução em cada intervalo de tempo. Esse tipo de avaliação é perfeitamente satisfatório para máquinas seqüenciais, no entanto é dificilmente utilizado para máquinas paralelas.

Além do paralelismo implícito, podemos criar estruturas numa linguagem funcional para atingirmos o paralelismo explicitamente, como, por exemplo, colocarmos anotações no código, tais como **seq** e **par**, que indicam se um trecho do programa será executado seqüencialmente ou em paralelo.

Duas formas de paralelismo explícito são comumente exploradas: paralelismo de estruturas de controle e paralelismo de dados. Para o paralelismo de dados, os programas funcionais apresentam uma construção chamada *map*, a qual executa uma função sobre uma lista de dados. A execução de tal expressão será feita em paralelo se o programa estiver sendo executado num sistema com arquitetura SIMD.

Para paralelizar estruturas de dados algumas técnicas são aplicadas, como a de anotações. Uma das mais utilizadas é a definição de construções que explicitam o

paralelismo, tais como em linguagens imperativas, através de criação de *threads*, laços paralelos explícitos, ou comandos (ou diretivas) de paralelização. Uma alternativa muito comum na programação funcional é o uso de *futures* (FLANAGAN, 1994*)*.

Uma *future* é uma expressão que "guarda o lugar" para o valor que será computado por um outro processo (uma função associada a essa *future*). Isto é, quando uma expressão `(future expr)` é executada, um novo processo é criado e a execução de expr é iniciada imediatamente nesse novo processo. A expressão `future` retorna imediatamente uma marca, para o processo pai, que continua a execução em paralelo ao processo criado. Quando o novo processo criado termina a execução, diz-se que a *future* foi resolvida. Assim, se um processo precisar saber o valor de uma expressão ainda não resolvida, ele é então bloqueado até que a expressão seja resolvida. Ainda, é garantido que a expressão `(future expr)` sempre retorna o mesmo valor.

A forma *future* pode ser vista como a paralelização de uma *lazy evaluation*, com a diferença que a primeira inicia o processamento de imediato. Assim, seu uso assegura que uma expressão pode ser executada imediatamente sem nenhuma alteração no resultado da computação e nenhum aspecto semântico do programa muda, a não ser pela introdução de concorrência (FLANAGAN, 1994).

Os compiladores para essas linguagens devem fazer importantes otimizações para aumentar a qualidade (velocidade e o uso de memória) do código gerado quando o programa é executado em arquiteturas multiprocessadas baseadas em máquinas baseadas na arquitetura de *von Neuman*. Por exemplo, para suportar uma simples regra de atribuição, um programa funcional geralmente faz uso de muito mais memória que um programa imperativo correspondente. Desde que o nome de um vetor não pode ser redefinido, um novo vetor deve ser alocado e definido, mesmo que a maioria dos valores não tenha sofrido mudanças. Esses compiladores podem fazer dois tipos de análise para otimizar a realocação dos valores do vetor, e experiências mostram que as análises são bastante efetivas (WOLFE, 1996):

- Análise *Build-in-place*: determina quando a memória para uma expressão pode ser pré-alocada (talvez estaticamente) e a computação convertida em atribuições para esses locais de memória. Em particular, em programas que concatenam dois vetores, essa otimização é importante: se eles puderem ser pré alocados para serem adjacentes, então a operação de concatenação não precisará ser feita.

- Análise *Update-in-place*: verifica se é possível, ao invés de alocar novo espaço de memória para um vetor, utilizar o espaço de memória de um vetor já instanciado e sem uso posterior. Se há potencialmente mais uso para o vetor, o compilador deve prolongar a atualização até que o uso do vetor antigo esteja completo.

5 Autômato finito adaptativo paralelo

A motivação principal deste trabalho é que a paralelização da execução de autômatos adaptativos é melhor na resolução de problemas que, se fosse restrita a computação seqüencial, tomaria mais tempo de processamento. (ROCHA; GARANHANI, 2006) Portanto, a proposta principal desse trabalho é construir e mostrar um *framework* de execução em paralelo do modelo de autômatos adaptativos e, posteriormente, fazer comparações de eficiência entre a execução do modelo em paralelo e do seqüencial.

O modelo dos autômatos de estados finitos pode ser dividido em duas classes: determinísticos e não-determinísticos. O autômato finito determinístico (ou autômato determinístico) é uma máquina de estados com entradas e saídas univocamente determinadas, isto é, dado seu estado atual e um símbolo de entrada, a máquina tem apenas um estado destino, um único caminho a seguir. Esta característica do autômato determinístico torna a implementação de tal modelo muito simples, pois não é preciso nenhum tipo “inteligência” do algoritmo para decidir qual o próximo estado do autômato. Os autômatos não-determinísticos, porém, exigem algumas tomadas de decisões por parte do algoritmo, pois, por serem um modelo onde o estado atual e o estímulo de entrada podem levar a mais de um estado, é preciso pensar em qual a estratégia usada para percorrer esses vários caminhos.

Uma estratégia muito usada para implementar tal tomada de decisão é utilizar *backtracking*, que consiste em memorizar as transições já passadas pelo autômato a fim de poder percorrer o caminho inverso no autômato e depois refazer o processo com uma nova transição. Assim, toda vez que o autômato encontra mais de um caminho a percorrer, o algoritmo escolhe apenas um dos caminhos e continua sua execução. Caso o autômato não reconheça a cadeia de entrada, ele retorna ao ponto de não-determinismo, fazendo o caminho de volta (*backtrack*), e toma um dos outros caminhos possíveis, até que todos os caminhos sejam tomados ou que a cadeia de entrada seja reconhecida.

O *backtracking* pode ser implementado de várias maneiras. A mais comumente utilizada é empilhar todas as transições durante sua execução e, ao chegar a um

estado de não aceitação, desempilhá-las para fazer o caminho de volta até o ponto de não-determinismo.

Podemos identificar dois grandes problemas neste método: 1) o mecanismo de *backtracking*, que pode ser implementado através do uso de uma pilha ou usando recursividade para explorar todos os caminhos possíveis, o que torna a execução pouco eficiente; 2) além disso, ele é lento, pois tem que percorrer trechos duplicados do autômato (ida e volta) – esse problema se torna ainda maior quando se implementa esse algoritmo para os autômatos adaptativos, pois, além de saber onde estão os pontos não-determinísticos, o algoritmo deve saber qual era o modelo vigente quando o não-determinismo foi encontrado.

Além disso, esse mecanismo existe apenas para executar um autômato não determinístico seqüencialmente. Na teoria, ao encontrar um caminho ambíguo, um autômato deve seguir os dois caminhos ao mesmo tempo, ou seja, executar as duas transições em paralelo. A proposta de paralelização do autômato finito adaptativo é justamente executar os vários caminhos não-determinísticos ao mesmo tempo. Assim, além de tornar o modelo teoricamente mais rápido, essa proposta se aproxima mais do modelo teórico de autômatos.

O algoritmo abaixo é a base da implementação de um autômato reconhecedor de linguagem. A diferença principal deste algoritmo para o seguido na execução de um autômato tradicional (implementado seqüencialmente e utilizando o *backtracking*) é justamente que ele trata as várias possíveis transições a serem seguidas quando em um determinado estado, mostrado nos passos 2 e 3.

1. O autômato inicia sua execução no seu estado inicial. Inicie o contador de passos sem leitura com o valor zero.
2. Se houver algum símbolo na cadeia de entrada, leia e retire o símbolo mais à esquerda. Se não houver nenhum símbolo na cadeia, vá para o passo 6.
3. Preencha uma lista com todas as transições que saem do estado corrente do dispositivo e respondem ao estímulo lido da cadeia ou a ε. Se todas as transições encontradas tiverem ε como símbolo, incremente o contador de

passos sem leitura. Senão, zere o contador de passos sem leitura. Se o valor do contador de passos sem leitura for maior o número máximo de passos sem leitura, rejeite a cadeia de entrada e vá para o passo 7.

4. Se a lista de transições estiver vazia, rejeite a cadeia de entrada e vá para o passo 7. Se a lista contiver n elementos $(n > 0)$, cria-se $n-1$ tarefas, onde cada uma fará uma cópia do dispositivo atual e da cadeia de entrada. A tarefa i criada seleciona a transição na posição i da fila. A tarefa corrente seleciona a transição na última posição da fila.
5. Se a transição selecionada tiver ação adaptativa anterior, aplique a ação adaptativa sobre o modelo, re-insira o símbolo na posição mais à esquerda da cadeia de entrada. Caso contrário, mude o estado do dispositivo de acordo com a transição selecionada. Se o símbolo da transição for ε, reinsira o símbolo lido na cadeia de entrada. Se a transição tiver uma função adaptativa posterior, aplique-a sobre o modelo. Vá para o passo 2
6. Se o estado atual do dispositivo for um estado de aceitação, aceite a cadeia de entrada e sinalize todas as outras tarefas para terminarem. Caso contrário, a cadeia é rejeitada.
7. Termine a execução do dispositivo, retornando a indicação a respeito da cadeia, se ela foi aceita ou rejeitada, terminando a tarefa corrente.

Nota-se neste algoritmo a inclusão de um dispositivo contador de passos sem leitura de cadeia. Esse dispositivo evita que haja repetição infinita na execução do dispositivo, com sucessivas transições em vazio.

O número máximo de transições em vazio pode ser um argumento da função ou uma definição no programa.

Ao paralelizar a execução dos autômatos adaptativos, espera-se que seu tempo de execução diminua drasticamente, pois sua execução passará a se comportar como a de um autômato determinístico, a não ser pela criação de tarefas.

5.1 Implementação

Para o desenvolvimento do modelo, optou-se por usar a linguagem *Scheme*, por ser uma linguagem funcional estrita que permite o *lazy binding* explícito (*futures*) e que possui recursos para execução de código em paralelo (*threads*). Além disso, por ser uma linguagem baseada no paradigma funcional, ela é propícia para a implementação de modelos matemáticos simbólicos, como autômatos.

Uma das preocupações durante implementação do modelo paralelo foi a facilidade de manutenção e modificação do modelo, pois se levou em conta que essa implementação poderá ser usada por todo um grupo de estudo e vários modelos de autômatos poderiam ser criados a partir desta implementação. Por este motivo, primeiramente foi desenvolvido a base do autômato adaptativo, o que inclui funções para criar o autômato, suas transições e suas funções adaptativas, além de mecanismos para execução da função adaptativa e inclusão, exclusão e pesquisa de transições e criar novos estados.

Também por facilidade de implementação e entendimento do código, as funções citadas acima, que alteram a estrutura do autômato adaptativo (inclusão e remoção de transições), foram implementadas usando-se macros.

Com essa base para criação e modificação do autômato pronta, foi, então, criada uma função para execução das transições do autômato adaptativo em paralelo. Essa função é a base de execução para o funcionamento de qualquer de autômato adaptativo, independente de sua natureza (reconhecedores ou transdutores). Assim, a diferença entre autômatos de natureza diferente será apenas o modo em como as informações de retorno dessa função serão tratadas (o que fazer com elas, critério de parada, etc.).

6 Framework de desenvolvimento de autômatos finitos adaptativos

Este anexo visa descrever as principais funções existentes no framework de desenvolvimento de autômatos finitos adaptativos.

Para tal, será mostrado, passo a passo, a criação de um dispositivo, definindo as regras que o programa deve seguir em seu desenvolvimento.

6.1 Estados, símbolos e transições

Os estados e os símbolos dos autômatos finitos adaptativos são representados de forma diferente no programa. Cada estado é representado por um número e cada símbolo, por uma cadeia de caracteres (tipo *Symbol* do *Scheme* – uma cadeia de caracteres precedido do caractere " ' " - *quote*).

Para melhorar a eficiência do programa, cada cadeia de caracteres que representa um símbolo de entrada é traduzida para um símbolo de um único caractere no programa. Desta forma, a rotina de busca de símbolos será mais eficiente, pois irá sempre fazer a comparação de cadeias de tamanho 1.

A definição das transições no programa é diferente da definição formal. Enquanto, formalmente, as transições são representadas uma lista de 5 elementos, no programa, elas são representadas por uma lista de 7 elementos. A diferença da definição formal para o programa consiste no fato de que os parâmetros passados para as funções adaptativas são definidos na transição, sendo representados por uma lista. As transições são definidas como:

- estado inicial da transição
- símbolo da transição
- estado final da transição
- chamada da função adaptativa anterior
- chamada da função adaptativa posterior
- lista com os parâmetros para a função adaptativa anterior
- lista com os parâmetros para a função adaptativa posterior

O nome da função adaptativa, assim como os símbolos da cadeia de entrada, é definido por uma cadeia de caracteres precedida de *quote*.

A função que cria uma transição é `make-automaton-transition`, cujos parâmetros são os sete elementos que a definem. Se uma transição não faz chamada para tem uma função adaptativa, os elementos da lista com o nome da função será preenchido com o símbolo `null-adapative-function` e a lista de parâmetros será uma lista vazia.

Além da criação de transições, também existem funções para busca, inserção e remoção delas no modelo, permitindo, desta maneira, a alteração da estrutura do autômato subjacente durante do modelo adaptativo. Para a busca de transições no autômato, existem 6 funções distintas: `search-transitions-match-f`, `search-transitions-match-fs`, `search-transitions-match-ft`, `search-transitions-match-s`, `search-transitions-match-st`, `search-transitions-match-t`. Nessas funções, o sufixo (*f – from, s – symbol, t – to*) indica quais elementos devem ser verificados. Por exemplo, a função `search-transition-ft` busca as transições em que os estados inicial e final são iguais aos seu dois parâmetros f e t, respectivamente. Dessa forma, essas funções irão receber número variado de parâmetros, sendo que em todos os casos, o primeiro parâmetro é sempre o dispositivo no qual a busca será realizada. Portanto, a função `search-transition-ft` irá receber 3 parâmetros: o dispositivo, o estado inicial e o estado final das transições. É importante notar que a busca das transições não leva em conta o nome das funções adaptativas, diferentemente do modelo descrito neste trabalho.

O retorno dessas funções é uma lista de transições, sendo que a constante `#f` marca o final da lista.

As funções de busca e remoção de transições do autômato finito adaptativo são `insert-transition!` e `remove-transition!`. Como no caso das funções de busca, essas funções recebem como primeiro parâmetro o dispositivo que sofrerá as alterações de inserção e remoção de transições. Além disso, elas recebem a transição

que deverá ser incluída ou excluída. A saída dessas funções é o dispositivo com as alterações sofridas.

A função de remoção de uma transição fará uma busca no dispositivo para saber qual elemento da lista de transições remover. Neste caso, o nome da função adaptativo é levado em conta para se fazer a busca.

6.2 Funções adaptativas

As funções adaptativas no programa são definidas por uma lista de dois elementos. O primeiro é o nome da função, um símbolo único que identifica a função. O segundo é um bloco de código, que representa o comportamento da função adaptativa e será executado quando a função adaptativa for invocada.

O bloco de código que representa o comportamento da função adaptativa é uma função lambda que deve receber sempre dois parâmetros: o dispositivo sobre o qual ele é executado e a lista de parâmetros da função adaptativa.

Por ser uma função lambda, é possível fazer qualquer coisa no comportamento da função adaptativa, porém, recomenda-se que apenas as ações primitivas das funções adaptativas sejam executadas nelas.

As funções `search-transition-*`, `insert-transition!` e `remove-transition!` representam as ações elementares de busca, inserção e remoção de transições. Além delas, existe a função `generate-state!`, que representa a ação elementar de criação de estados. Ela recebe como parâmetro o dispositivo e retorna o estado criado nele.

`make-adaptive-function` é a função que cria as funções adaptativas do dispositivo. Ela recebe sempre dois parâmetros: o nome da função e seu bloco de código (expressão lambda).

A execução de uma função adaptativa é feita através da função `exec-adaptive-function`, que será invocada toda vez que uma transição tenha uma função

adaptativa não nula. Essa função irá invocar o bloco de código através do nome da função adaptativa, passando sempre dois parâmetros: o autômato adaptativo e a lista de parâmetros da função adaptativa, definida na transição. Por este motivo, o bloco de código da função adaptativa deverá sempre ser uma expressão lambda com dois parâmetros.

6.3 Definição do autômato

Uma das preocupações no desenvolvimento do framework foi manter o programa o mais próximo das definições formais do autômato. Porém, por facilidade de implementação e entendimento, algumas definições foram simplificadas.

Como já mostrado, a representação formal desse modelo se dá por $AFA = (Q, \Sigma, q_0, F, T, Q_\infty, \Gamma)$. Já no framework, este dispositivo é definido por uma lista de 8 elementos, sendo:

- `'adaptive-automaton` – uma constante que identifica a lista como sendo a definição de um autômato adaptativo;
- C - a lista inicial de estados;
- Σ - a lista de símbolos;
- TA - a lista inicial de transições;
- c_0 - estado inicial;
- F - a lista de estados finais;
- AF - a lista de funções adaptativos; e
- Dicionário para a lista de estados – uma lista de pares ordenados que traduzem o nome dos estados para um único caractere (uso interno do framework).

Assim, para a criação de um autômato, é preciso defini-lo através desta lista e, para tal, foi criada a `make-finite-state-automaton`, função a qual recebe 6 parâmetros, que vão definir os itens 2 a 7 dessa lista, e retorna o dispositivo criado.

6.4 Fluxo de execução

Com a finalidade de tornar o desenvolvimento de novos modelos de autômatos uma tarefa simples, pensou-se em criar um *framework* de execução do autômato, que serviria de base para a criação de qualquer tipo de autômato (reconhecedor de linguagem ou transdutor). Com isso em mente, foi desenvolvida a função `execute-transition`, que nada mais é que o dispositivo de transição do autômato de um estado para outro. Ela é encarregada de verificar se há funções adaptativas e executá-las. Além disso, é nesta função que será feita a paralelização da execução das transições.

A função `execute-transition`, recebe 4 argumentos: a próxima transição a ser executada, o dispositivo, a cadeia de entrada e um *flag* de paralelismo. Essa função irá fazer a transição, executando as funções adaptativas que houver, alterando, assim, o autômato recebido por parâmetro. Se o *flag* de paralelismo estiver ligado, toda essa execução acontecerá numa *thread* nova. O retorno dessa função deve ser o autômato após a transição concluída (alterado por uma função adaptativa ou não).

A função que define o comportamento do dispositivo é `automaton-move`. Nesta função são definidos os critérios de parada, a forma de execução das transições (em paralelo ou seqüencialmente), a leitura da cadeia de entrada, etc. Ela recebe três parâmetros: o dispositivo adaptativo, sua cadeia de entrada e o estado corrente do autômato subjacente.

As funções `automaton-move` e `execute-transition` são recursivas entre si, isto é, uma faz chamada para outra. Essas funções foram feitas dessa maneira para simplificar a diferenciação entre a execução seqüencial e a paralela no *framework*. Dessa maneira, a única diferença entre seqüencial e o paralelo é a sua função principal `automaton-move`.

6.5 Exemplo

A seguir é representado o autômato finito adaptativo utilizado para resolver o problema do caixeiro viajante.

O caixeiro viajante é um problema clássico na computação e na matemática. Neste problema, um viajante deve visitar um número x de cidades, passando por cada uma delas apenas uma vez, e retornar a cidade de onde partiu, fazendo o percurso menos custoso, ou seja, percorrer a menor distância possível. Sua importância se dá pelo fato de ser um problema de resolução muito complexa, com alto custo computacional (SCHRIJVER, 1991).

Muitos trabalhos apresentam diferentes maneiras de resolver este problema, utilizando algoritmos de solução exata ou usando algum tipo de heurística para uma solução aproximada, mas de menor custo computacional. Neste trabalho, optou-se por um autômato finito adaptativo na busca de solução exata. Desta maneira, será mostrada uma versão do autômato que percorrerá todas as possíveis rotas do viajante e calculará a menor distância a ser percorrida.

Dado um conjunto de cidade $C = \{c_1, c_2, c_3, c_4\}$, com matriz de distâncias

$$M_C = \begin{bmatrix} 0 & 3 & 4 & 4 \\ 3 & 0 & 6 & 2 \\ 4 & 6 & 0 & 8 \\ 4 & 2 & 8 & 0 \end{bmatrix},$$

constrói-se o autômato finito adaptativo tal que cada estado do autômato representa uma cidade e as transições representam o caminho a ser percorrido pelo caixeiro viajante. Desta maneira, define-se, também, que os símbolos de entrada do autômato são as distâncias entre cada cidade. Assim, cada transição executada pelo autômato representa a passagem do viajante de uma cidade para outra. O autômato finito adaptativo construído para resolver tal problema acima é representado na figura 6.1.

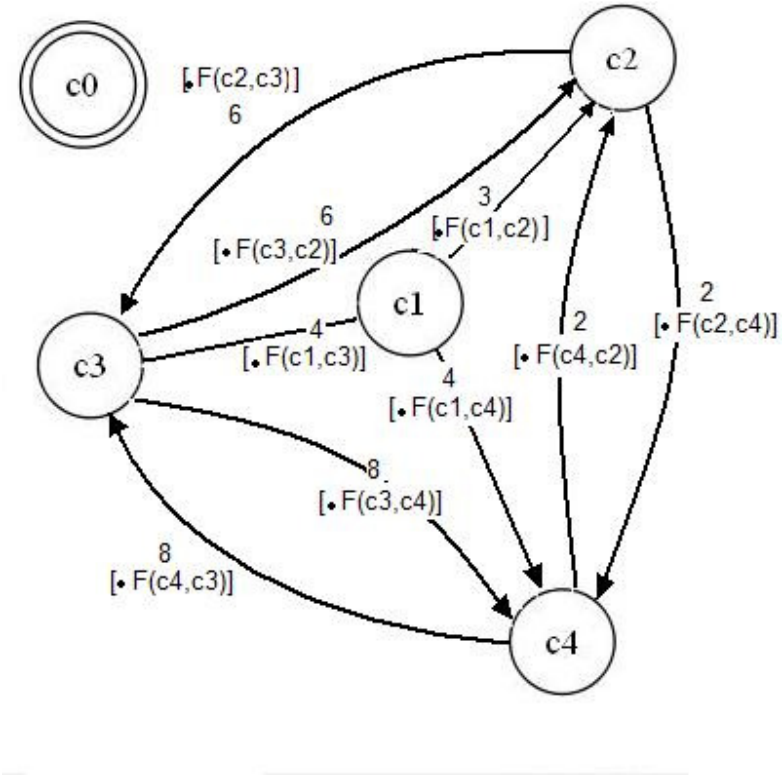

Figura 6.1 – autômato que resolve o problema do caixeiro viajante, em sua configuração inicial

Este autômato irá reconhecer toda cadeia de entrada que se inicia em c_1, passa por todas as cidades apenas uma vez, e termina em c_0. A transição dos estados c_2, c_3 e c_4 para c_0 simula o retorno a c_1. Para isso, definimos as funções adaptativas posteriores $corta\_caminhos(\ )$, $marca\_cidade\_atingida(\ )$ e $cria\_caminho\_de\_volta(\ )$, as quais farão com que o caixeiro não passe pela mesma cidade mais de uma vez ($corta\_caminhos(\ )$) e não o deixará retornar a cidade de origem sem antes passar por todas as cidades ($marca\_cidade\_atingida(\ )$ e $cria\_caminho\_de\_volta(\ )$). Essas funções são representadas na figura 6.2.

Assim, para encontrar o menor caminho a ser percorrido pelo viajante, basta abastecer o modelo com todas as possíveis combinações de distâncias na cadeia de entrada do autômato. Toda vez que o dispositivo reconhecer uma cadeia de entrada, guarda-se o modelo e a soma das distâncias percorridas (soma dos símbolos da cadeia de entrada). O modelo que tiver a menor das somas será a resposta para o problema.

A seguir, será mostrado como implementar o autômato finito adaptativo que resolve o problema descrito acima.

```
corta_caminhos() : {
     ; remove todos os caminhos que chegam a %2,
     ; a não ser os que saem de c1 e %1 (corrente)
     ? [ ε  1 ?p %2 f( %1 %2 ) ]
     ? [ ε %1 ?q %2 f( %1 %2 ) ]
     ? [ ε ?x ?y %2 f( %1 %2 ) ]
     - [ ε ?x ?y %2 f( %1 %2 ) ]
     + [ ε  1 ?p %2 f( %1 %2 ) ]
     + [ ε %1 ?q %2 f( %1 %2 ) ]
}

marca_cidade_atingida() : {
     + [ ε 0 m %1 ε ]
}

cria_caminho_de_volta() : {
     ; cria o caminho do estado %1 para 0, se todas as
     ; cidades já foram alcançadas
     ? [ ε ?x  m  2 ε ]
     ? [ ε ?x  m  3 ε ]
     ? [ ε ?x  m  4 ε ]
     ? [ ε  1 ?y %1 corta_caminhos( %1 ) ]
     + [ ε %1 ?y ?x ε ]
}

f() : {
     ; Parâmetros: estados inicial e final da transição
     corta_caminhos( %1 %2 )
     marca_cidade_atingida( %2 )
     cria_caminho_de_volta ( %2 )
}
```

Figura 6.2 – Definição das funções adaptativas posteriores

6.5.1 Implementação

O framework para desenvolvimento e execução dos autômatos adaptativos, resultado deste trabalho, define o autômato finito adaptativo em duas fases: 1) a criação do autômato, definindo cada uma de suas partes separadamente, e 2) a lógica de

execução, onde é necessário definir também se sua computação se dará seqüencialmente ou em paralelo.

6.5.2 Criação

A criação de um autômato finito adaptativo é feita por passos. Primeiramente, é necessário definir o comportamento das funções adaptativas $corta\_caminhos(\ )$, $cria\_caminhos\_de\_volta(\ )$, $marca\_cidade\_atingida(\ )$ e $f(\ )$. Para isso, implementa-se uma função (expressão lambda) para cada uma das funções adaptativas. As figuras 6.3, 6.4, 6.5 e 6.6 mostra como cada uma dessas funções é definida no *Scheme*.

Como mostrado no capítulo anterior, essas funções são definidas para receberem dois parâmetros: 1) o autômato finito adaptativo que sofrerá as alterações da ação adaptativa e 2) a lista de parâmetros da função adaptativa. Além disso, diferentemente da notação formal, as variáveis destas funções são definidas explicitamente.

É importante notar nessas funções as diferenças delas para a notação formal. No fundo, essas funções transcrevem o comportamento das funções adaptativas, ao invés de transcreverem as próprias. Neste exemplo, as funções `cut-paths`, `create-return-path` e `mark-reached-city` transcrevem o comportamento das funções adaptativas $corta\_caminhos(\ )$, $cria\_caminhos\_de\_volta(\ )$ e $marca\_cidade\_atingida(\ )$ respectivamente.

O melhor exemplo dessa diferença entre as funções adaptativas e as funções no *Scheme* é a função `cut-paths`. Nesta função, o laço de remoção de transições é uma função recursiva que remove as transições uma a uma. Além disso, nessa função são removidas todas as transições que chegam ao estado final da transição, a não ser a transição corrente e a transição que parte de 1 pare este estado. Já na função adaptativa, são removidas todas as transições que chegam a este estado e depois adicionado de volta a transição corrente e a transição de c_1 para este.

Uma vez definidos esses comportamentos, é preciso defini as funções adaptativas de tal forma que o autômato finito adaptativo poderá executá-los. Para tal, utilizamos a função `make-adaptive-function`, que recebe como parâmetros o nome da função adaptativa e a função *Scheme* que define seu comportamento. Dessa maneira, é possível executar uma ação adaptativa através da função `exec-adaptive-function`, que recebe o nome da função adaptativa e executa a função ligada a ela.

A função `make-automaton-transition` é a função que cria as transições do autômato finito adaptativo. Ela recebe como parâmetros os componentes que definem a transição, na seguinte ordem: estado inicial, símbolo, estado final, chamada da função adaptativa anterior, chamada da função adaptativa posterior, parâmetros a ser passado para a função adaptativa anterior, parâmetros para a função adaptativa posterior.

Uma vez criadas essas estruturas, podemos definir o autômato finito adaptativo utilizando a função `make-automaton-transition`. A criação do autômato que

```
;; cut-paths - removes all paths to city2 but the one just taken (from city1)
(define cut-paths
  (lambda (sm param-list)
    (let* ((current-city (first  param-list))
           (next-city    (second param-list))
           (return-paths (remove
                          (car (search-transitions-match-ft sm 1 next-city))
                          (remove
                           (car (search-transitions-match-ft sm current-city next-city))
                           (search-transitions-match-t sm next-city)))))
;     laço de remoção
      (letrec
          ((remove-paths
            (lambda (paths)
              (if (car paths)
                  (let* ((return-path (car paths)))
                    (remove-transition! sm return-path)
                    (remove-paths (cdr paths)))))))
        (remove-paths return-paths)))
    sm))

(define mark-reached-city
  (lambda (sm param-list)
    (let ((city (first param-list)))
      (insert-transition! sm (make-automaton-transition
                              0 'm city
                              null-adaptive-function
                              null-adaptive-function
                              '()
                              '())))
    sm))
```

Figura 6.3 – Definição do comportamento da função adaptativa $corta\_caminhos(\ \ )$

```
(define mark-reached-city
  (lambda (sm param-list)
    (let ((city (first param-list)))
      (insert-transition! sm (make-automaton-transition
                              0 'm city
                              null-adaptive-function
                              null-adaptive-function
                              '()
                              '())))
    sm))
```

Figura 6.4 – Definição do comportamento da função adaptativa $marca\_cidade\_atingida(\ \ )$

resolve o problema do caixeiro viajante proposto aqui, e de suas estruturas, é mostrado na figura 6.7.

```
(define create-return-path
  (lambda (sm param-list)
    (let ((city (first param-list))
          (c2-mark (search-transitions-match-ft sm 0 2))
          (c3-mark (search-transitions-match-ft sm 0 3))
          (c4-mark (search-transitions-match-ft sm 0 4)))
      (if (and (car c2-mark) (and (car c3-mark) (car c4-mark)))
          (let ((transition-from-c1 (car (search-transitions-match-ft sm 1 city)))
                (transitions-from-0 (search-transitions-match-f sm 0)))
            (letrec ((remove-marks
                      (lambda (tl)
                        (if (car tl)
                            (begin
                              (remove-transition! sm (car tl))
                              (remove-marks (cdr tl)))))))
              (remove-marks transitions-from-0))
            (insert-transition! sm (make-automaton-transition
                                    city
                                    (translated-symbol
                                     (symbol-of-transition transition-from-c1)
                                     (symbol-map-of-automaton sm))
                                    0
                                    null-adaptive-function
                                    null-adaptive-function
                                    '()
                                    '()))))))
    sm))
```

Figura 6.5 – Definição do comportamento da função adaptativa $cria\_caminho\_de\_volta(\ \ )$

```
(define adaptive-function
  (lambda (sm param-list)
    (let ((city1 (first param-list))
          (city2 (second param-list)))
      (set! sm ((exec-adaptive-function sm 'corta-caminhos) sm (list city1 city2)))
      (set! sm ((exec-adaptive-function sm 'marca-cidade-atingida) sm (list city2)))
      (set! sm ((exec-adaptive-function sm 'cria-caminho-de-volta) sm (list city2))))
    sm))
```

Figura 6.6 – Definição do comportamento da função adaptativa $f(\ \ )$

```
(let
    ((trans-list
      (list
       (make-automaton-transition 1 (as-symbol 3 ) 2
                                  null-adaptive-function 'f () '(1 2))
       (make-automaton-transition 1 (as-symbol 4 ) 3
                                  null-adaptive-function 'f () '(1 3))
       (make-automaton-transition 1 (as-symbol 4 ) 4
                                  null-adaptive-function 'f () '(1 4))
       (make-automaton-transition 2 (as-symbol 6 ) 3
                                  null-adaptive-function 'f () '(2 3))
       (make-automaton-transition 3 (as-symbol 6 ) 2
                                  null-adaptive-function 'f () '(3 2))
       (make-automaton-transition 2 (as-symbol 2 ) 4
                                  null-adaptive-function 'f () '(2 4))
       (make-automaton-transition 4 (as-symbol 2 ) 2
                                  null-adaptive-function 'f () '(4 2))
       (make-automaton-transition 3 (as-symbol 8 ) 4
                                  null-adaptive-function 'f () '(3 4))
       (make-automaton-transition 4 (as-symbol 8 ) 3
                                  null-adaptive-function 'f () '(4 3)))))
  (let ((aa (make-adaptive-automaton '(0 1 2 3 4)
                                     (map as-symbol '(2 3 4 6 8 m))
                                     trans-list
                                     1
                                     '(0)
                                     (list (make-adaptive-function
                                            'corta-caminhos cut-paths)
                                           (make-adaptive-function
                                            'marca-cidade-atingida mark-reached-city)
                                           (make-adaptive-function
                                            'cria-caminho-de-volta create-return-path)
                                           (make-adaptive-function
                                            'f adaptive-function)))))
    (automaton-move aa '(0 0) (start-state-of-finite-state-automaton aa))))
```

Figura 6.7 – Criação do autômato finito adaptativo que resolve o problema do caixeiro viajante proposto

6.5.3 Definindo a execução

Para execução do autômato, é preciso implementar a função de execução e aceitação deste: `automaton-move`. Esta função é responsável por disparar todo o mecanismo de execução do autômato: ler a cadeia de entrada, fazer as transições e definir as condições de parada do autômato. Nela, também, defini-se se o autômato irá executar em paralelo ou seqüencialmente.

A interface da função `automaton-move` é definida por 3 parâmetros: o autômato adaptativo, a cadeia de entrada e o estado atual do autômato, sendo que, quando chamada pela primeira vez, deve receber o estado inicial do autômato.

Para o caso do autômato finito adaptativo que resolve o problema proposto, iremos definir a função `automaton-move` de forma diferente da habitual. Neste exemplo, esta função não irá ler uma cadeia de entrada, mas sim disparar todas as transições a partir do estado corrente. Além disso, os critérios de parada não se basearão na cadeia de entrada do autômato, mas sim em sua configuração atual: caso o estado corrente do autômato seja seu estado de aceitação e todos os outros estados já tenham sido atingidos, o autômato pára sua execução e calcula a distância percorrida pelo caixeiro viajante. Dessa forma, todos os caminhos possíveis que o caixeiro viajante poderá percorrer serão cobertos.

Duas versões deste método foram implementadas para este exemplo: uma delas executando todas as transições a partir do estado corrente em paralelo, outra executando o autômato seqüencialmente, utilizando o mecanismo de *backtracking*. Essas funções são mostradas nas figuras 6.8 (versão em paralelo) e 6.9 (versão seqüencial).

```
(define (find-minimum-path-par automaton symbol-list cur-state)
  (let* ((distance (as-number (translated-symbol (first symbol-list) (symbol-map-of-automaton automaton))))
         (total-cost (+ distance (as-number (second symbol-list))))
         (transition-list (search-transitions-match-f automaton cur-state)))
    (if (<= (length transition-list) 2)
        (begin
          (if (and (belong? cur-state (final-states-of-finite-state-automaton automaton))
                   (reached-all-cities automaton))
              (begin
                (cons (list automaton total-cost) all-paths)
                (set! timings-end (current-milliseconds))))
          (stop-thread (current-thread)))
        (letrec ((step
                  (lambda (trans)
                    (let ((current-trans (car trans)))
                      (if current-trans
                          (let ((syml (list #f (symbol-of-transition current-trans) total-cost)))
                            (if (second current-trans)
                                (begin
                                  (execute-transition current-trans automaton syml #t)
                                  (step (cdr trans)))
                                (execute-transition current-trans automaton syml #f))))))))
          (if (car transition-list)
              (step transition-list)
              (stop-thread (current-thread)))))))
```

Figura 6.8 – Função de execução em paralelo do autômato finito adaptativo

```
(define (find-minimum-path-seq automaton symbol-list cur-state)
  (let* ((distance (as-number (translated-symbol (first symbol-list) (symbol-map-of-automaton automaton))))
         (total-cost (+ distance (as-number (second symbol-list))))
         (transition-list (search-transitions-match-f automaton cur-state)))
    (if (belong? cur-state (final-states-of-finite-state-automaton automaton))
        (verify-minimum-path automaton total-cost))
    (letrec ((step
              (lambda (trans)
                (let ((current-trans (car trans)))
                  (if current-trans
                      (let ((syml (list #f (symbol-of-transition current-trans) total-cost)))
                        (execute-transition current-trans automaton syml #f)
                        (step (cdr trans))))))))
      (if (car transition-list)
          (step transition-list)))))

(define (automaton-move automaton symbol-list cur-state)
  (find-minimum-path-seq automaton symbol-list cur-state))
```

Figura 6.9 – Função de execução seqüencial do autômato finito adaptativo e definição da função `automaton-move`

6.5.4 Execução e análise

Nesta seção, será mostrado como o autômato finito adaptativo se comporta durante a execução da função de busca da solução do problema do caixeiro viajante apresentada na figura 6.9.

Decidiu-se por mostrar a solução utilizando o algoritmo seqüencial por questões e entendimento, uma vez que a execução do algoritmo paralelo não apresenta uma seqüência natural desta resolução, trilhando vários caminhos de uma só vez. Além disso, decidiu-se por mostrar um dos possíveis caminhos completos que o caixeiro viajante pode percorrer, saindo da c_1 (estado 1 do autômato finito adaptativo), passando por todas as outras 3 cidades, e chegando à cidade destino c_1 (estado 0 – que simula a volta ao estado 1).

O autômato finito adaptativo da figura 1 é representado, no programa, como a figura 7. Essa figura mostra o autômato, representado por uma lista de 7 elementos – sendo eles: 1) o identificador do autômato, 2) a lista de estados, 3) a lista símbolos de entrada, 4) a lista de transições, 5) o estado inicial do autômato, 6) a lista de estados de aceitação e 7) as funções adaptativas – e o estado em que se encontra o autômato.

```
automato:
      'adaptive-finite-state-automaton
      (0 1 2 3 4)
      ('2 '3 '4 '6 '8 'm)
      #((1 '3 2 null-adaptive-function f () (1 2))
        (1 '4 3 null-adaptive-function f () (1 3))
        (1 '4 4 null-adaptive-function f () (1 4))
        (2 '6 3 null-adaptive-function f () (2 3))
        (3 '6 2 null-adaptive-function f () (3 2))
        (2 '2 4 null-adaptive-function f () (2 4))
        (4 '2 2 null-adaptive-function f () (4 2))
        (3 '8 4 null-adaptive-function f () (3 4))
        (4 '8 3 null-adaptive-function f () (4 3)))
      1
      (0)
      ((corta-caminhos #<procedure:cut-paths>)
       (marca-cidade-atingida #<procedure:mark-reached-city>)
       (cria-caminho-de-volta #<procedure:create-return-path>)
       (f #<procedure:adaptive-function>))
estado corrente:
      1
```

Figura 6.10 – Representação do autômato finito adaptativo que resolve o problema do caixeiro em sua configuração inicial

Isto é, o autômato, definido pela lista de 8 elementos, contém os estados 0, 1, 2, 3 e 4, uma lista de símbolos composta pelas distâncias entre as cidades e a marcação *m*, a lista de transições, estado inicial 1, o conjunto de estados de aceitação {0} e o conjunto de funções adaptativas.

Diferentemente do modelo formal, este autômato finito adaptativo não fará leitura de uma cadeia de entrada. Isto porque o modelo implementado visa percorrer todos os possíveis caminhos para chegar ao seu destino. Desta forma, o comportamento do autômato será de percorrer todas as transições a partir do estado corrente. Além disso, o critério de parada do autômato também é diferente do formal. Ele irá se basear apenas no conjunto de estados de aceitação para decidir se deve para a execução.

Assim, a execução da função `automaton-move` irá procurar todas as transições que partem de 1, colocá-las numa lista, ordenando por estado final da transição, e executará a primeira delas. Portanto, ao iniciar a execução, a primeira transição a ser

executada será $(1,'4) \rightarrow 2, f(1,2)$, e a configuração resultante dessa transição é o autômato representado na figura 11.

```
automato:
      'adaptive-finite-state-automaton
      (0 1 2 3 4)
      ('2 '3 '4 '6 '8 'm)
      #((1 '3 2 null-adaptive-function f () (1 2))
        (1 '4 3 null-adaptive-function f () (1 3))
        (1 '4 4 null-adaptive-function f () (1 4))
        (2 '6 3 null-adaptive-function f () (2 3))
        (2 '2 4 null-adaptive-function f () (2 4))
        (3 '8 4 null-adaptive-function f () (3 4))
        (4 '8 3 null-adaptive-function f () (4 3))
        (0 'm 2 null-adaptive-function null-adaptive-function () ()))
      1
      (0)
      ((corta-caminhos #<procedure:cut-paths>)
       (marca-cidade-atingida #<procedure:mark-reached-city>)
       (cria-caminho-de-volta #<procedure:create-return-path>)
       (f #<procedure:adaptive-function>))
estado corrente:
      2
```

Figura 6.11 – Autômato finito adaptativo após execução da transição $(1,'4) \rightarrow 2, f(1,2)$

Consecutivamente, o autômato irá alterar sua configuração, executando as transições e as funções adaptativas, até atingir uma configuração de aceitação do autômato. As próximas configurações do autômato em questão são mostradas na figura 12.

```
automato:
      'adaptive-finite-state-automaton
      (0 1 2 3 4)
      ('2 '3 '4 '6 '8 'm)
      #((1 '3 2 null-adaptive-function f () (1 2))
        (1 '4 3 null-adaptive-function f () (1 3))
        (1 '4 4 null-adaptive-function f () (1 4))
        (2 '6 3 null-adaptive-function f () (2 3))
        (2 '2 4 null-adaptive-function f () (2 4))
        (3 '8 4 null-adaptive-function f () (3 4))
        (0 'm 2 null-adaptive-function null-adaptive-function () ())
        (0 'm 3 null-adaptive-function null-adaptive-function () ()))
      1
      (0)
      ((corta-caminhos #<procedure:cut-paths>)
       (marca-cidade-atingida #<procedure:mark-reached-city>)
       (cria-caminho-de-volta #<procedure:create-return-path>)
       (f #<procedure:adaptive-function>))
estado corrente:
      4

automato:
      'adaptive-finite-state-automaton
      (0 1 2 3 4)
      ('2 '3 '4 '6 '8 'm)
      #((1 '3 2 null-adaptive-function f () (1 2))
        (1 '4 3 null-adaptive-function f () (1 3))
        (1 '4 4 null-adaptive-function f () (1 4))
        (2 '6 3 null-adaptive-function f () (2 3))
        (3 '8 4 null-adaptive-function f () (3 4))
        (0 'm 2 null-adaptive-function null-adaptive-function () ())
        (0 'm 3 null-adaptive-function null-adaptive-function () ())
        (0 'm 4 null-adaptive-function null-adaptive-function () ())
        (4 '4 0 null-adaptive-function null-adaptive-function () ()))
      1
      (0)
      ((corta-caminhos #<procedure:cut-paths>)
       (marca-cidade-atingida #<procedure:mark-reached-city>)
       (cria-caminho-de-volta #<procedure:create-return-path>)
       (f #<procedure:adaptive-function>))
estado corrente:
      0
```

Figura 6.12 – Configurações atingidas pelo autômato finito adaptativo

7 Considerações finais

Os primeiros experimentos foram desenvolvidos para avaliar sem provar formalmente a corretude do programa. Para tal, foram criados alguns autômatos finitos adaptativos reconhecedores: autômatos com funções adaptativas anteriores, com funções posteriores, autômatos com funções anteriores e posteriores e autômatos sem funções adaptativas. Ainda, para cada autômato, escolheram-se dois tipos de cadeias de entrada diferentes: uma que o autômato deveria aceitar, outra que deveria rejeitar. Além disso, cada teste foi executado numa máquina seqüencial e em uma máquina paralela. Com isso, tentou-se cobrir todos os casos de teste necessários para garantir que o programa funciona corretamente.

Os experimentos com o algoritmo foram realizados em uma máquina seqüencial e outra paralela. Procurou-se avaliar o desempenho do algoritmo, comparando seu tempo de execução ao de um algoritmo seqüencial que implementa *backtracking* usando recursividade. Assim, foi desenvolvido, também, um modelo seqüencial dos autômatos adaptativos em *Scheme*.

Além da comparação básica com o modelo seqüencial de autômatos adaptativos, foi implementada uma solução para o problema do caixeiro viajante a fim de comparação dos resultados e desempenho de diferentes algoritmos com os autômatos adaptativos paralelos. Para comparação, foram desenvolvidas 3 diferentes técnicas para solução do problema: 1) usando força bruta com autômatos adaptativos paralelos; 2) usando força bruta com autômatos adaptativos seriais (modelo clássico); e 3) usando autômatos genéticos. Cada técnica utilizada foi executada em duas diferentes máquinas: um sistema com Athlon X2 64bits, com 1 Gb de memória e com sistema operacional Windows XP, e um cluster de 8 processadores Intel com sistema operacional Linux Ubuntu.

Por ser um problema de difícil resolução, usou-se o problema do caixeiro viajante como teste de desempenho dos autômatos adaptativos, no qual, partindo de uma cidade *x*, o caixeiro deve visitar todas as *n* cidades e voltar para a cidade *x* percorrendo a menor distância possível.

Além de ser um problema com solução de alta complexidade (complexidade exponencial), sua representação gráfica é compreensível. Por estes motivos, o problema foi escolhido como forma de mostrar o funcionamento e viabilidade do uso do *framework* desenvolvido neste trabalho. Além disso, propões-se comparar os resultados deste trabalho e de algoritmos já existentes a fim de comparação de eficiência.

Como parâmetro de comparação de eficiência, levou-se em conta o tempo médio de execução dos programas, que foram executados com 4 diferentes configurações: 5 cidades, 7 cidades e 9 cidades. Os resultados são apresentados na tabela 1.

Algoritmo	***Tempo de execução 5 cidades***	***Tempo de execução 7 cidades***	***Tempo de execução 9 cidades***
Força bruta – seqüencial	0,072 seg	5,291 seg	408,718 seg
Força bruta – paralelo	0,030 seg	3,084 seg	*ERRO* - out of memory

Tabela 7.1 - Comparação do tempo de execução do algoritmo do caixeiro viajante

A execução de uma linguagem funcional nas máquinas mais usualmente utilizadas apresenta problemas de desempenho e consumo de recursos, pois aqueles sistemas são baseados na arquitetura de von Neuman. Por este motivo, não foi possível rodar experimentos com uma configuração maior de cidades.

7.1 Contribuições

O resultado deste trabalho é um método de implementação de autômatos finitos adaptativos com execução em paralelo. Neste método, sempre que, dado sua configuração corrente e o símbolo lido da cadeia de entrada, exista mais de uma transição a ser executada, o modelo irá disparar a execução de todas as transições de uma só vez.

Desta forma, o modelo de autômato pode se tornar mais eficiente, diminuindo seu tempo de execução, como ilustrado nos experimentos realizados. Além disso, a

criação de *threads* para tratar o não-determinismo nos autômatos mostrou ser uma forma eficiente.

Além disso, um *framework* para criação e execução de autômatos finitos adaptativos foi desenvolvido. Este ambiente de desenvolvimento no qual qualquer autômato finito adaptativo pode ser formulado deve servir de apoio para grupos de estudos que fazem pesquisas utilizando este modelo.

Quanto aos resultados do exemplo acima, pode-se concluir que o objetivo de se construir um modelo mais eficiente que o modelo seqüencial foi atingido.

7.2 Trabalhos futuros

Ainda são necessários estudos mais aprofundados quanto a como implementar a paralelização das transições, a fim de melhorar o desempenho do programa. O esquema descrito neste trabalho, utilizando *threads* no *Scheme* se mostra eficiente quanto à velocidade de execução do código. Apesar disso, este esquema consome muitos recursos do sistema nas máquinas baseadas no modelo de von Neuman. O que indica que um controle de criação de *threads*, ou outro esquema de paralelização, é desejável.

Além disso, uma expansão do framework para desenvolvimento de outros modelos adaptativos, principalmente aqueles baseados no modelo de autômatos, pode ser útil para os pesquisadores da área.

É previsto, ainda, implementar a heurística da colônia de formigas (DORIGO, 1999) como um conjunto de funções adaptativas usando o *framework*, comparando o resultado de sua execução com outras heurísticas, como os algoritmos genéticos.

7.3 Conclusão

Este trabalho apresenta um modelo de execução em paralelo dos autômatos finitos e, conseqüentemente, dos autômatos finitos adaptativos. Ao definirmos mecanismos que

fazem as transições em paralelo, o modelo pode tornar-se mais eficiente que o modelo seqüencial, como mostrado na tabela 7.1.

Além disso, um *framework* para desenvolvimento de autômatos finitos adaptativos, paralelo ou seqüencial, é apresentado como resultado final deste trabalho. Esse desenvolvimento tem o intuito de auxiliar futuras pesquisas na área das técnicas adaptativas.

O desenvolvimento desse *framework* visa propor um padrão de implementação destes modelos. Espera-se que este padrão seja de simples compreensão e utilização.

8 Bibliografia

ABELSON, H.; SUSSMAN, G. J.; and SUSSMAN, J. Structure and Interpretation of Computer Programs, *The MIT Press, Cambridge, Mass., and McGraw-Hill New York.*

ABELSON, H.; DYBVIG, R. K.; HAYNES, C.T.; ROZAS, G. J.; ADAMS, N. I.; FRIEDMAN, D. P.; KOHLBECKER, E.; STEELE JR., G. L.; BARTLEY, D. H.; HALSTEAD, R.; OXLEY D.; SUSSMAN, G. J.; BROOKS, G.; HANSON, C.; PITMAN, K. M. and WAND, M. Revised(5) Report on the Algorithmic Language Scheme, 2007.

ALMEIDA JR, J. R., NETO, J.J e HIRAKAWA, A. R. Adaptive Automata for Independent Autonomous Nagivation in Unknown Enviroment. *Annals of ASM 2000 IASTED International Conference on Applied Simulation and Modelling*, Banff, Alberta, 2000.

BACKUS, J. Can Programming be Liberated from the von Neumann Style? *Communications of the ACM 21, 8 (1978) 613-641.*

BARENDREGT, H. P. The impact of the lambda calculus. *Bulletin of Symbolic Logic*, *v. 3, n. 2,* 1997, pp 181-215.

BARENDREGT, H. P. The Lambda Calculus: Its Syntax and Semantics. *North Holland,* 1984

CLINGER, W.; REES, J. Macros that work**,** *Proceedings of the 1991 ACM Conference on Principles of Programming Languages, pp. 155-162.*

DORIGO, M.; CARO, G.; GAMBARDELLA, L. M. Ant Algorithms for Discrete Optimization. Artificial Life, v. 5, n. 2, MIT Press, 1999 pp 137-172.

FELLEISEN, M.; FINDLER, R. B.; FLATT, M.; KRISHNAMURTHI, S. How to Design Programs – An Introduction to Programming and Computing. *The MIT Press* Cambridge, Massachusetts, London, England, 2003

FLANAGAN, C. The Semantics of Futures. *Department of Science – Rice University, 1994.*

GEHANI, N.; MCGETTRICK, A. D. Concurrent Programming. *AT&T Bell Laboratories, 1988.*

GORDON, M. J. C. Programming Language Theory and its Implementation, *London, Prentice Hall, 1988.*

GRAHAM, P. On LISP*, Prentice Hall Ltd., 1993.*

HALSTEAD, R. H. Parallel Symbolic Computing. *The institute of Eletrical and Eletronics Engineers, Inc., 1986*.

HAMMOND, K.; MICHAELSON, G. Research Directions in Parallel Functional Programming. *Springers, UK, 1999.*

HUDAK, P. Conception, Evolution, and Application of Functional Programming Languages. *ACM Computing Survay, Vol 21, No. 3, September 1989, pp 359-411.*

HUDAK, P. Para-funcional Programming. *The institute of Eletrical and Eletronics Engineers, Inc., 1986.*

IWAI, M. K. Um formalismo gramatical adaptativo para linguagens dependentes de contexto, *Tese de Doutorado*, EPUSP, São Paulo, 2000.

JONES, S. L. P. The Implementation of Functional Programming Languages, *Prentice Hall International Ltd., 1987.*

KUMAR, V. Introduction to Parallel Computing – Design and analysis of Algorithms. *The Benjamin/Cummings Publishing Company, Inc., 1994.*

LI, M.; VITÁNYI, P. An introduction to Kolmogorov complexity and its applications, *2nd. ed., New York, Springer-Verlag, 1997.*

LOIDL, H-W.; RUBIO, F.; SCAIFE, N; HAMMOND, K.; HORIGUCHI, S.; KLUSIK, U.; LOOGEN, R.; MICHAELSON, G. J.; PEÑA, R.; PRIEBE S.; REBÓN PORTILLO, A. J. and TRINDER, P. W. Comparing Parallel Functional Languages: Programming and Performance, *Kluwer Academic Publishers,* Netherlands, 2002.

LUZ, J. C.; NETO, J. J. Tecnologa Adaptativa Aplicada à Otimização de Código em Compiladores**.** *IX Congreso Argentino de Ciencias de la Computación, La Plata, Argentina, 6-10 de Outubro, 2003.*

MATUSUNO, I. P. Um Estudo do Processo de Inferência de Gramáticas Regulares e Livres de Contexto Baseados em Modelos Adaptativos, *Dissertação de Mestrado, EPUSP, São Paulo, 2006.*

MCCARTHY, J. Recursive Functions of Symbolic Expressions and Their Computation by Machine, *Communications of the ACM 3, 4 (1960) 184-195*

MENEZES, C. E. D. de e NETO, J. J. Um método híbrido para a construção de etiquetadores morfológicos, aplicado à língua portuguesa, baseado em autômatos adaptativos. *Anais da Conferencia Iberoamericana en Sistemas, Cibernética e Informática*, 19-21 de Julio, 2002, Orlando, Florida.

NETO, J. J. Contribuições à metodologia de construção de compiladores. *Tese de Livre Docência*, EPUSP, São Paulo,1993.

NETO, J. J. Adaptive Automata for Context-dependent Sensitive Languages. *SIGPLAN NOTICES, Vol. 29, n. 9, pp. 115-124, September, 1994.*

NETO, J. J. Adaptive Rule-Driven Devices - General Formulation and Case Study, *Lecture Notes in Computer Science. Watson, B.W. and Wood, D. (Eds.): Implementation and Application of Automata 6th International Conference, CIAA 2001, Vol.2494, Pretoria, South Africa, July 23-25, Springer-Verlag, 2001, pp. 234-250.*

NETO, J. J. Adaptive Rule-Driven Devices - General Formulation and Case Study, I*n: CIAA'2001 Sixth International Conference on Implementation and Application of Automata. Pretoria, South Africa: [s.n.], 2001. p. 234.250.*

NETO, J. J. and MORAES, M. de. Using Adaptive Formalisms to Describe Context-Dependencies in Natural Language**.** *Lecture Notes in Artificial Intelligence. N.J. Mamede, J. Baptista, I. Trancoso, M. das Graças, V. Nunes (Eds.): Computational Processing of the Portuguese Language 6th International Workshop*, PROPOR 2003, Volume 2721, Faro, Portugal, June 26-27, Springer-Verlag, 2003, pp 94-97

PAPADIMITRIU, C. H.; VEMPALA, S. On the Appoximability of the Travaling Salesman Problem. *Proceedings of the 32nd Annual ACM Symposium on Theory of Computing, 2000.*

PISTORI, H. Tecnologia Adaptativa em Engenharia de Computação: Estado da Arte e Aplicações. *Tese de Doutorado, EPUSP, São Paulo, 2003.*

PISTORI, H., MARTINS, P. S., CASTRO JR., A. A. Adaptive Finite State Automata and Genetic Algorithms - Merging Individual Adaptation and Population Evolution. *Proceedings of International Conference on Adaptive and Natural Computing Algorithms - ICANNGA 2005*, Coimbra, March 21-23, 2005.

PULSON, L. Foundations of Functional Programming, Computer Laboratory, *University of Cambridge, 1995.*

QUEINNEC, C. A Concurrent and Distributed Extension of Scheme, *Parallel Architectures and Languages Europe,* Paris, France, 1992, pp. 431-446.

RICCHETTI, P. M. Geração de Analisadores Sintáticos Baseados em Transdutores a partir de Especificações Gramaticais, *Dissertação de Mestrado, EPUSP, São Paulo, 2005.*

ROCHA, R. L. A. A Proposal of a Computational Device Based on Adaptive Automata, *Proceedings of the International Congress of Logic Applied to Technology (LAPTEC'2001), IOS Press/Ed.* Pleiade, Sao Paulo, Brazil, 2001, pp. 145-152.

ROCHA, R. L. A. Um Estudo sobre o Método Indutivo e sua Relação com a Computação. *Anais do I Congresso de Lógica Aplicada à Tecnologia - LAPTEC´2000, São Paulo: Faculdade SENAC de Ciências Exatas e Tecnologia, p.95 – 109, 2000.*

ROCHA, R. L. A. Um método de escolha automática de soluções usando tecnologia adaptativa*. 211p. Tese de Doutorado, Escola Politécnica, Universidade de São Paulo, São Paulo, 2000.*

ROCHA, R. L. A.; GARANHANI, C. E. C. Parallel adaptive finite state automata, *Proceedings of the XII Argentine Congress in Computer Science (CACIC'2006) (2006) 1–12*

ROCHA, R. L. A.; NETO J. J. Uma proposta de Linguagem de Programação Funcional com Características Adaptativas, *Proceedings of the IX Congreso Argentino de Ciencias de Computación – CACIC'2003, CD/ROM, Buenos Aires, Argentine, 2003, pp. 1664-1674*

ROCHA, R. L. A; NETO, J. J. Construction of Models based on Adaptive Automata Through Grammar Descriptions, *Proceedings of the IASTED International Conference on Applied Simulation and Modeling (ASM'2000), IASTED/ACTA Press, Calgary, Canada, 2000, pp. 228-234*

SCHRIJVER, A. On the history of combinatorial optimization (till 1960), *Handbook of Discrete Optimization (K. Aardal, G.L. Nemhauser, R. Weismantel, eds.), Elsevier, Amsterdam, 2005, pp. 1-68.*

SEBESTA, R. T. W. Concepts of Programming Languages. *Addison-Wesley Publishing Company, Inc, 2003.*

SUTTER, H; LARUS, J. Software and the concurrency revolution. *Queue, v. 3, n. 7, ACM, New York, USA, 2005*.

VALIANT, L. G. A theory of learnable. *Communications of the ACM, v.27, n.11, 1984, pp.1134-1142.*

WANG, P. S. Symbolic Computation and Parallel Software, *Department of Mathematics and Computer Science, Kent State University, Kent, Ohio, 1991*

WOLFE, M. High Performance Compilers for Parallel Computing. *Addison-Wesley Publishing Company, Inc., 1996.*