EDER JOSÉ PELEGRINI

CÓDIGOS ADAPTATIVOS E LINGUAGEM PARA PROGRAMAÇÃO ADAPTATIVA: CONCEITOS E TECNOLOGIA

SÃO PAULO
2009

EDER JOSÉ PELEGRINI

CÓDIGOS ADAPTATIVOS E LINGUAGEM PARA PROGRAMAÇÃO ADAPTATIVA: CONCEITOS E TECNOLOGIA

Dissertação apresentada à Escola Politécnica da Universidade de São Paulo para obtenção do Título de Mestre em Engenharia.

SÃO PAULO
2009

EDER JOSÉ PELEGRINI

CÓDIGOS ADAPTATIVOS E LINGUAGEM PARA PROGRAMAÇÃO ADAPTATIVA: CONCEITOS E TECNOLOGIA

Dissertação apresentada à Escola Politécnica da Universidade de São Paulo para obtenção do Título de Mestre em Engenharia.

Área de Concentração:
Sistemas Digitais

Orientador:

Prof. Dr. João José Neto

SÃO PAULO
2009

Este exemplar foi revisado e alterado em relação à versão original, sob responsabilidade única do autor e com a anuência de seu orientador.

São Paulo, de maio de 2009.

Assinatura do autor _____

Assinatura do orientador _____

FICHA CATALOGRÁFICA

Pelegrini, Eder José
Códigos adaptativos e linguagem para programação adaptativa : conceitos e tecnologia / E.J. Pelegrini. -- ed.rev. -- São Paulo, 2009.
143 p.

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

1. Computação reconfigurável 2. Linguagem de programação I. Universidade de São Paulo. Escola Politécnica. Departamento de Engenharia de Computação e Sistemas Digitais II. t.

AGRADECIMENTOS

Ao amigo e orientador Professor Doutor João José Neto, pelo apoio, pela paciência, pelos conselhos e opiniões ao longo do desenvolvimento desse trabalho, o qual não seria possível sem a sua ajuda.

Ao Professor Doutor Ricardo Luis de Azevedo Rocha e ao Doutor César Alberto Bravo Pariente, que contribuíram para o aperfeiçoamento desse trabalho.

Ao Professor Doutor Aparecido Valdemir de Freitas, cujo trabalho na aplicação da Tecnologia Adaptativa em linguagens de programação e códigos serviu de inspiração para o desenvolvimento dessa dissertação.

A minha família, pela compreensão e apoio dado durante essa jornada do conhecimento.

A todas as pessoas que diretamente ou indiretamente contribuíram com a realização desse trabalho.

EPÍGRAFE

"*There is nothing either good or bad,*
but thinking makes it so."

(William Shakespeare, "Hamlet")

RESUMO

Esse trabalho relata o estudo sobre a aplicação da tecnologia adaptativa na área de linguagens de programação e códigos, tendo como objetivo a proposição de mecanismos que permitam a escrita de códigos que possam se auto-modificar segundo os conceitos da tecnologia adaptativa. Essa proposta é feita por meio da descrição de uma linguagem de montagem para programação adaptativa e de seu mecanismo de execução.

Em adição à linguagem, foi desenvolvido um ambiente de execução baseado em um novo dispositivo adaptativo (autômato de execução adaptativo), com o intuito de evitar certas dificuldades existentes à modificação de código impostas pelos mecanismos de execução. Para poder representar e executar códigos adaptativos, esse dispositivo agrega características de execução de ambientes tradicionais e dos autômatos adaptativos.

Essa dissertação apresenta o resultado dessa pesquisa, consolidando os conceitos desenvolvidos por meio de exemplos de funcionamento de códigos adaptativos e considerações sobre a linguagem.

Palavras-Chave: Computação Reconfigurável, Linguagem de Programação.

ABSTRACT

This research studies the application of the Adaptive Technology in the field of programming Language and codes, which objective is the proposition of a mechanism designed to build codes capable of self-modifying in compliance which the concepts of Adaptive Technology. This proposal is formulated by means of the description of an assembly language for adaptive programming and its run-time mechanism.

In addition to the language proposal, a run-time mechanism based on a new adaptive device (adaptive execution automata) is conceived. That avoids some difficulties to perform code modification existent in traditional run-time mechanisms. In order to represent and execute the adaptive codes, this device aggregates characteristic of the traditional run-time mechanism and the adaptive automata.

The present dissertation describes the results of this study, consolidating the concepts developed by means of examples about the code execution operation and considerations about the language.

Keywords: Reconfigurable computing, Programming Language.

LISTA DE ILUSTRAÇÔES

LISTA DE TABELAS

SUMÁRIO

1 INTRODUÇÃO

A Tecnologia Adaptativa compreende o ramo de pesquisa no qual os dispositivos apresentam a característica de se auto-modificarem dinamicamente em resposta a estímulos de entrada, sem interferência de agentes externos.

Os dispositivos adaptativos habitualmente são propostos por meio da extensão de dispositivos não-adaptativos consolidados, adicionando a estes a capacidade de auto-modificação. Freitas e Neto (FREITAS; NETO, 2005) (FREITAS, 2008), vinculam os conceitos de programação e linguagens de programação ao conceito de adaptatividade, com o intuito de criarem programas cujas instruções que o compõem possam se auto-modificar seguindo os conceitos da Tecnologia Adaptativa.

O conceito de códigos com a capacidade de se auto-modificarem não é recente. É possível utilizar determinadas linguagens de programação de baixo nível (como a linguagem de montagem (AHO et al., 2006), (DETMER, 2001)), para escrever códigos que alteram suas instruções durante a execução. Para esses casos, a máquina associada a tal linguagem não deve apresentar algum tipo de mecanismo de proteção que impossibilite um determinado código de modificar a si mesmo (um exemplo desse tipo de mecanismo é a proteção contra a escrita na área de memória onde se encontra o código do programa). A alteração de uma instrução pode, nesse caso, ser feita por meio da execução de certas instruções de máquina (exemplo: a instrução `mov` da arquitetura Intel® 32 bits) que promovem a movimentação dos códigos desejados para o endereço que deverá conter o código em questão.

Alguns exemplos que utilizam linguagens de baixo nível para a escrita de códigos auto-modificáveis podem ser encontrados: (i) em (ANCKAERT; MADOU; BOSSCHERE, 2001), onde o processo de alteração de código na linguagem de montagem é ilustrado por meio de um exemplo; (ii) em (SCHEIBLER, 2008), que exemplifica como escrever código auto-modificável utilizando à linguagem de montagem associado aos processadores da arquitetura Intel® 32 bits no sistema operacional Linux; e (iii) em (GIFFIN; CHRISTODORESCU; KRUGER, 2005), trabalho que descreve como usar códigos auto-modificáveis, escritos em linguagem de baixo nível, para garantir a integridade de um código por meio de *self-checksumming*.

Entretanto, ao se analisar as técnicas e os modelos usados para escrita de códigos auto-modificáveis na linguagem de montagem (estudando trabalhos como o desenvolvido em (ANCKAERT; MADOU; BOSSCHERE, 2001)) e tendo em mente os conceitos gerais da arquitetura de computadores, como os apresentados em (HENNESSY; PATTERSON, 2003), é possível observar características da linguagem, bem como do seu mecanismo de execução, os quais permitem afirmar que ambos não foram projetados visando priorizar a escrita de códigos que modifiquem a suas próprias instruções (apesar de ser possível escrever tais programas).

Adicionando a tal argumentação o fato de que grande parte das técnicas formais de verificação de códigos pressuporem que o código de um programa armazenado na memória é fixo e imutável (HONGXU; ZHONG; ALEXANDER, 2007), têm-se que a compreensão e depuração de programas contendo códigos auto-modificáveis podem ser consideradas atividades complexas.

Seja pelos motivos citados anteriormente ou por outros motivos, a auto-modificação é um recurso pouco explorado (ANCKAERT; MADOU; BOSSCHERE, 2001) (FREITAS; NETO, 2007).

Motivado pelo interesse por este tipo de programação e pelas pesquisas desenvolvidas na área de tecnologia adaptativa (NETO, 2007) (FREITAS; NETO, 2007) dentro do Laboratório de Linguagens e Técnicas Adaptativas (LTA, 2006), e na área de *hardware* reconfigurável (KEUNG; TYAGI, 2006), este trabalho propõe um mecanismo para o suporte a auto-modificação de códigos escritos em linguagens de baixo nível.

Ao importar os conceitos da tecnologia adaptativa na elaboração dos mecanismos de modificação de código, espera-se que, além de permitir a escrita de códigos com a capacidade de se auto-modificarem, o processo de auto-modificação seja estruturado segundo os conceitos dessa tecnologia, visando assim organizar a utilização de tal recurso. Adicionalmente, espera-se garantir a coerência sintática do código a cada auto-modificação (i.e. a auto-modificação do código não deve introduzir erros de caráter sintáticos).

Para descrever tais mecanismos é proposta uma linguagem de programação de baixo nível com suporte a mecanismos de modificação, bem como seu ambiente de execução. Espera-se que os recursos de auto-modificação não apresentem grandes

dificuldades de uso, possibilitando que futuramente esta linguagem seja usada como substrato por propostas de linguagens de alto nível. Ou seja, espera-se que propostas de linguagens de alto nível para programação adaptativa poderão mapear os recursos de auto-modificação fornecidos nos recursos existentes na linguagem de baixo nível.

Por fim, ressalta-se que este trabalho não busca discutir vantagens e desvantagens de utilizar um estilo de programação baseado em auto-modificação de código, nem citar em quais situações pode ou não ser mais vantajosa, uma vez que a técnica de programação utilizada é, freqüentemente, uma opção do programador. Tal argumento é reforçado pelo fato de que a proposta aqui presente possui a mesma expressividade que outras máquinas e linguagens de programação capazes de simular máquina de Turing com memória finita. O objetivo é apresentar mais uma opção de programação, que cria um novo tipo de identidade similar ao da equivalência entre a implementação de um algoritmo via *software* ou via *hardware*: a possibilidade de implementar estrutura de dados dinâmicas (como uma lista ligada) via código, bem como descrever um *hardware* virtual (ambiente de execução) que, baseado em dispositivos adaptativos, consiga executar os códigos adaptativos propostos ao longo desse trabalho.

1.1 MOTIVAÇÃO

Código com a capacidade de alterar as suas próprias instruções ao longo de sua execução vem sendo utilizado na construção de programas com diversas finalidades. Pode-se citar como exemplos: otimização de código (MASSALIN, 1992), proteção do código-fonte (GIFFIN; CHRISTODORESCU; KRUGER, 2005), esconder detalhes internos de um programa (ANCKAERT; MADOU; BOSSCHERE, 2001) (KANZAKI et al., 2003) (MADOU et al., 2006), construção de *software* com a capacidade de resistir e recuperar-se de erros e falhas (*resilient software*) (TSCHUDIN; YAMAMOTO, 2006). Adicionalmente, este tipo de código mostra-se interessante para a construção de outras aplicações, como na segurança de dados

(seguindo como exemplo o esquema de segurança apresentado em (PELEGRINI; NETO, 2007).

No entanto, conforme mencionado anteriormente, as linguagens de montagem de máquinas usuais, bem como os mecanismos de execução tradicionais, não se apresentam como mecanismos nativos práticos para a implementação de códigos auto-modificáveis.

Os mecanismos de execução tradicionais operam como se o programa fosse um vetor de instruções. Após a execução de uma instrução, a próxima a ser computada é determinada implicitamente, sendo a instrução contida na posição seguinte do vetor, exceto nos casos de instruções que quebrem a seqüência de execução (exemplos: instrução de salto, chamada e retorno de procedimento).

Em termos de endereçamento físico, a próxima posição do vetor representa o endereço da última instrução executada mais o número de posições de memória que a instrução previamente executada ocupa (HENNESSY; PATTERSON, 2003). Ou seja, as máquinas trabalham com um esquema de execução cujo funcionamento padrão é baseado em iteração, no qual o endereço onde se inicia a primeira instrução a ser executada constitui o caso inicial da iteração, e o passo iterativo, que calcula o endereço onde se inicia a próxima instrução a ser executada, é feito por meio da soma do endereço onde se inicia a instrução associada ao passo anterior com o tamanho dessa instrução. As instruções que determinam o próximo passo da computação, como as instruções de salto, constituem-se exceções ao funcionamento padrão.

Este modelo de execução dificulta a inserção e remoção de instruções, uma vez que exige a realocação das mesmas, para garantir a consistência com o modelo de vetor de instruções, e, conseqüentemente, o devido ajuste das referências feitas às posições realocadas.

Para ilustrar o mecanismo descrito, considere-se o exemplo de código mostrado na Figura 1. Para facilitar a compreensão, apenas referências a endereços de instrução estão expressos numericamente (as instruções, registradores e referências a dados são aqui mantidos em notação simbólica). Para simplificar o exemplo, adota-se a hipótese que referência a endereços e valores imediatos ocupem apenas uma posição de memória.

Endereço	Instrução *Assembly*
0	`mov eax,5`
3	`mov ebx,3`
6	`add eax,3`
9	`cmp eax,8`
12	`je 17`
14	`add eax,1`
17	`nop`

Figura 1 – Exemplo de um programa *assembly*.

Caso se deseje inserir a instrução `add eax,1` entre as duas primeiras instruções do trecho de código mostrado na Figura 1, ou seja, entre as instruções que começam no endereço `0` e no endereço `3`, será necessário realocar todo o código contido na memória referente às posições `3` e seguintes, bem como realizar as devidas atualizações nos operandos que referenciam posições de instruções, como é o caso do operando da instrução de desvio condicional `je 17`, o qual após a realocação do código, deve passar a referenciar o endereço `20` (i.e. em caso de desvio, a execução é desviada para a instrução associada à posição `20`). O código resultante da inserção da instrução é apresentado na Figura 2(a).

Endereço	Instrução
0	`mov eax,5`
3	`add eax,1`
6	`mov ebx,3`
9	`add eax,ebx`
12	`cmp eax,8`
15	`je 20`
17	`add eax,1`
20	`nop`

(a)

Endereço	Instrução
0	`je eax`
2	`mov ebx,3`
5	`add eax,ebx`
8	`cmp eax,8`
11	`je 16`
13	`add eax,1`
16	`nop`

(b)

Figura 2 – Exemplo de modificação do código da Figura 1. (a) Inserindo uma nova instrução. (b) Alterando a primeira instrução para `je eax`.

Outro ponto importante reside no fato de a arquitetura do *hardware* em questão não oferecer uma instrução específica que seja apropriada para a alteração de código (o

termo apropriada refere-se ao tratamento da modificação de código como um todo, tratando as relocações e as realocações necessárias após a alteração de uma ou mais instruções), sendo por esta razão usada também para isso a instrução de movimentação de dados (`mov`). Apesar de ser suficiente para alterar o conteúdo de uma única posição de memória, esta instrução não é auto-suficiente para realizar, por si só, inserções e remoções de uma instrução isolada ou de um grupo de instruções, nem garante a integridade sintática do programa resultante, uma vez que as diversas instruções possuem geralmente um número variável de operandos (exemplo: a instrução de soma – `add` – da linguagem *assembly* associada à arquitetura Intel® 32 bits possui obrigatoriamente dois operandos, enquanto a instrução de retorno de procedimento – `ret` – utiliza-se de nenhum ou de um operando) e aceitam diferentes tipos de operandos (exemplo: a instrução de pilha `push` aceita como primeiro – e único – operando um valor imediato, enquanto a instrução de soma `add` não aceita o valor imediato como primeiro operando).

Considere o caso de mover o código da instrução `je` para o endereço `0` (que armazena o código de operação da instrução `mov`) do código apresentado na Figura 1. Desta operação resultaria um código incoerente, dado que as primeiras três posições de memória conteriam `je eax` e a constante `5`, que não faz parte da instrução recém-movida, pois uma instrução `je` ocupa apenas 2 bytes. Para evitar esse efeito, a substituição deve ser acompanhada da correta realocação do código, que consiste na remoção de três posições de memória e da inserção de duas outras (acompanhadas, naturalmente, das devidas realocações das demais instruções), resultando assim em um código sintaticamente integro, como mostra a Figura 2(b).

As constantes necessidades de realocações dificultam o controle sobre as operações de auto-modificação do código. Como as realocações alteram os endereços associados às instruções, é necessário acompanhar cuidadosamente as modificações do código para manter a sua integridade, bem como para possibilitar que as futuras alterações atendam ao planejamento do programador. Para ilustrar este inconveniente, suponha-se alterar a instrução `mov ebx,3` (segunda instrução do código mostrado na Figura 1, que começa no endereço `3`) após a alteração que resulta no código apresentado na Figura 2(a). Caso o programador não preveja adequadamente a ordem das alterações efetuadas ou não preste suficiente atenção

nas modificações efetuadas, ele poderá vir a modificar a instrução que começa no endereço `3`, o qual anteriormente iniciava o armazenamento da instrução `mov ebx,3`. Entretanto, devido à primeira realocação (causada pela primeira modificação do código), o endereço referente à segunda instrução do código original passou de `3` (Figura 1) para `6` (Figura 2(a)).

É possível optar por um modelo de escrita que não contemple tais realocações e relocações dos códigos. Para tanto, escreve-se o código a ser inserido em posições de memórias não-utilizadas e, por meio da instrução de salto, liga-se as posições de memórias adequadamente. Essa metodologia de escrita, em processadores com instrução de tamanho variável, pode acarretar no subaproveitamento do espaço de memória. Mesmo que as instruções tivessem tamanho fixo, essa abordagem ainda exige o gerenciamento da memória do programa por parte do programador (como, por exemplo, a distinção de posições de memórias utilizadas das não utilizadas). A problemática desse gerenciamento é idêntica ao estudado em gerenciamento de memória descrito em (TANENBAUM, 2003) (SILBERSCHATZ; GALVIN; GAGNE, 2005).

Não obstante, é necessário conhecer as posições de memória em que os novos códigos são adicionados e a ordem das posições que reconstituem o código. Essas informações são necessárias ao se modificar grupos seqüenciais de instruções.

Motivado pelos trabalhos na área de *hardware* reconfigurável (KEUNG; TYAGI, 2006), pela ausência de mecanismos práticos de modificação de código e pelos conceitos da Tecnologia Adaptativa, esse trabalho procura explorar o tema de códigos adaptativos e linguagens para programação adaptativa (vide seção de conceitos).

1.2 OBJETIVOS

Este trabalho apresenta os estudos e utilização dos conceitos da tecnologia adaptativa e dos dispositivos adaptativos para especificar um mecanismo de modificação de código (programas de computadores) e construir o ambiente de

execução desse código, bem como estender a área de aplicação da tecnologia adaptativa.

A estruturação da proposta de códigos com a capacidade de automodificar segundo os conceitos da tecnologia adaptativa é feito por meio de um artifício: a descrição de uma linguagem para programação adaptativa. Essa linguagem, baseada em linguagens de baixo nível (no caso, instruções *assembly*), é projetada para dar suporte à alteração de código, no nível das instruções da máquina.

O esquema de modificação do código e a linguagem para programação adaptativa serão especificados de forma a impor algumas restrições ao processo de alteração, definindo o que pode ser alterado e como modificá-lo, com o intuito de estabelecer uma disciplina de alteração de código, baseada nos conceitos da Tecnologia Adaptativa. Essa disciplina visa facilitar a construção, manuseio e o entendimento de códigos com a capacidade de se auto-modificarem.

Associado a essa linguagem, também será definido um ambiente de execução responsável pelo apoio à alteração do código e pela garantia da manutenção de sua integridade. Para a proposição deste mecanismo, foi utilizado como ponto de partida um formalismo desenvolvido no contexto dos fundamentos da Tecnologia Adaptativa, o autômato adaptativo.

Como o mecanismo de execução proposto difere do modelo executado pela máquina física, a linguagem é proposta como uma linguagem intermediária, a ser executada por uma máquina virtual (NAIR; SMITH, 2005).

Resumindo, o objetivo dessa pesquisa é definir um processo estruturado de modificação de código, baseado nos conceitos da tecnologia adaptativa, e do ambiente de execução desse código.

1.3 ORGANIZAÇÃO

Este documento encontra-se organizado em sete capítulos e dois anexos, sendo que:

- O segundo capítulo apresenta os conceitos associados a esse trabalho. Por questões de estruturação do texto, os conceitos foram logicamente divididos em: modelos de processamento, tecnologia adaptativa, programação e programação adaptativa e, por fim, linguagens de programação e adaptatividade.
- O capítulo 3 descreve a proposta da linguagem de montagem para programação adaptativa AdaptCode. Nesse capítulo é apresentada a forma da escrita de código adaptativo por meio dessa linguagem. O foco reside na descrição do ferramental de modificação de código, realizado por meio de três instruções denominadas `adapt`, `mark` e `emark`.
- O capítulo 4 descreve o dispositivo adaptativo denominado de autômato de execução adaptativo, base para a proposição do ambiente de execução da linguagem AdaptCode. Esse dispositivo adaptativo de processamento une características do autômato adaptativo com características de execução dos computadores, de forma a aproveitar as funções adaptativas como base para o mecanismo de auto-modificação dos códigos escritos na linguagem AdaptCode;
- O capítulo 5 contém o detalhamento do ambiente de execução associado à linguagem Adaptcode. O objetivo desse capítulo é apresentar como o processo de modificação do código funciona como um todo. Para tanto é apresentado um modelo de ambiente de execução baseado em máquina de Turing, cuja função é didática (detalhar os mecanismos). Na seqüência, esse capítulo apresenta a estruturação desse ambiente e como ele possibilita a execução de códigos adaptativos em ambientes não-adaptativos;
- O capítulo 6 desenvolve exemplos de programas escritos na linguagem, bem como ilustra o seu funcionamento. Adicionalmente, apresenta algumas considerações sobre o protótipo desenvolvido ao longo dessa pesquisa;
- O capítulo 7 apresenta os comentários finais
- Referências bibliográficas;
- O anexo 1 apresenta a prova de equivalência entre modelos de autômatos adaptativos que realizam a chamada de função adaptativa na transição ou no estado;
- O anexo 2 detalha como escrever máquinas de Turing (com memória limitada) na linguagem AdaptCode, visando provar que a linguagem tem poder de máquina de Turing (com memória limitada).

2 CONCEITOS

Este capítulo constitui-se de uma breve revisão conceitual sobre diversos tópicos citados ou associados a esse trabalho. Com o intuito de organizar a apresentação de tais conceitos, os mesmos foram organizados nos seguintes tópicos: modelos de processamente; tecnologia adaptativa; programação e programação adaptativa; e, por fim, linguagens de programação e adaptatividade.

No tópico sobre modelos de processamento, apresenta-se uma breve discussão acerca dos conceitos associados à máquina de Turing, computadores e máquinas virtuais.

No tópico acerca da tecnologia adaptativa, o foco da revisão concentra-se sobre a definição de termos associados, bem como descreve os autômatos adaptativos (base para a proposição do autômato de execução adaptativo) e o dispositivo adaptativo de propósito geral.

No tópico sobre programação e programação adaptativa é feita a conceitualização dos termos bases utilizados nessa dissertação, como programa, código e programa adaptativo. O intuito desse tópico é completar e padronizar a utilização dos termos apresentados ao longo desse trabalho.

Por fim, no tópico de linguagem de programação e adaptatividade definem-se os termos linguagem de programação e linguagem para programação adaptativa.

2.1 MODELOS DE PROCESSAMENTO

Um dos fundamentos bases de computação é o estabelecimento de modelos matemáticos formais capazes de realizarem computação. Podem-se citar como exemplos de modelos de computação: autômatos, máquinas de Turing, cálculo λ e funções μ-recursivas (LEWIS; PAPADIMITRIOU, 1981).

Esses modelos costumam ser categorizados em função do seu poder de expressão. Essa categorização consiste em associar esses modelos aos tipos de linguagens capazes de representar, reconhecer ou gerar. As linguagens, por sua vez, são

classificadas de acordo com algum esquema, sendo comumente utilizada a hierarquia de Chomsky (NETO, 1987) (LEWIS; PAPADIMITRIOU, 1981).

Dentre os modelos de computação formais com maior poder de expressão, pode-se citar a máquina de Turing (detalhada no item 2.1.1). Assim como alguns outros formalismos (exemplos: funções μ-recursivas, cálculo λ e autômatos adaptativos), a máquina de Turing é capaz de aceitar as linguagens do tipo 0 (também chamadas de linguagens irrestritas). Dentro da hierarquia de Chomsky, essas linguagens são as mais gerais, sendo que quando descritas por meio de gramáticas, as regras de produções gramaticais não apresentam nenhum tipo de restrição (LEWIS; PAPADIMITRIOU, 1981) (NETO, 1987) (NETO; PARIENTE, 2003). Dentro da classe de linguagens irrestritas, máquinas de Turing são capazes de reconhecer as classes de linguagens recursivas (LEWIS; PAPADIMITRIOU, 1981).

Em função do poder de expressão da máquina de Turing e de possuir características similares ao de computadores (i.e. computadores pessoais), adota-se que máquina de Turing é associada ao conceito de algoritmo, ou seja, um procedimento não é considerado algoritmo a não ser que possa ser representado por meio de máquina de Turing (LEWIS; PAPADIMITRIOU, 1981) (BLASS; GUREVICH, 2003). Ressalta-se que isso é uma tese (tese de Church), dado que não é matematicamente provado (LEWIS; PAPADIMITRIOU, 1981).

Portanto, a tese de Church pode ser visualizada como uma tentativa de estabelecer o que é e o que não é um algoritmo (LEWIS; PAPADIMITRIOU, 1981). Apesar de contestada por discussões como as publicadas em (BLASS; GUREVICH, 2003) e (BRINGSJORD; FERRUCI, 1999), assume-se, nessa dissertação, que o termo algoritmo é similar ao de função computável, e designa todos os algoritmos que podem ser computados por máquina de Turing.

Máquinas de Turing são dispositivos teóricos, devido à consideração de memória infinita. Adicionalmente, seu caráter genérico e as definições de apenas quatro operações tornam a tarefa de representar um procedimento em máquina de Turing algo complicado, quando comparado com máquinas que fornecem um conjunto de instruções otimizado para a representação do problema. Por exemplo, a representação de uma soma binária da máquina de Turing demanda a definição do processo e da forma de representação dos números, enquanto em uma máquina

baseado na arquitetura Intel® 32 bits resume-se a utilizar a instrução de soma binária.

Como conseqüência, em termos de utilização prática, o destaque fica para outra máquina: as máquinas PC-compatível (comumente chamado de computadores).

O termo computador é associado a qualquer máquina universal e programável, composta por três elementos: CPU, Memória e Entrada/Saída (DETMER, 2001) (HENNESSY; PATTERSON, 2003), onde:

- O CPU é o *hardware* responsável por executar as instruções de um programa que se encontra armazenado na memória, bem como localizar a posição (na memória) dos dados e das instruções;
- A memória é o *hardware* responsável por armazenar as informações (incluindo o programa a ser executado);
- As entradas/saídas consistem nos *hardwares* dedicados a operações de fornecer (entrada – exemplo: teclado) ou extrair (saída – exemplo: monitor) informações do computador.

Devido à diversidade de arquiteturas e organizações de sistemas computacionais, como conjunto de instruções, hierarquia de memória, políticas de cache, tamanho de instruções, dentre outros (HENNESSY; PATTERSON, 2003) (SILBERSCHATZ; GALVIN; GAGNE, 2005) (TANENBAUM, 2003), existem diversos tipos de computadores, cada qual apresentando vantagens e desvantagens em determinadas aplicações.

Os computadores são modelos de processamento que possuem o mesmo poder de expressão da máquina de Turing Universal com fita finita. Esse fato é constatado pela possibilidade de simular uma máquina de Turing com fita finita em um computador. O programa Visual Turing V1 (CHERANSOFT, 2006) permite a declaração e simulação do funcionamento de tal máquina em um computador PC compatível.

O funcionamento de um computador pode ser, simplificadamente, descrito pela execução cíclica dos seguintes passos: (i) buscar, na memória, a instrução a ser executada; (ii) executar, na CPU, a instrução previamente encontrada; (iii) calcular o endereço, na memória, da próxima instrução a ser executada. O ciclo de execução

somente é interrompido caso durante o passo (ii) a instrução a ser executada consista na instrução de parada (*halt*).

2.1.1 Máquina de Turing

Dentre os diversos modelos de computação, destaca-se a máquina de Turing. A máquina de Turing pode ser descrita na forma de um dispositivo baseado em regras que é composta por dois elementos: uma fita infinita, a qual representa a memória do dispositivo, e uma máquina de estados finitos, responsável pelo controle da execução do dispositivo (LEWIS; PAPADIMITRIOU, 1981).

A fita representa uma seqüência de infinitos símbolos (à direita) dispostos seqüencialmente como indicado pela Figura 3. A manipulação dessa fita (leitura ou escrita da fita) pela máquina de controle é feita por meio de um cabeçote, que pode ser deslocado para a esquerda ou para a direita.

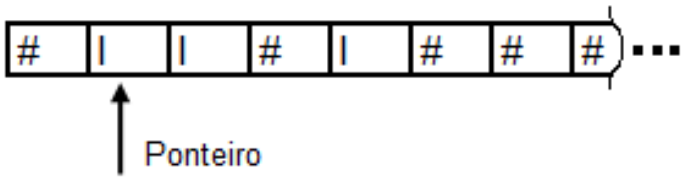


Figura 3 - Exemplo de Fita para Máquina de Turing

Uma das formulações de máquina de Turing consiste numa 4-upla (K, Σ, δ, s), onde: (i) K é um conjunto finito de estados; (ii) Σ é o alfabeto de símbolos que podem ser escritos na fita. Este conjunto deve incluir o símbolo # (símbolo branco) e não pode conter os símbolos L e R; (iii) s é o estado inicial (pertencente a K); e (iv) δ é uma função da forma $K \times \Sigma \rightarrow (K \cup \{h\}) \times (\Sigma \cup \{L,R\})$, onde h é um estado especial que indica o término da computação (estado *halt* que não é especificado em K), L e R denota a ação de mover o cabeçote da fita para a esquerda e para a direita, respectivamente (LEWIS; PAPADIMITRIOU, 1981).

Em cada passo da computação de uma máquina de Turing, executam-se as seguintes operações em função do estado corrente e do símbolo da fita indicado pelo cabeçote:

- Passo 1: Transita-se a máquina de controle para um novo estado corrente;
- Passo 2: Executa-se uma das seguintes ações possíveis: mover o cabeçote para o símbolo à direita, mover o cabeçote para o símbolo à esquerda ou escrever um determinado símbolo (pertencente a Σ) na posição da fita indicada pelo cabeçote.

Estas duas operações formam a aplicação de uma das regras definidas em δ. O processo de computação começa no estado inicial s e termina ao atingir o estado de *halt* (h). Ressalta-se que para alguns casos (como aceitação de linguagens irrestritas), a máquina pode entrar em *loop* e nunca atingir o estado de *halt* (chamados de casos em que o reconhecimento é Turing aceitável (LEWIS; PAPADIMITRIOU, 1981)).

Diversas modificações e recursos para a declaração de máquina de Turing foram elaborados, com o intuito de facilitar a utilização da mesma. Dentre eles, pode-se citar o uso de mais de uma fita infinita ou a composição de máquinas de Turing. Destaca-se que essas modificações não alteram o poder de expressão da máquina de Turing.

A máquina de Turing universal é uma máquina de Turing capaz de simular a execução de outras máquinas de Turing, ou seja, adere ao conceito de "programabilidade" (i.e. capacidade de criar programas com diferentes propósitos a serem executados em uma mesma máquina).

Para tanto, é definida uma codificação para especificar uma máquina de Turing, de forma que esta possa ser declarada na fita (um exemplo de possível codificação encontra-se descrito em (LEWIS; PAPADIMITRIOU, 1981)). A máquina de Turing universal é construída de forma a ser capaz de interpretar as informações contidas na fita, simulando o comportamento da máquina codificada na fita.

2.1.2 Máquinas Virtuais

A virtualização é uma técnica utilizada por diversos sistemas computacionais com o objetivo principal de desacoplar os recursos físicos de um determinado *hardware* da sua representação lógica a um usuário (NAIR; SMITH, 2005) (BARHAM et al., 2003). Para este fim é introduzida uma camada de abstração, a qual permite a criação de múltiplas instâncias lógicas, cada qual de posse de uma fração da totalidade dos recursos de uma instância física, ou ainda, a criação de uma única instância lógica a partir de múltiplas instâncias físicas (BARHAM et al., 2003).

A técnica de virtualização pode ser aplicada em diversos níveis de um sistema computacional. É possível virtualizar o *hardware*, a arquitetura de computador, o sistema operacional, o ambiente de execução de uma linguagem (NAIR; SMITH, 2005), dentre outros níveis. A versatilidade de aplicação permite o desenvolvimento de uma gama de aplicações (como tradução de binários, interpretação de código, execução de múltiplos sistemas operacionais em uma máquina simultaneamente).

A virtualização é a construção de um isomorfismo que mapeia um sistema virtual em um sistema real. O processo de virtualização consiste na construção de uma camada de *software* responsável por mapear as operações fornecidas pelo sistema virtual (hóspede) nas operações fornecidas pelo sistema físico (hospedeiro) (NAIR; SMITH, 2005). A camada de software, responsável por prover uma interface entre os recursos reais e as máquinas virtuais freqüentemente é chamada de monitor de máquina virtual (ROSE, 2004).

Pode-se citar como exemplos de sistemas de virtualização: VMWare (VMWARE, 2009), responsável por promover a virtualização de computadores pessoais e servidores; a máquina Virtual Java, que executa sobre um sistema operacional (plataforma baseada em *hardware*) e é responsável pela interpretação do programa Java (DEITEL, 2003); e o .NET Framework, ambiente de execução e desenvolvimento que permite diferentes linguagens, incluindo C#, e bibliotecas trabalharem juntas (MICROSOFT, 2009).

2.2 TECNOLOGIA ADAPTATIVA

A tecnologia adaptativa é resultado da busca de formulações que sejam tanto simples de representar quanto expressivas para lidar com determinados fenômenos (PISTORI, 2003), como o tratamento da dependência de contexto das linguagens de programação (problema que motivou a elaboração do autômato adaptativo (NETO, 1993)). Esta tecnologia trabalha com a aplicação de dispositivos (NETO, 2007) que podem alterar dinamicamente seu comportamento, por meio de auto-modificação em resposta a um estimulo externo (PEDRAZZI; TCHEMRA; ROCHA, 2005) (NETO, 1993).

Diversos formalismos clássicos, como autômatos finitos, gramáticas, cadeias de Markov, *Statecharts*, tabelas de decisão e árvores de decisão tiveram sua expressividade aumentadas quando estendidas com a capacidade de auto-modificação, gerando versões adaptativas desses dispositivos (PEDRAZZI; TCHEMRA; ROCHA, 2005).

As versões adaptativas destes dispositivos estão sendo usadas com sucesso em diversos campos de aplicação como: navegação robótica, composição musical automatizada, visão computacional, reconhecimento de padrões, processamento de linguagens naturais e construção de compiladores (NETO, 2007). Recentemente, esta tecnologia também vem sendo empregada para o desenvolvimento de programas de computadores com a capacidade de auto-modificação (NETO, 2007).

Ainda nessa linha de pesquisa, pode-se citar trabalhos desenvolvidos fora do Laboratório de Linguagens e Técnicas Adaptativas, como as gramáticas recursivas adaptativas (SHUTT, 1993) e os autômatos finitos auto-modificáveis (RUBINSTEIN; SHUTT, 1995).

2.2.1 Dispositivo Adaptativo de Propósito Geral

Em (NETO, 2001), apresenta-se uma proposta geral para a especificação de dispositivos adaptativos baseados em regras. Essa publicação propõe um modelo

geral de dispositivo adaptativo, que pode ser instanciado para a obtenção de quaisquer dispositivos adaptativos específicos.

Segundo a proposta, a descrição de um dispositivo adaptativo pode ser dividida conceitualmente na descrição de um dispositivo subjacente e de uma camada adaptativa.

O dispositivo subjacente representa um dispositivo baseado em regras sem a capacidade de auto-modificação (NETO, 2001), o qual será estendido com características de auto-modificação. Pode-se citar como exemplo o autômato de pilha estruturado, que é o dispositivo subjacente para o autômato adaptativo.

Apesar de tipicamente os dispositivos adaptativos possuírem como dispositivos subjacentes formalismos clássicos e não-adaptativos (PISTORI, 2003) (PEDRAZZI; TCHEMRA; ROCHA, 2005), é possível que um dispositivo adaptativo possa ser usado como dispositivo subjacente, contanto que a capacidade de auto-modificação adicionada não seja sobre um elemento que já possa ser modificado (é inócuo adicionar o suporte à auto-modificação de um elemento que já pode se auto-modificar).

A camada adaptativa, por sua vez, é acrescida ao dispositivo subjacente, sendo responsável por prover a capacidade de auto-modificação (nessa camada descreve-se o mecanismo de auto-modificação do dispositivo subjacente). Conceitualmente, a camada adaptativa é formada por ações adaptativas, as quais mapeiam o conjunto corrente de regras do dispositivo subjacente em outro conjunto de regras, por meio da adição ou remoção destas (NETO, 2001). Em termos práticos, as ações adaptativas são implementadas por meio de chamadas de funções adaptativas paramétricas (NETO, 2001).

Esse modelo procura desvincular a notação do dispositivo não-adaptativo utilizado (subjacente) e o da camada adaptativa, o que facilita a proposição e a compreensão de dispositivos adaptativos (principalmente nos casos em que o formalismo subjacente é conhecido pelo leitor ou pelo autor).

Não obstante, este modelo busca preservar as propriedades do dispositivo não-adaptativo e evita notações complexas, reaproveitando a notação do dispositivo original (NETO, 2001) (PISTORI, 2003). Tal característica possibilita aproveitar diversas das propriedades provadas dos formalismos existentes, evitando redefinir

ou especificar todos os pormenores dos dispositivos adaptativos cujo comportamento se assemelha a algum outro dispositivo existente.

Para finalizar a apresentação do dispositivo adaptativo de propósito geral, segue a definição de elementos básicos de alteração e função adaptativa, descritos nos itens a seguir (NETO, 2001) (NETO; PARIENTE, 2003) (FREITAS, 2008) (PISTORI, 2003).

2.2.1.1 Elementos básicos de alteração

Define-se como elemento(s) básico(s) de alteração o(s) elemento(s) referente(s) ao dispositivo subjacente que pode(m) ser diretamente modificado(s) pela camada adaptativa (especificamente, estes elementos podem ser modificados pelas ações adaptativas elementares). Exemplo: o elemento básico de alteração do dispositivo adaptativo baseado em regras é a própria regra (PISTORI, 2003).

Para ilustrar essa definição, é analisado o caso do autômato adaptativo (NETO, 1994) (NETO; PARIENTE, 2003). O elemento básico de alteração deste formalismo adaptativo é a transição, dado que a camada adaptativa deste formalismo foi projetada para modificar as transições do autômato de pilha estruturado (dispositivo subjacente), por meio de ações de inserção, remoção e consulta às transições (NETO; PARIENTE, 2003).

Elementos do dispositivo subjacente que sofrem modificação de forma indireta não são considerados elementos básicos de alteração. Define-se que a modificação é de forma indireta quanto o elemento pode ser modificado de forma a atender uma ação de automodificação sobre outro elemento, mas não pode ser modificado em outros casos. Seguindo o modelo de autômatos adaptativos descrito em (NETO; PARIENTE, 2003), as ações adaptativas não foram projetadas para modificar os estados (i.e. modificar o conjunto de estados), mas estes podem ser alterados conforme a necessidade de criar ou remover uma transição. Como não existe uma ação de criação ou remoção de estados, a modificação do mesmo é uma modificação de forma indireta. Portanto, o estado não é um elemento básico de alteração.

2.2.1.2 Função Adaptativa e Ação elementar adaptativa

O termo função adaptativa designa os procedimentos responsáveis por implementar as ações de modificação das regras de um dispositivo adaptativo. As ações adaptativas, termo que se refere às operações de auto-modificação de um dispositivo adaptativo, são definidas como sendo chamadas ou ativações das funções adaptativas (NETO, 2001).

Em contrapartida a essa definição genérica, em trabalhos cujo foco é a proposição de um dispositivo adaptativo específico, o termo função adaptativa pode agregar características peculiares relacionadas ao dispositivo proposto. Como exemplo de caso, para o autômato adaptativo, uma função adaptativa pode ser definida pela 9-upla apresentada em (NETO, 1994). Outro exemplo pode ser encontrado no capítulo 6 de (PISTORI, 2003), que propõe algumas alterações e refinamentos na definição das funções adaptativas (relacionadas a autômatos adaptativos) para que a execução de ações elementares de consulta possam ser tratada como um problema de satisfação seqüencial de restrições.

Define-se que um procedimento é uma função adaptativa se for responsável por implementar uma ou mais ações adaptativas e atenda, obrigatoriamente, às seguintes características (NETO, 2001):

- O procedimento responsável pela auto-modificação do dispositivo é baseado em funções (no sentido de funções de linguagem de programação), as quais podem ser paramétricas. Estas podem ser declaradas à parte ou como parte do dispositivo;
- Os procedimentos modificam os elementos básicos de alteração, especificados na camada adaptativa;
- Para realizar a modificação, estas funções são compostas (NETO, 2001) (NETO; PARIENTE, 2003): (i) por ações que atuam como lógica de suporte e, portanto, não estão diretamente relacionadas e nem podem modificar o dispositivo subjacente (exemplos de lógica de suporte: tratamento de geradores e

contadores); (ii) por chamadas a outras funções adaptativas; e/ou (iii) por coleções de ações adaptativas elementares.

As ações adaptativas elementares especificam as possíveis modificações que podem ser aplicadas sobre o dispositivo adaptativo. Com o intuito de organizar a descrição da camada adaptativa, para cada elemento básico de alteração definido, existem três ações adaptativas elementares (análogo às definidas para autômato adaptativo (NETO, 1994) (NETO; PARIENTE, 2003)). São elas:

- Ação adaptativa elementar de consulta: responsável por realizar buscas sobre o conjunto de um determinado elemento básico de alteração, retornando como resultado os elementos que sigam a algum padrão de busca (i.e. critério da busca). Como exemplo de ação elementar de consulta, pode-se citar, para o caso do elemento básico de alteração ser a regra (dispositivo adaptativo baseado em regras), a busca a uma regra que atenda a um determinado padrão (NETO, 2001).
- Ação adaptativa elementar de remoção: responsável pela operação de eliminação de um determinado elemento básico de alteração sobre o conjunto desse elemento. A eliminação de uma determinada regra do conjunto de regras que define o dispositivo adaptativo baseado em regras é uma ação adaptativa elementar de remoção (NETO, 2001);
- Ação adaptativa elementar de inserção: responsável pela operação de inserção no conjunto de um elemento básico de alteração. A adição de uma determinada regra ao conjunto de regras que define o dispositivo adaptativo baseado em regras é uma ação adaptativa elementar de inserção (NETO, 2001);

2.2.2 Autômatos Adaptativos

O autômato adaptativo (NETO, 1993, 1994) (NETO; PARIENTE, 2003) (PISTORI, 2003) foi o primeiro formalismo adaptativo publicado no Laboratório de Linguagens e Técnicas Adaptativas. Sua origem remete à busca de um mecanismo de análise

sintática simples capaz de lidar com a complexidade das linguagens de programação, especialmente em seus aspectos dependentes de contexto (NETO, 1993).

Seguindo a mesma linha de definição de autômato finito adaptativo apresentado em (ROCHA; NETO, 2000), um autômato adaptativo M pode ser visto, inicialmente, como um autômato de pilha estruturado M_0. Devido à capacidade de auto-modificação, a computação de uma determinada cadeia $w_0w_1\ldots w_n$ pelo autômato adaptativo M descreve uma seqüência do tipo $(M_0,w_0) \vdash (M_1,w_1) \vdash (M_2,w_2) \vdash^* (M_n,w_n)$, onde o par (M_i, w_i) – com i de 0 a n – representa a situação de um passo da computação do autômato. Para este par, M_i descreve o autômato de pilha estruturado associado ao passo i da computação e w_i corresponde ao símbolo da cadeia de entrada a ser utilizado no passo. Após cada passo da computação desta máquina têm-se: (i) o mesmo autômato de pilha estruturado (M_{i+1} igual a M_i), caso nenhuma função adaptativa seja executada; ou (ii) um autômato de pilha estruturado novo (M_{i+1} derivado de M_i), caso seja executada uma função adaptativa (NETO, 1994) (ROCHA; NETO, 2000). Fundamentalmente, um autômato adaptativo (que também pode ser chamado de autômato de pilha estruturado adaptativo) é um autômato de pilha estruturado (NETO; MAGALHÃES, 1981) com capacidade de auto-modificação.

Nesse caso, o elemento básico de alteração é a transição. O mecanismo básico de auto-modificação do autômato adaptativo são as ações elementares adaptativas, que são utilizadas para formar as funções adaptativas. Estas funções tipicamente são funções paramétricas declaradas separadamente do autômato, via notação própria (NETO; PARIENTE, 2003).

Chamadas de funções adaptativas denominam-se ações adaptativas, e podem ser associadas às transições do autômato. Por questões de organização, apenas as transições internas e, eventualmente, as transições vazias podem possuir funções adaptativas associadas (NETO; PARIENTE, 2003). Uma ação adaptativa pode ser ativada antes (ações adaptativas tipo B) e/ou depois (ações adaptativas tipo A) da execução de uma transição.

Está provado que utilizar apenas um tipo de ativação de ações adaptativas (antes ou depois da execução da transição) não reduz o poder de expressão deste dispositivo (PISTORI, 2003) (NETO; PARIENTE, 2003) (IWAI, 2000). Como não existe perda de

expressividade, não são raros os casos de autômatos adaptativos que utilizam apenas um dos dois tipos de chamadas.

Um ponto particularmente interessante dos autômatos adaptativos consiste no tratamento do conjunto de estados. Uma das formalizações da alteração indireta dos estados é por meio de uma ligeira modificação do conceito de conjunto de estados. Adota-se a hipótese de infinitos estados preexistentes e se trabalha somente com o conjunto dos estados referenciáveis (PISTORI, 2003).

Em implementações, com certa freqüência a adoção de um conjunto infinito de estados é substituído por criação e remoção implícitas de estados (utilização de geradores). Considera-se que um estado é criado quando ao conjunto de transições é adicionada uma transição com origem ou destino a um estado não-referenciado anteriormente (PISTORI, 2003). De maneira semelhante, o processo de remoção de um estado é representado por meio da eliminação de todas as transições que referenciem um determinado estado. Ressalta-se que, como a criação e remoção de estados não são atividades definidas nas ações elementares adaptativas (sendo que a remoção de estados não é definida nos modelos), este não é um elemento básico de alteração.

A introdução do processo de auto-modificação no autômato de pilha estruturado modificou o poder de expressão do formalismo. Está provado que autômatos adaptativos possuem o mesmo poder de expressão da máquina de Turing (NETO; PARIENTE, 2003) (ROCHA; NETO, 2000).

2.3 CONCEITUALIZAÇÃO DE PROGRAMAÇÃO E PROGRAMAÇÃO ADAPTATIVA

O desenvolvimento da tecnologia adaptativa, bem como outras linhas de pesquisa relacionadas a dispositivos capazes de se auto-modificar, estão definindo uma nova linha de programação, denominada por Freitas (FREITAS, 2008) como programação adaptativa.

O objetivo da programação adaptativa é fornecer um novo estilo de programação, a qual permita o programador escrever códigos adaptativos (códigos com capacidade de se auto-modificarem em tempo de execução segundo os preceitos da tecnologia

adaptativa), permitindo assim a escrita do que é definido como algoritmos adaptativos (algoritmos que, se visualizados como uma seqüência de regras a serem executadas, possuem regras que modificam outras regras).

Visando formar a base conceitual, esse tópico procura definir os conceitos básicos associados à programação adaptativa e utilizados ao longo desse texto, tais como programa, código, código auto-modificável, programação adaptativa e código adaptativo.

2.3.1 Programas e códigos

Em (PRESSMAN, 2002), define-se *software* como um produto que abrange: programas a serem executados em computadores, documentos e dados. Reestruturando a definição apresentada, pode-se afirmar que um *software* é um produto formado por uma documentação e de um ou mais programas de computador.

Um programa de computador consiste na representação, utilizando alguma linguagem de programação, de um ou mais algoritmos (função computável) especificados por um programador. Ao analisar a escrita de programas em diversas linguagens e tendo em mente os conceitos da teoria de compilação descritos em (AHO et al., 2006) e (NETO, 1987), pode-se afirmar que um programa pode ser logicamente decomposto em código fonte e dados.

A divisão lógica de programa em código fonte e dados nem sempre é bem definida, como em casos em que uma variável armazena comandos a serem interpretados, bem como pode não ser adequada para alguns paradigmas de programação. No entanto, essa divisão lógica é adotada com o intuito de estruturar esse trabalho.

O código fonte é constituído da seqüência de instruções, de uma determinada linguagem de programação ou de uma máquina, que compõem o programa. Estabelecendo um paralelo com o trabalho publicado em (NETO, 2001), pode-se considerar as instruções de uma linguagem como sendo as regras que podem ser utilizadas para a declaração de um dispositivo dirigido por regras. Seguindo essa

analogia, o código corresponderá ao conjunto de regras que define o dispositivo (que no caso é o programa).

Já os dados são representações de informações associadas ao programa e manipuladas pelo código. É o conjunto posições de memória, variáveis e/ou estruturas de armazenamento similares (exemplo: *structs* da linguagem C) cujas informações são manipuladas pelo programa. Apesar de serem importantes para a execução do programa (podendo ser, por exemplo, o fator responsável pela decisão de desviar ou não o fluxo da execução), os dados constituem-se de informações que, via de regra, não são instruções a serem executadas (i.e. tradicionalmente não correspondem a seqüência de instruções que definem o código). Pode-se citar, como exceção a essa regra, o caso em que o dado é passado para algum comando de interpretação, como o *eval* do LISP (STEELE, 1990), onde é tratado como um trecho de código.

Para ilustrar essa distinção, considere-se o seguinte exemplo: um programa que faz a soma da variável inteira `a` com a `b`, sendo o resultado armazenado em `c`. O código consiste nos comandos da linguagem de programação que foram utilizados para descrever o algoritmo (uma dessas regras é a instrução de soma, que na linguagem de programação C (MIZRAHI, 1990) seria escrita como `c = a + b;`). Por fim, temos que `a`, `b` e `c` nomeiam os dados utilizados pelo programa.

2.3.2 Códigos auto-modificáveis e Códigos adaptativos

Define-se como código auto-modificável (em inglês, *self-modifying code*) o código que modifica as suas próprias instruções, intencionalmente ou não, ao longo de sua execução (HONGXU; ZHONG; ALEXANDER, 2007). Trabalhos sobre códigos adaptativos, bem como exemplos desse tipo de código podem ser encontrados em (MASSALIN, 1992) (GIFFIN; CHRISTODORESCU; KRUGER, 2005) (ANCKAERT; MADOU; BOSSCHERE, 2001).

Define-se que códigos adaptativos são uma subclasse de códigos auto-modificáveis, que atendem aos seguintes critérios:

- Um código adaptativo pode modificar-se segundo o mecanismo definido pela camada adaptativa associada a este código (que deve ser baseada nos conceitos da Tecnologia Adaptativa). Como conseqüência desse critério, define-se que se um determinado código adaptativo CA_0 modificar-se para um código CA_1 e a modificação realizada não se encontra definida na camada adaptativa associada, então o código CA_0 não é um código adaptativo (esse tipo de situação pode ocorrer caso uma determinada instrução da linguagem, que não seja contemplada pela camada adaptativa, possa modificar o código).
- O mecanismo responsável pela modificação do código são as funções adaptativas. Estas funções podem:
    - Ser declaradas no próprio código. Para tanto, as ações adaptativas elementares devem fazer parte do conjunto de instruções disponíveis na linguagem;
    - Ser previamente especificadas junto ao ambiente de execução da linguagem, sendo estas invocadas por meio de instruções ou ativações de procedimentos previamente definidos. Neste caso, o projetista da linguagem define o conjunto de funções adaptativas, não sendo possível criar novas funções adaptativas sem a modificação da linguagem.
- As ativações das funções adaptativas devem ser feitas no próprio código a ser modificado. Por exemplo, a função adaptativa pode ser uma instrução da linguagem e sua ativação consiste na utilização dessa instrução no código (de maneira análoga como a ativação das funções adaptativas no autômato são realizadas por meio da associação das chamadas às transições).
- As modificações do código devem garantir a integridade sintática do mesmo a cada alteração. Se L é o conjunto infinito de todos os códigos sintaticamente corretos da linguagem utilizada, CA_0 é um código adaptativo escrito nesta mesma linguagem e K é o conjunto finito ou infinito de todos os códigos que podem ser obtidos por meio da auto-modicação a partir de um CA_0, define-se que CA_0 é um código que garante a integridade sintática se, e somente se, K intersecção com L for igual a K (ou seja, o conjunto K está contido ou é igual ao conjunto L). Como caso excepcional a este requisito, admite-se que CA_0 seja um código que garante a integridade caso, durante ou logo após a execução de qualquer ação

adaptativa que introduza erros sintáticos no código, a execução do mesmo seja interrompida.

Formalmente, códigos adaptativos foram definidos em (FREITAS; NETO, 2005, 2006, 2007) como códigos que se modificam por meio das funções adaptativas (seguindo o modelo descrito em (NETO, 1994, 2001) (NETO; PARIENTE, 2003)). Um código escrito em uma linguagem para programação adaptativa qualquer consiste inicialmente no código-fonte CF_0. Conforme se executam as funções adaptativas, esse código vai se modificando para CF_1, CF_2, ..., CF_n (FREITAS; NETO, 2005, 2006).

2.3.2.1 Modelos de execução para códigos adaptativos

Este tópico apresenta três modelos de execução para códigos adaptativos. Estes procedimentos foram definidos baseados nos trabalhos (FREITAS, 2008) e (NETO, 2001).

O primeiro modelo consiste na programação adaptativa por meio de bibliotecas. Neste esquema, todo o programa é escrito em uma linguagem para programação não-adaptativa, sendo desenvolvida uma biblioteca responsável por prover o recurso de modificação do código segundo o paradigma adaptativo (ilustrado na Figura 4). Esse modelo é um dos que mais se assemelham à idéia de estender o dispositivo não-adaptativo para formar um dispositivo adaptativo, definido em (NETO, 2001).

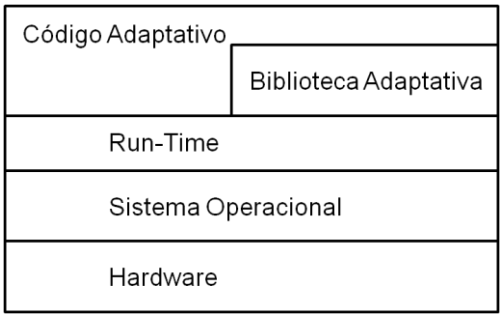


Figura 4 – Criação de um código adaptativo por meio do uso de biblioteca adaptativa em uma linguagem para programação não-adaptativa

O segundo consiste em estender uma linguagem de programação não-adaptativa com o mecanismo adaptativo, formando assim uma linguagem para programação adaptativa com a qual é possível escrever os códigos adaptativos.
Neste modelo, o ambiente de execução (*run-time*) da linguagem é estendido com rotinas adaptativas, as quais são responsáveis por implementar a camada adaptativa associada a linguagem (conforme ilustrado na Figura 5).

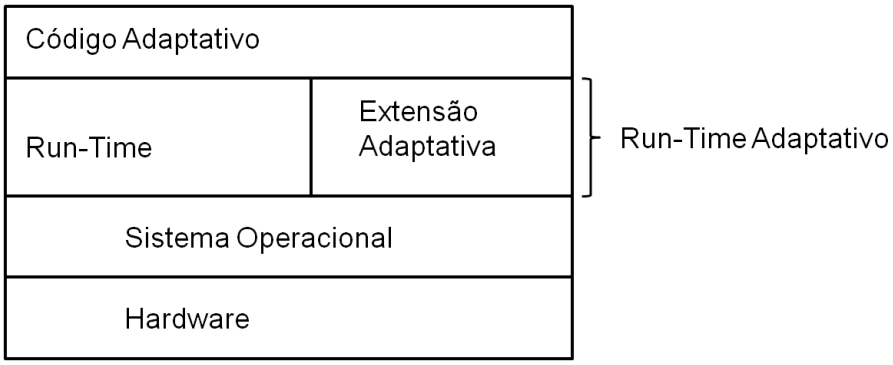


Figura 5 – Criação de código adaptativo em uma linguagem estendida com recursos adaptativos.

Por fim, pode-se propor um ambiente de execução com suporte nativo à adaptatividade, o qual pode ser construído sobre um *hardware* convencional (não-adaptativo) ou adaptativo (com suporte nativo à adaptatividade), como ilustrado na Figura 6. Este caso é similar ao anterior, com a exceção de que o modelo do ambiente de execução é, no máximo, baseado em alguma proposta de modelo existente (portanto, esse caso não considera extensões de modelos existentes).

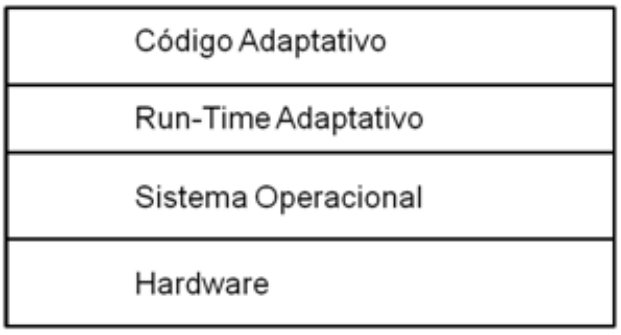


Figura 6 – Criação de código adaptativo em uma linguagem nativamente adaptativa.

2.3.3 Programas Adaptativos

Programas podem ser vistos como um caso particular de dispositivos dirigidos por regras, em que as regras correspondem à descrição do código do programa. Seguindo a definição anterior, programas adaptativos podem ser vistos como dispositivos dirigidos por regras adaptativos, no qual o código (conjunto de regras) pode se auto-modificar ao longo da execução.

Resumindo, define-se que um programa adaptativo é um programa cujo código é adaptativo.

2.4 LINGUAGENS DE PROGRAMAÇÃO E ADAPTATIVIDADE

Na teoria de computação, define-se linguagem como um conjunto de cadeias sobre um determinado alfabeto (LEWIS; PAPADIMITRIOU, 1981) (NETO, 1987).

Caso esse conjunto seja finito, a linguagem pode ser especificada por meio da enumeração desse conjunto. No entanto, a enumeração das cadeias não é a única maneira de descrever uma linguagem.

Linguagens também podem ser declaradas por meio da enumeração de suas propriedades, seja por meio da definição de leis de formação destinadas a gerar as cadeias pertencentes à linguagem (geração da linguagem por meio de gramáticas) ou por meio da definição de leis capazes de verificar se a cadeia pertence ou não à linguagem (verificar se um texto pertence ou não a uma linguagem por meio de um reconhecedor, como máquina de Turing e autômatos) (LEWIS; PAPADIMITRIOU, 1981).

Partindo-se dessa definição, esse tópico procura definir os conceitos de linguagens de programação para, em seguida, conceitualizar linguagens para programação adaptativa.

2.4.1 Linguagens de Programação

Linguagens de programação são notações para descrever funções computáveis (AHO et al., 2006). Assim como as linguagens naturais constituem uma das formas mais importantes de expressar algum pensamento (FREITAS, 2008), as linguagens de programação representam a forma de descrever um programa para o computador e, eventualmente, para outras pessoas que conheçam a linguagem de programação em questão.

Existem duas maneiras usuais de formalizar uma linguagem de programação. A primeira consiste no uso de dispositivos reconhecedores, os quais avaliam se um determinado texto (no caso, um programa) pertence ou não à linguagem especificada. Neste caso a linguagem é formada por todos os programas que o reconhecedor verifique que pertença a mesma.

A segunda maneira consiste na especificação de leis de formação para a construção dos programas. A partir dessas leis de formação, denominada de gramática, pode-se gerar qualquer programa válido para essa linguagem. Neste caso a linguagem é formada por todos os códigos gerados pela gramática.

As linguagens de programação diferem da definição de linguagem apresentada anteriormente devido ao fato de que programas escritos nas linguagens de programação possuem algum significado semântico, o que não é necessariamente verdade para todas as linguagens. Não obstante, nem todas as linguagens que possuem semântica associada são linguagens de programação.

Os programas gerados por uma determinada linguagem de programação podem ser compilados (i.e. convertidos) para outras linguagens (como a de máquina) nas quais serão executadas – comumente programas escritos na linguagem C são compilados antes de serem executados (MIZRAHI, 1990) –, ou podem ser interpretadas por algum ambiente de execução (AHO et al., 2006) – programas escritos na linguagem Java costumam ser interpretados na máquina Virtual Java (DEITEL, 2003).

O resultado de sua execução é função do significado semântico atribuído ao programado escrito.

2.4.2 Linguagens para Programação Adaptativa

As linguagens para programação adaptativa são aquelas que, independentemente do paradigma de programação associado, fornecem aos programadores meios para a modificação em tempo de execução dos códigos escritos nestas (ACAR; BLELLOCH; HARPER, 2006) (FREITAS; NETO, 2005, 2006, 2007).

Algumas propostas de linguagens ou trabalhos correlacionados a este assunto são apresentadas em (ACAR; BLELLOCH; HARPER, 2006), (ANCKAERT; MADOU; BOSSCHERE, 2001), (LIEBERHERR; ORLEANS; OVLINGER, 2001), (LIEBERHERR, 1996), (LIU; PRADHAN, 1996), (YODER; JOHNSON, 2007) e (ENGLER; HSIEH; KAASHOEK, 1996).

Dentro do contexto da tecnologia adaptativa, foram denominadas inicialmente de linguagens adaptativas (FREITAS; NETO, 2005, 2006). Devido ao fato do termo linguagem adaptativa poder ser interpretado como linguagem capaz de se auto-modificar, a denominação foi alterada para linguagem para programação adaptativa. Devido a sua própria definição, uma linguagem não é capaz de se auto-modificar.

Ao proporem a primeira linguagem para programação adaptativa dentro desse contexto, Freitas e Neto, em (FREITAS; NETO, 2005, 2006), procuraram: (i) seguir o esquema de extensão apresentado em (NETO, 2001), acrescentando uma camada adaptativa ao ambiente de execução de códigos de uma linguagem de programação existente. Não obstante, a descrição do ambiente de execução desta linguagem é fundamentada na própria linguagem a ser estendida (FREITAS; NETO, 2005, 2006); e (ii) respeitar e incorporar as características da camada adaptativa, tal como originalmente definidas para o dispositivo adaptativo de propósito geral, ao se propor a camada adaptativa associada ao código adaptativo escrito nesta linguagem (FREITAS; NETO, 2005, 2006).

A formalização de linguagens para programação adaptativa foi elaborada por Freitas e Neto (FREITAS; NETO, 2005, 2006, 2007), sendo definidas como linguagens de programação que possibilitam a escrita de códigos que se modificam por meio das funções adaptativas (seguindo o modelo descrito em (NETO, 1994, 2001) (NETO; PARIENTE, 2003), ou seja, que permitam a escrita de códigos adaptativos.

Apesar das linguagens para programação adaptativa, no contexto da tecnologia adaptativa, serem linguagens de programação que fornecem ao programador algum recurso que permita a escrita de códigos adaptativos, estas não são um dispositivo adaptativo. As linguagens para programação adaptativa são o meio ou a notação no qual o programador especifica um dispositivo adaptativo (programa adaptativo).

Pela mesma razão, uma das formas de se especificar uma classe de dispositivos adaptativos do tipo programa adaptativo é descrever a linguagem de programação adaptativa. Essa linguagem de programação adaptativa é capaz de gerar todos os possíveis programas adaptativos que podem ser escritos e executados no ambiente de execução associado a essa linguagem.

Como freqüentemente um dispositivo adaptativo é logicamente decomposto em dispositivo subjacente e camada adaptativa, é possível se aproveitar de tal esquema e especificar uma linguagem para programação adaptativa por meio da descrição da linguagem de programação sem os recursos de modificação de código (metadescrição do dispositivo subjacente) e, em seguida, apresentar a camada adaptativa associada ao programa adaptativo.

3 CÓDIGOS ADAPTATIVOS E LINGUAGEM ADAPTCODE

A linguagem Adaptcode é uma linguagem a programação adaptativa que possui sua origem baseada na linguagem de montagem associada à arquitetura Intel® 32 bits (INTEL, 2007) (INTEL N-Z, 2007) a qual será utilizada como notação para a descrição dos códigos adaptativos.

A idéia que rege a proposta desta linguagem é fornecer ao programador, que deseja escrever códigos adaptativos de baixo nível, um conjunto de instruções que permita e facilite especificar a modificação do código durante a codificação, bem como durante a sua execução, efetue as modificações planejadas, delegando ao ambiente de execução a responsabilidade do processo de modificação em si. O programador, em relação ao processo de modificação do código, deve planejar que instruções serão removidas e quais instruções serão adicionadas.

Não obstante, tais instruções devem garantir a integridade sintática do código, evitando casos como a inserção de uma instrução mal formatada (um exemplo de tal instrução é uma instrução com o número incorreto de operandos, portanto, um exemplo de situação que não deve ocorrer é permitir a inserção de uma instrução sem seus devidos operandos). A busca em garantir a integridade sintática do código é motivada em diminuir os erros associados ao uso do recurso de modificação do código ao programador (evitando casos com o anteriormente mencionado), bem como manter aderência com os demais dispositivos adaptativos desenvolvidos (as ações adaptativas não devem transformar um dispositivo adaptativo em outro dispositivo adaptativo).

Para atender os requisitos citados, a alteração de código, para a linguagem proposta, é planejada por meio de instruções que delimitem trechos de códigos passíveis de alteração (composto por um número inteiro, maior ou igual a zero, de instruções em seqüência, que no modelo de execução clássico ocupam posições consecutivas de memória). Adicionalmente a estas, existe outra instrução responsável pela alteração de tais trechos, substituindo o código existente neste trecho por outro, cuja imagem esteja armazenada na área de dados.

A proposta descrita é constituída de uma formulação teórica, no qual são definidos os mecanismos adaptativos responsáveis pela modificação de código, e de uma

especificação protótipo, derivado de tal modelo, implementada de forma a testar os mecanismos descritos.

Destaca-se, por fim, que a proposta presente possui certo nível de independência em relação à linguagem utilizada como base, ou seja, pode ser transposta para máquinas (nativas ou virtuais) cujo conjunto de instrução seja diferente, desde que se mantenham certas similaridades, especialmente em relação às instruções de controle de fluxo (fato que é comum em máquinas que utilizam a mesma arquitetura).

3.1 DESCRIÇÃO DA LINGUAGEM ADAPTCODE

Ao se aproveitar de uma linguagem existente para a proposição da linguagem AdaptCode, procura-se reduzir a complexidade da proposta, por meio da preservação de parte das propriedades e características existentes nas linguagens de montagem. Não obstante, também se reaproveita as instruções, sintaxes e a forma de escrita de códigos existentes, simplificando a compreensão do código fonte escrito.

Entretanto, diferentemente da proposta publicada em (FREITAS, 2008), onde o mecanismo adaptativo é construído sobre o ambiente da linguagem a ser utilizada, como uma biblioteca, o ambiente de execução da linguagem AdaptCode é modificado de tal forma a suportar a operação de modificação de código. Portanto, devido a características do ambiente de execução, esta linguagem sem os recursos de auto-modificação não é idêntica a linguagem em que se baseia (devido a limitações do modelo proposto).

Para descrever a linguagem AdaptCode, primeiramente é descrita a linguagem origem sem os recursos que permitem a escrita de códigos adaptativos. Em seguida, são descritos os mecanismos adaptativos associados aos códigos escritos na linguagem.

3.1.1 Linguagem Origem: Linguagem de Montagem

A linguagem AdaptCode é formado por um conjunto de instruções associada à arquitetura Intel® 32 bits (INTEL, 2007) (INTEL N-Z, 2007) (MASM32, 2006) (DETMER, 2001). Tal escolha é motivada pelo fato desta ser a arquitetura utilizada em computadores domésticos, bem como utilizar um padrão de escrita de código de baixo nível freqüentemente utilizada por montadores.

A forma de escrita de código é similar à escrita na linguagem de montagem associada à arquitetura Intel® 32 bits, ou seja, o código é descrito por meio de uma seqüência de instruções da linguagem. Utilizando a notação de Wirth (AHO et al., 2006) (NETO, 1987), se `C` representa o código e `I` representa as possíveis instruções da linguagem, então um código dessa linguagem pode ser descrito por meio de `C = I { I }.` (para facilitar a visualização, entre cada instrução é feito uma quebra de linha).

O conjunto de instruções desta linguagem é composto das: instruções de movimentação de dados (`mov`), instrução baseadas em operadores aritméticos (sobre números inteiros de 32 bits) de soma (`add`), subtração (`sub`), multiplicação com sinal (`imul`), divisão com sinal (`idiv`), salto (`jmp`), desvio condicional (série de comandos do tipo `jxx`), instrução de comparação (`cmp`), instrução sem operação (`nop`), instrução de chamada de procedimento (`call`), retorno de procedimento (`ret`), instrução de *push* na pilha de sistema (`push`) e *pop* na pilha de sistema (`pop`) (INTEL, 2007) (INTEL N-Z, 2007). Entretanto, algumas ressalvas são feitas:

- Não é permitida a instrução de movimentação de dados (`mov`) acessar a área de código do programa (o que evita a possibilidade de outros tipos de modificação no código, salvo os especificados pela camada adaptativa);
- A instrução de chamada de procedimento (instrução `call`) deve possuir como operando apenas os *labels* associados aos procedimentos (ou seja, podem apenas chamar os procedimentos declarados). Explicações relacionadas a esta restrição são apresentas no item 4.2.1.1.

- Existe uma instrução `label` responsável por declarar os *labels* utilizados nas instruções de saltos e desvios. Em tempo de execução, essa instrução serve apenas como suporte à execução (associação entre o estado e o nome do *label* que designa o estado).
- Os *labels* associados às rotinas não são declarados por instrução, e no código são diferenciados dos *labels* relacionados aos saltos e desvios por virem acompanhados dos identificadores `proc` e `endproc`.
- As instruções de saltos e desvios devem possuir como operando apenas os *labels* criados via instrução `label`.

Uma vez que o programador não tem controle do processo de alteração, fica difícil controlar a associação endereço/instrução, portanto, a linguagem não suporta a utilização de endereços físicos nas instruções que desviam o fluxo da execução, como as instruções `call`, `ret`, `jmp` e `jne`. Para esse fim são utilizados os *labels*, que indiretamente indicam os endereços associados aos desvios de fluxo.
Apesar de o subconjunto utilizado possuir um número reduzido de instruções, este conjunto permite a criação de diversos programas. Ressalta-se também que, da forma como o ambiente de execução foi planejado, instruções que não estão associadas ao fluxo de controle (HENNESSY; PATTERSON, 2003) (como instrução de soma, movimentação de dados) podem ser facilmente incorporadas ao modelo.
O conjunto completo de instruções da arquitetura Intel® 32 bits não é implementado por não fornecer contribuição a este trabalho.

3.1.2 Mecanismo de modificação de código da linguagem AdaptCode

A descrição dos recursos dedicados a modificação desta linguagem é composta pela descrição do elemento básico de alteração do código adaptativo que a linguagem permite escrever, que consiste no trecho de código adaptativo (definido no item 3.1.2.1), e da função adaptativa (definido no item 3.1.2.3).

O mecanismo adaptativo pode, resumidamente, ser descrito como a substituição de um trecho de código adaptativo por outro trecho de código, montado a partir de uma imagem existente na área de dados.

3.1.2.1 Elemento básico de alteração: trecho de código adaptativo

A proposta dos códigos adaptativos que podem ser descrito pela linguagem AdaptCode define um único elemento básico de alteração: o trecho de código adaptativo. Define-se como trecho de código adaptativo um trecho de código que compreenda uma seqüência arbitrária (eventualmente vazia) de instruções contidas no código e que atende a determinados requisitos (vide item 3.1.2.2).

Esta escolha é motivada por uma série de fatores. Primeiramente, pela hipótese de que uma alteração planejada de um código tipicamente reflete algum desejo de modificação do comportamento de uma parte do programa. Adicionalmente, este comportamento, em código, é representado por uma seqüência de tamanho variado de instruções justapostas. Portanto, modificar o comportamento de parte do código consiste em alterar a seqüência de instruções que representam tal comportamento. Esta hipótese procura refletir o conceito presente em (ACAR; BLELLOCH; HARPER, 2006), no qual programas adaptativos atualizam suas saídas em função de mudanças na entrada, por meio de modificações em porções do código.

Adicionado a tal fato, têm se que este esquema pode realizar qualquer tipo de modificação no trecho de código demarcado, evitando assim a necessidade de definir mais de um elemento básico de alteração (como, por exemplo, definir comandos para modificar os operadores e outro para modificar os operandos).

Não obstante, ao se trabalhar com trechos que englobem apenas números inteiros de instruções, evitam-se situações problemáticas, como por exemplo, a substituição do código de operação de uma instrução por outro que precisa de um número diferente de operandos, sem a devida correção do número de operandos existentes na memória (exemplo deste tipo de problemática foi citado no item 1.1).

Este esquema aproveita da própria forma de escrita do código para inserir o mecanismo adaptativo, simplificando assim a notação. Devido à forma de escrita de

códigos na linguagem de montagem, o processo de delimitar instruções não justapostas ou de marcar informações internas a escrita de uma instrução dificultaria o entendimento do código. O primeiro caso remete a dificuldade de marcar dois trechos de textos não consecutivos, ilustrado na Figura 7(a), o segundo caso remete ao caso do símbolo de delimitação se sobrepor a escrita da instrução, como ilustrado na Figura 7(b).

```
mark e1
add eax,2
emark e1
cmp eax,ebx
mark e1
sub ebx,1
emark e1
```

(a)

```
add mark e1 eax emark e1 , 2
```

(b)

Figura 7 – Modelos de marcações de códigos desconsiderados. (a) tentando marcar instruções não justapostas (instruções `add eax,2` e `sub ebx,1`). (b) tentando marcar operando `eax` dentro de uma instrução.

Os trechos de códigos adaptativos são definidos pelo programador durante a escrita do código, utilizando instruções que marcam o inicio e o fim de tais trechos. Durante a definição de um trecho de código adaptativo, a este é associado um identificador único, definido pelo programador, por meio da instrução de marcação dos trechos (ou seja, o identificador é um operando da instrução que define o trecho de código adaptativo). O processo de definição do trecho esta descrito no item 3.1.2.2.

A justificativa de adotar um esquema de indexação dos trechos de códigos adaptativos origina-se do fato de simplificar a identificação de tais trechos para o programador, facilitando assim o uso do recurso de modificação de código. Ao utilizar a instrução responsável por modificar o código, o programador necessita apenas conhecer o nome que designa o trecho de código a ser modificado, eliminando, assim, a necessidade de o programador conhecer o conteúdo do trecho (que eventualmente pode ser muito extenso) ou o seu endereço de memória.

Ao permitir que o programador defina o indexador associado a cada trecho, têm-se a possibilidade que o mesmo aplique seu próprio critério ou abordagem de identificação desses trechos. Por exemplo, ao implementar o código de um autômato

adaptativo, pode-se usar uma convenção de nome que segue o padrão Ex para nomear o trecho de código adaptativo que contêm o código associado ao estado x do autômato adaptativo, o que pode facilitar a escrita no código.

Por fim, têm-se a simplificação do processo de busca do elemento básico de alteração (ação adaptativa elementar de consulta). Dado que um código de baixo nível pode possuir milhares de instruções, realizar buscas não indexadas sobre o código implica em uma quantidade de tempo significativa para o processo de busca, devido à complexidade computacional, idêntico ao do problema de achar um *string* ou de um padrão de *string* dentro de um texto.

3.1.2.2 Modelo de marcação dos trechos de códigos adaptativos

O modelo de marcação baseado em blocos foi primeiramente descrito no artigo (PELEGRINI; NETO, 2008). Neste modelo, a marcação de um trecho de código adaptativo é feito por meio de duas instruções: `mark id` e `emark id`, onde o termo `id` refere-se ao indexador do trecho de código adaptativo. A instrução `mark` é utilizada para iniciar um trecho modificável, enquanto a instrução `emark` encerra tal trecho (exemplos de marcação apresentados na Figura 8).

Apesar da delimitação dos trechos de códigos adaptativos ser o principal objetivo das instruções `mark` e `emark`, o ambiente de execução e o montador utilizam estas instruções para outras finalidades, como permitir ao ambiente de execução saber dentro de quais trechos de códigos modificáveis a instrução corrente se encontra.

Este esquema de marcação deve seguir apenas duas regras: (i) dois ou mais trechos de códigos adaptativos podem ser aninhados, mas não podem ser mutuamente sobrepostos (conforme indicado pela Figura 8); (ii) os trechos de códigos adaptativos devem estar contidos dentro de um procedimento (motivo explicado no item 4.3.1), o que torna impossível, teoricamente, remover procedimentos existentes nesta linguagem (adicionando esta propriedade a forma de criação de procedimentos, a linguagem também não permite a criação de procedimentos).

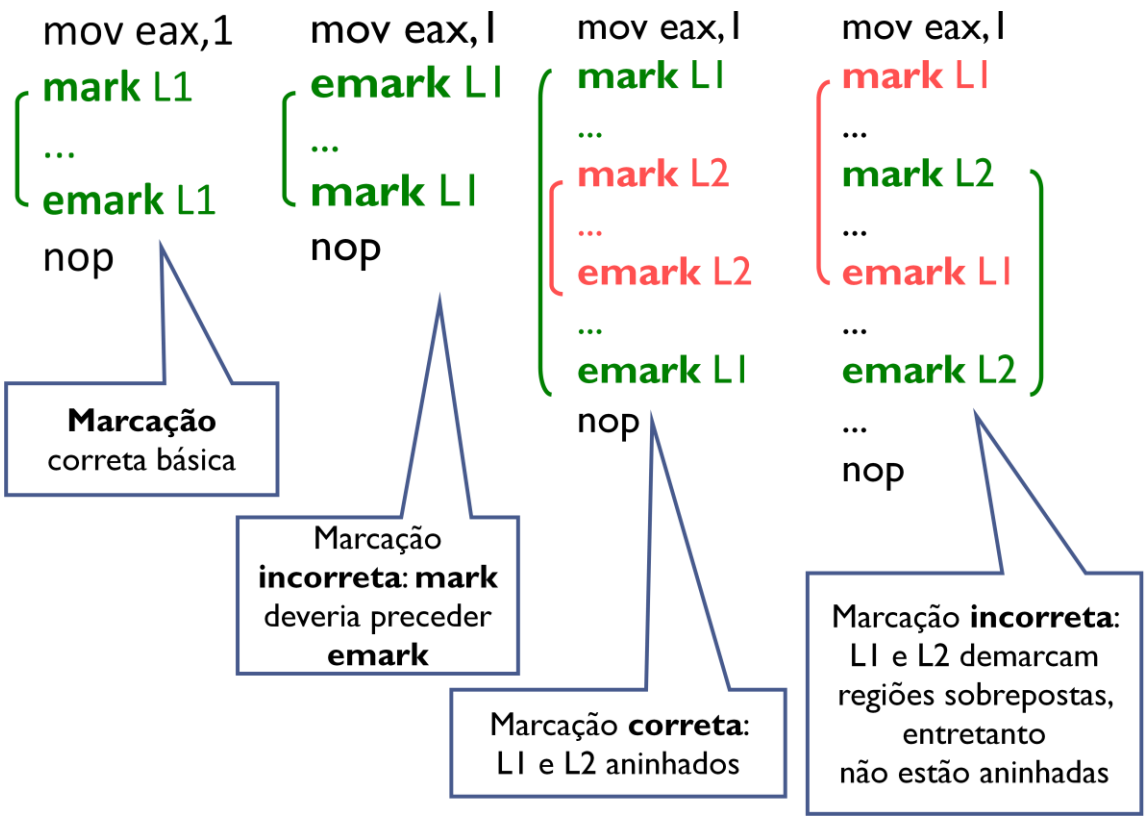


Figura 8 – Exemplos de marcações corretas e incorretas. Retirado da apresentação do artigo (PELEGRINI; NETO, 2008).

3.1.2.3 Mecanismo de alteração: Função Adaptativa

Nesta proposta, a modificação do código, em si, é feita utilizando-se uma única instrução chamada `adapt`. Esta instrução recebe como operandos:

- O indexador de um determinado trecho de código adaptativo (seja por meio direto ou indireto). Especificamente, o operando a ser passado é o indexador do trecho de código adaptativo que será alterado.
- A posição, na área de dados do código, onde se inicia a imagem da seqüência de instruções a ser inserida (caso não se deseje inserir nada, deve-se passar como operando o valor imediato 0 ou o endereço de uma imagem de código que contenha apenas a pseudo-instrução *endp*). Esta imagem consiste de um trecho de código escrito na linguagem AdaptCode que deve, obrigatoriamente, terminar

com a pseudo-instrução `endp`, a qual é responsável por indicar o final físico da imagem a ser inserida.

O processo de modificação é baseado na substituição, ou seja, este processo, primeiramente, remove o trecho de código adaptativo indicado e todos os trechos de códigos adaptativos aninhados dentro deste, destruindo-os (nota: as instruções que marcam este trecho também são removidas neste processo). Em seguida, monta-se a imagem da seqüência de instruções a serem inseridas (que, eventualmente, podem criar novos trechos de códigos adaptativos), inserindo o código montado no lugar do trecho de código adaptativo previamente removido. Ressalta-se que a imagem situada na área de dados não é destruída pelo processo, podendo ser modificada ou reutilizada em outras instâncias.

Para ilustrar tal esquema, considere o código apresentado na Figura 9 e na Figura 10 (rotina fatorial adaptativa). Essas figuras estão formatadas segundo o seguinte esquema: *labels* são identificadas por letras maiúsculas – o *proc* após um *label* identifica que o mesmo indica o inicio de um procedimento, instruções estão em negrito, pseudo-instruções e identificadores auxiliares (`proc` e `endproc`) estão em itálico.

Área de Código	Área de Dados

FAT *proc*
  pop n
  mov eax,1
  cmp n,0
  jle X
  adapt 1,step
  mark 2
  mark 1
  emark 1
  emark 2
label X
  mov res,eax
  adapt 2,base
  ret
FAT *endproc*
	<<dado em step>>
	  mul eax,n
	  dec n
	  cmp n,0
	  jz Y
	  adapt 1,step
	  mark 1
	  emark 1
	label Y
	  endp

Figura 9 - Exemplo de código adaptativo escrito na linguagem AdaptCode antes da execução da instrução `adapt 1,step`.

Área de Código	Área de Dados

FAT *proc*
  pop n
  mov eax,1
  cmp n,0
  jle X
  adapt 1,step
  mark 2
  mul eax,n
  dec n
  cmp n,0
  jz Y'
  adapt 1,step
  mark 1
  emark 1
  label Y'
  emark 2
label X
  mov res,eax
  adapt 2,base
  ret
FAT *endproc*
	<<dado em step>>
	  mul eax,n
	  dec n
	  cmp n,0
	  jz Y
	  adapt 1,step
	  mark 1
	  emark 1
	label Y
	  endp

Figura 10 - Exemplo de código adaptativo escrito na linguagem AdaptCode após a execução da instrução `adapt 1,step`.

Ao executar a instrução `adapt 1,step` no código ilustrado pela Figura 9: (i) as instruções `mark 1` e `emark 1` são apagadas; e (ii) o código existente na área de dados (partindo do endereço `step`) será montado e inserido entre as instruções `mark 2` (instrução que era imediatamente anterior à antiga instrução `mark 1`, apagada) e `emark 2` (instrução que era imediatamente posterior à antiga instrução `emark 1`, apagada), obtendo o código ilustrado pela Figura 10 (uma explicação mais detalhada do funcionamento do processo de modificação por ser encontrada no capítulo 6).

Destacam-se os seguintes pontos deste exemplo:

- Enquanto a instrução `adapt 1,step` remove o trecho marcado, indexado por `1`, também cria um novo trecho marcado, com o mesmo indexador. Esta modificação é permitida porque antes de realizar a montagem do código que se inicia no endereço `step`, o trecho de código adaptativo que utiliza o indexador `1`

é removido (frisando que este trecho é o que está sendo modificado pela instrução `adapt`).

- Durante a montagem do código que começa em `step`, o *label* `Y` (endereço de destino da instrução `jz Y` existente no código a ser inserido) é resolvido em função da declaração do *label* existente no próprio código a ser inserido. Portanto, a cada inserção (montagem), o endereço relacionado ao *label* `Y` fica associado a um endereço diferente (observe que na Figura 10 esta distinção se faz alterando-se `Y` para o nome `Y'`, instância diferente de `Y` que representa diferente endereço resolvido durante a montagem).

A principal diferença entre a montagem da imagem do código situado na memória e a da montagem inicial do código é o tratamento dos *labels*. Ao realizar a montagem da imagem do código para substituição, os *labels* relacionados a saltos são tratados segundo o seguinte esquema:

- Caso alguma instrução de salto ou desvio da imagem utilize um *label* declarado dentro da própria imagem, o *label* a ser utilizado na imagem é resolvido em função desta declaração, sendo para tanto instanciado novos *labels* (como mencionado anteriormente).
- Caso alguma instrução de salto ou desvio da imagem utilize um *label* não declarado dentro da própria imagem, o montador verificará com o ambiente de execução se o *label* em questão encontra-se declarado dentro do procedimento associado ao trecho de código a ser modificado. Se o estiver, o *label* é resolvido em função da declaração existente (diferentemente do caso anterior, esse processo não implica em instanciação). Se não estiver declarado no procedimento, o processo de montagem da imagem é abortado, o que resulta no término da execução do código.

Em relação a este processo de modificação, existem dois casos excepcionais de modificação. O primeiro caso remete a quando a instrução de modificação de código encontra-se contida no trecho de código adaptativo a ser alterado. Neste caso, após a auto-modificação do código, a próxima instrução a ser executada é a imediatamente seguinte ao trecho de código adaptativo alterado. O segundo caso

remete a situação em que a modificação do código remove uma instrução cujo endereço está salvo na pilha (um exemplo desta situação consiste em apagar a instrução na qual a execução deve retornar ao término da execução do procedimento corrente). Neste caso, o endereço da instrução salvo na pilha é corrigido pelo endereço imediatamente seguinte ao trecho de código adaptativo alterado.

No caso de algum erro acontecer durante o processo de modificação, a execução do código será abortada. Dentre exemplos que resultam em erros, pode-se citar: tentativa de inserção de trechos de códigos sintaticamente incorretos ou utilizar como operando da instrução `adapt` um indexador de trecho de código adaptativo inexistente.

O processo de modificação baseado em substituição acima descrito foi escolhido por possibilitar a existência de uma única função adaptativa (a instrução `adapt`), que permite realizar todas as ações adaptativas previstas na camada adaptativa desta linguagem, por meio da modificação dos parâmetros (operandos da instrução). Por definição, a construção da imagem do código a ser inserido na memória não é considerada parte da função adaptativa. A motivação desta escolha reside no fato de não atribuir ao programador a responsabilidade da criação da função adaptativa, restringindo a ele o planejamento das ações adaptativas.

4 AUTÔMATO DE EXECUÇÃO ADAPTATIVO

Similar ao trabalho desenvolvido em (FREITAS, 2008), nessa dissertação procura-se utilizar dispositivos adaptativos existentes para representar um código adaptativo, de tal forma a mapear o processo de auto-modificação do código adaptativo no processo de auto-modificação de um dispositivo adaptativo existente. Dentre os diversos dispositivos adaptativos, foi escolhido o autômato adaptativo.

Esta escolha foi motivada pelo seguinte raciocínio: Um código pode ser representado por meio de um diagrama de fluxo de controle (ANCKAERT; MADOU; BOSSCHERE, 2001) (tanto diagramas de fluxo de controle quanto o de fluxo de dados são utilizados para modelagem de análise convencional em engenharia de *software* (PRESSMAN, 2002)). Admitindo-se certa liberdade, um autômato também pode representar um diagrama de controle de fluxo, onde os estados do autômato representam os nós do diagrama de controle e as transições do autômato representam as arestas do grafo (tal frase pode ser justificada pelo fato de um autômato finito ou uma sub-máquina do autômato de pilha estruturado ser graficamente descrito por uma representação similar a de um grafo (LEWIS; PAPADIMITRIOU, 1981) (NETO; PARIENTE, 2003)). Então, se um código pode ser representado por um diagrama de fluxo de controle, e um autômato pode ser usado para simular este tipo de diagrama, pode-se afirmar que autômatos podem representar códigos.

Autômatos finitos adaptativos e autômatos adaptativos, por terem poder de expressão de máquina de Turing (NETO; PARIENTE, 2003) (ROCHA; NETO, 2000), também podem ser vistos como dispositivos de processamento (apesar de serem caracterizados como dispositivos de reconhecimento, dado que o autômato clássico é um dispositivo de reconhecimento de linguagens (LEWIS; PAPADIMITRIOU, 1981)).

Visando obter um modelo de computação mais simples e com maior semelhança ao modelo de execução usado em computadores domésticos, optou-se por estender o autômato adaptativo, adicionando a este as características do modelo de execução tradicional dos computadores domésticos, obtendo um novo dispositivo adaptativo denominado de autômato de execução adaptativo.

Portanto, no ambiente de execução proposto, um código escrito na linguagem AdaptCode é representado por meio de um autômato de execução adaptativo, cuja extensão em relação ao autômato adaptativo reside na associação de uma instrução do código (a ser efetivamente executada por um interpretador) a cada estado do autômato.

Enquanto a execução de um código de montagem adaptativo CF_0 pode ser representado por um espaço de códigos { CF_0, CF_1, CF_2, ... CF_n }, onde CF_i representa o código CF_{i-1} após a atuação da i-ésima ação adaptativa, em sua forma de execução, este espaço de códigos é representado por um espaço de autômatos de execução adaptativos { AEA_0, AEA_1, AEA_2, ... AEA_n } , onde AEA_n é o formato de execução do código CF_n (i.e. AEA_n representa o código CF_n convertido para a representação por meio de autômato de execução adaptativo).

```
Área de programa
FAT proc        //procedimento fatorial(n)
  pop n         //desempilha n (fat (n))
  mov eax,1     //acumulador ← 1
  cmp n,0       //compara n com 0
  // para os casos em que N <= 0, desvia para o fim do
  // procedimento, retornando o valor 1.
  jle X         //desvia para X se n <= 0
  // caso n > 0, modifica o código para calcular o fatorial
  adapt 1,step //adapta trecho 1
  mark 2        //trecho adaptativo 2
  mark 1        //trecho adaptativo 1 (vazio)
  emark 1       //fim de trecho adaptativo 1
  emark 2       //fim de trecho adaptativo 2
label X
  mov res,eax   //res ← eax
  adapt 2,base //adapta trecho 2
  ret           //retorna
FAT endproc
```

Figura 11 - Exemplo de código adaptativo de baixo nível (rotina fatorial). Baseado no exemplo desenvolvido em (PELEGRINI; NETO, 2008)

A relação entre o código escrito na linguagem AdaptCode e seu formato de execução encontra-se ilustrada na Figura 11 e na Figura 12. A Figura 11 apresenta o código do procedimento fatorial (calculado por meio de auto-modificação). Esta figura representa o código na forma no qual o programador escreve (código escrito

via linguagem AdaptCode). Em sua forma de execução, este código é representado como o autômato de execução adaptativo apresentado na Figura 12. Portanto, ambas as figuras representam um mesmo código, cada qual com um formato de representação diferente.

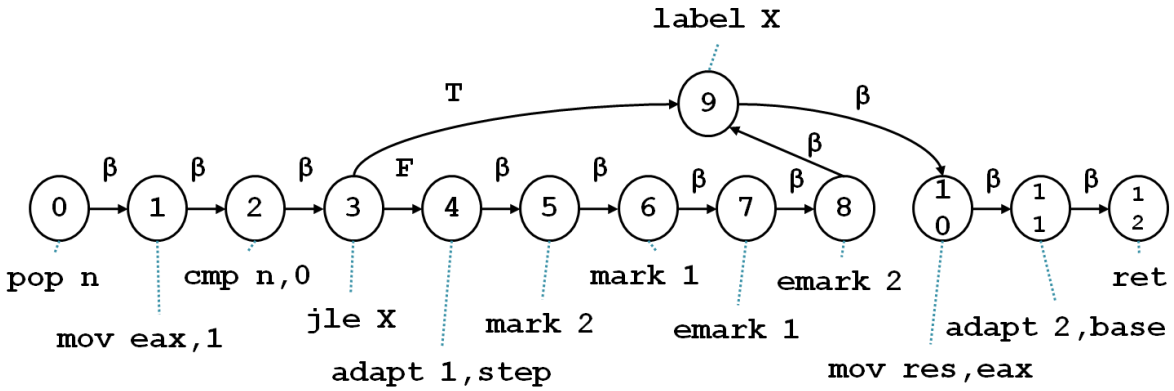


Figura 12 – Exemplo de formato de execução (representação por autômato de execução adaptativo) do código *assembly* adaptativo ilustrado na Figura 11.

4.1 ORIGEM DA PROPOSTA – MECANISMO DE EXECUÇÃO BASEADO EM GRAMÁTICAS

Apesar da fundamentação anteriormente descrita para o uso de autômatos na representação de códigos adaptativos, a origem da proposta remete à proposição de outro modelo de execução, baseado em gramáticas dinâmicas, apresentado em (AYCOCK, 2003).

Na proposta de Aycock, o código de um programa no formato de execução é representado por meio de uma gramática dinâmica (AYCOCK, 2003). O mecanismo de execução trabalha com os símbolos terminais da gramática representando instruções de máquina (ou seja, a sentença final representa o código efetivamente executado) e de que o programa inicial é descrito pelo não-terminal inicial da gramática (AYCOCK, 2003). A execução do programa é composta por duas fases que se alternam: a de geração do código a ser executado, por meio das regras de derivação descrita na gramática (expansão de não-terminais existentes na sentença) e a de execução das instruções geradas (terminais gerados).

O princípio de execução, que se assemelha ao processo dos analisadores sintáticos top-down (NETO, 1987) (AHO et al., 2006), consiste em varrer as formas de código sentenciais da esquerda para a direita, executando as instruções associadas aos terminais gramaticais até atingir um símbolo não-terminal. Ao atingir um símbolo não terminal, a execução de código é temporariamente interrompida e o não-terminal é expandido. Após a expansão, a execução retorna ao modo de funcionamento anterior, partindo do ponto onde foi interrompido.

Neste contexto de execução, as regras dinâmicas (AYCOCK, 2003) são utilizadas para tratar os casos de desvio de fluxo do programa, eliminando assim a necessidade de tais instruções (AYCOCK, 2003). Esta característica desmotivou o uso dessa abordagem ou do uso de modelos derivados das gramáticas adaptativas para o ambiente de execução, dado o fato que é desejado que o mecanismo de alteração dinâmica seja focado apenas para o processo de alteração do código. Como as gramáticas não apresentam tratamento prático para representar os mecanismos de controle de fluxo existentes nos códigos, recorreu-se a outros dispositivos.

Aproveitando-se do fato que formalismos adaptativos da classe dos autômatos e das gramáticas são equivalentes (em (PARIENTE, 2004) faz-se um estudo detalhado das gramáticas adaptativas, e em (IWAI, 2000) mostra-se como converter gramáticas adaptativas em autômatos adaptativos) e ambos têm poder de expressão de máquina de Turing, surgiu à opção de montar um modelo de execução que permita a alteração do código baseado em autômatos adaptativos.

4.2 DEFINIÇÃO

O autômato de execução adaptativo é um dispositivo derivado do autômato adaptativo (NETO, 1993) (NETO; PARIENTE, 2003), proposto com o objetivo de representar, no ambiente de execução, códigos escritos na linguagem AdaptCode.

Uma determinada instância de um autômato de execução adaptativo representa um determinado código adaptativo. Cada procedimento declarado no código é

associado a uma sub-máquina do autômato de execução adaptativo, implementando assim a forma de endereçamento das instruções do ambiente de execução.

A extensão do autômato adaptativo, feita para obter-se o autômato de execução adaptativo, consiste na execução de instruções, pertencente à linguagem e as quais são interpretadas pelo ambiente de execução, associadas aos estados do autômato. Em outras palavras, no estabelecimento de um conjunto de instruções I, formado de instruções suportadas pelo ambiente de execução, e na associação de uma instrução pertencente ao conjunto I a cada estado do autômato.

Um autômato de execução adaptativo é definido por uma 4-upla (S, s_0, I, Γ), onde S representa o conjunto de sub-máquinas declaradas, s_0 representa a sub-máquina inicial ($s_0 \in S$), I representa o conjunto que descreve todas as instruções aceitas pelo ambiente de execução (incluindo as instruções associadas à modificação de código/função adaptativa) e que podem ser associadas a um estado (para simplificar a descrição, nessa dissertação as instruções são apresentadas por meio da correspondente notação simbólica usada na linguagem AdaptCode) e Γ representa a pilha do autômato.

Cada sub-máquina do autômato de execução é declarada pela 6-upla (Q, W, Σ, δ, q_0, F), onde:

- Q é o conjunto finito e não vazio de estados da sub-máquina;
- W é o conjunto finito e não vazio que associa cada um dos estados da sub-máquina a alguma instrução suportada pelo ambiente de execução (conjunto I). Essa associação é representada por meio de pares ordenados da forma (q_i, i_j), onde $q_i \in Q$ e $i_j \in I$. Como a cada estado deve ficar associada apenas uma instrução, o conjunto W deve conter, obrigatoriamente, apenas um elemento para cada estado declarado no conjunto Q;
- Σ é o alfabeto binário de entrada do autômato: Σ = {T, F};
- δ é o conjunto das transições da sub-máquina. Cada estado declarado aceita uma das seguintes transições: (i) duas transições (a primeira, rotulada com o símbolo T e a outra, com o símbolo F); (ii) uma transição de chamada de sub-máquina; ou (iii) uma transição de retorno de sub-máquina;
- q_0 representando o estado inicial da sub-máquina, que contém a primeira instrução do procedimento representado;

- F representando o conjunto finito e não-vazio de estados finais da sub-máquina. Este conjunto é composto por todos os estados que estiverem associados à instrução de fim de execução do procedimento.

4.2.1 Transições e correspondentes instruções.

Assim como no autômato de pilha estruturado, existem três tipos distintos de transições no autômato de execução adaptativo (NETO; MAGALHÃES, 1981): as transições internas, as transições de chamada de sub-máquina e as transições de retorno de sub-máquina.

A construção do conjunto de transições ð de cada uma das sub-máquinas de um determinado autômato de execução adaptativo deve atender a regras específicas. Dado que o esquema de endereçamento de instruções é baseado nos estados do autômato, as transições são responsáveis por determinar a próxima instrução a ser executada. Portanto, a(s) transição(ões) que partem de um determinado estado é(são) função da instrução associada ao estado em questão.

Para explicar as regras de construção do conjunto de transições (e, conseqüentemente, determinar o esquema de obtenção da próxima instrução), o conjunto de instruções I é dividido inicialmente em duas partições (baseado nas classificações de instruções existentes em (HENNESSY; PATTERSON, 2003)): instruções que não alteram o fluxo de execução e as que alteram o fluxo de instruções (conforme ilustrado na Figura 13.

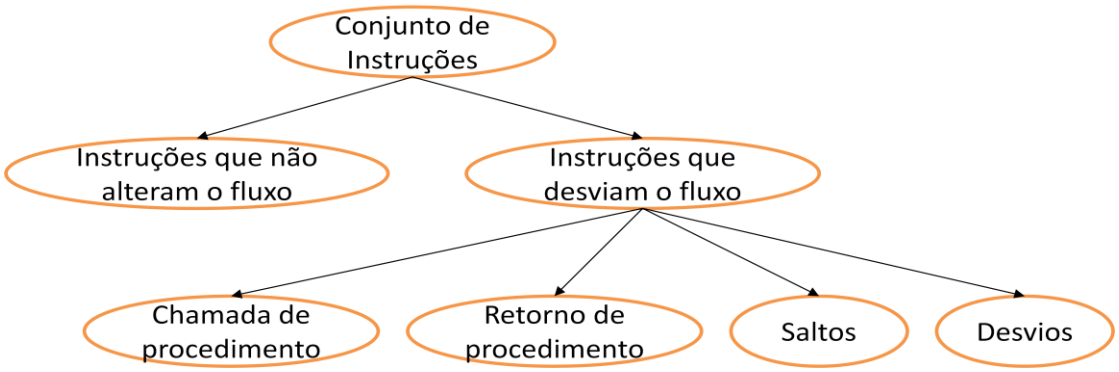


Figura 13 - Particionamento do conjunto de instruções I

As instruções que não alteram o fluxo de execução são aquelas cuja operação não influênciam na determinação da próxima instrução a ser executada. Podem-se citar como exemplos as instruções: de soma (`add`), de multiplicação (`mult`), operações de pilha de sistema (`push` e `pop`) e a instrução de modificação de código `adapt`. Um estado q_i que contenha este tipo de instrução indica a próxima instrução, contida no estado q_j, através das transições internas $(q_i, T) \rightarrow q_j$ e $(q_i, F) \rightarrow q_j$, sendo que q_i, $q_j \in Q$ de uma determinada sub-máquina declarada (ilustração Figura 14(a)). Por questões de simplificação, adota-se que a transição $(q_i, \beta) \rightarrow q_j$ é equivalente as transições $(q_i, T) \rightarrow q_j$ e $(q_i, F) \rightarrow q_j$ (i.e. $\beta = T \mid F$).

O tratamento deste tipo de instrução consiste no suporte a esta por parte do mecanismo de interpretação de instruções, que após a execução pode retornar tanto T ou F, com o intuito de realizar a transição para a próxima instrução a ser executada (por padrão retorna-se o símbolo T).

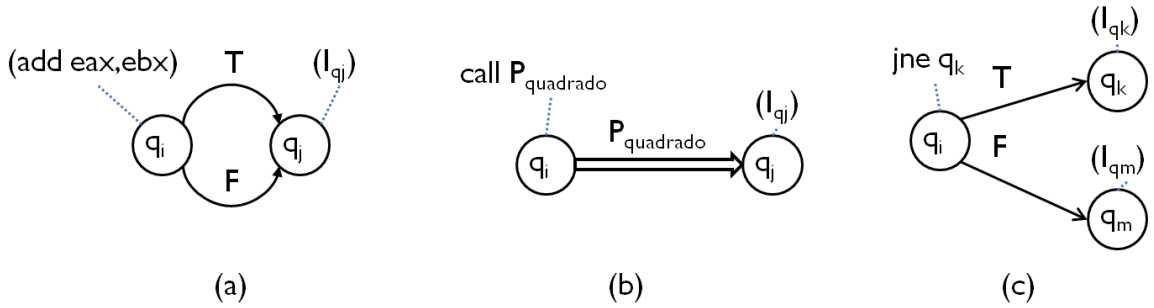


Figura 14 - Esquema ilustrativo de execução de instruções. (a) esquema da instrução de soma `add eax,ebx`. (b) esquema da invocação da sub-rotina quadrado. (c) esquema da instrução de desvio condicional `jne` q_k (salta para q_k se um determinado bit de controle for igual a zero).

As instruções que alteram o fluxo de execução são aquelas cuja operação afeta ou pode afetar (para os casos dos desvios, que são saltos condicionais) diretamente na determinação da próxima instrução a ser executada. Seguindo a categorização de (HENNESSY; PATTERSON, 2003), podem ser subdividas em quatro classes distintas (veja Figura 13): saltos (desvios incondicionais), desvios (desvios condicionais), chamada de procedimentos e retorno de procedimentos. Cada uma dessas classes possui um esquema particular de representação.

As instruções de rotinas são aquelas que invocam um procedimento (exemplo: `call`) e que retornam de um procedimento (exemplo: `ret`). Ao se entrar em um estado (q_i) que contenha uma chamada de procedimento, deve-se salvar o valor do

próximo estado indicado pela transição de chamada de sub-máquina (por exemplo, no caso da Figura 14(b), deve-se salvar o estado q_j da transição de chamada de sub-máquina $(q_i, \varepsilon) \rightarrow (\downarrow(q_{j,} s_{corrente}), s_{pquadrado})$, onde $q_i, q_j \in Q$ da sub-máquina $s_{corrente}$ e $s_{corrente}, s_{pquadrado} \in S$) e o identificador da sub-máquina corrente, junto com os dados dos registradores. Uma vez que a posição de retorno é armazenado na pilha do autômato, não existe necessidade de armazená-la na pilha de sistema, como é feito na linguagem de montagem. No entanto, a pilha de sistema é utilizada para armazenar passagem de parâmetros e dos valores dos registradores.

Após o término da execução da máquina chamada (atingiu o estado final desta máquina – instrução `ret`), restaura-se o contexto salvo, representado pelo par (estado, sub-máquina), mais as informações de registradores salvas. No exemplo da Figura 14(b), têm-se a transição $(q_f, \varepsilon) \rightarrow (\uparrow(q_{j,} s_{corrente}))$, onde $q_f \in F$ da sub-máquina $s_{pquadrado}$.

Como evidenciado pela descrição no parágrafo anterior, o processo é similar a chamada de sub-máquina, definido no autômato de pilha estruturado (NETO; MAGALHÃES, 1981) e no autômato adaptativo (NETO; PARIENTE, 2003), exceto pelo fato de salvar e recuperar as informações dos registradores e passagem de parâmetro. Resumindo, de um estado associado à chamada de procedimento parte uma transição de chamada de sub-máquina (como exemplo da Figura 14(b)) e de um estado associado à instrução de retorno de procedimento parte uma transição de retorno de sub-máquina.

Já as instruções de desvio são aquelas que podem saltar para alguma instrução especificada (o endereço de destino para o caso que ocorre o desvio é indicado como operando da instrução). Neste esquema, uma instrução de desvio apenas testa a condição e, se for satisfeita, escreve T na cadeia de trabalho, caso contrário, escreve F (NETO, 1994). De um estado que está associado a uma instrução deste tipo (exemplo, o estado q_j da Figura 14(c)), partem duas transições internas: (i) uma do tipo $(q_i, T) \rightarrow q_k$, onde $q_i, q_k \in Q$ e q_k é o endereço da instrução a ser executada caso ocorra desvio; e (ii) outra do tipo $(q_i, F) \rightarrow q_m$, onde $q_i, q_m \in Q$ e q_m é o endereço da instrução a ser executada caso não ocorra o desvio (exemplo Figura 14(c)).

Caso um estado esteja associado a uma instrução de salto, a próxima instrução é determinada de maneira semelhante ao caso das instruções que não alteram o fluxo

de execução. A próxima instrução a ser executada é aquela apontada pela transição interna (q_i, T) → q_j, onde q_i, $q_j \in Q$ e q_j é o endereço da instrução a ser executada (ressaltando a existência de outra transição interna do tipo (q_i, F) → q_k, onde q_i, $q_k \in Q$ e q_k é o endereço da instrução imediatamente posterior a instrução de salto na representação do código via linguagem AdaptCode).

A instrução de salto, quando o destino é sempre fixo (i.e. o destino não pode ser modificado ao longo da execução do código) não é necessária, uma vez que neste modelo cada instrução indica, explicitamente, a próxima. Entretanto, como este tipo de instrução existe na linguagem AdaptCode (devido à forma de escrita do código, similar ao *assembly* tradicional, exemplo no item 1.1), o tratamento deste tipo de instrução foi considerada.

Pelo fato da representação via autômato indicar a próxima instrução, processos de otimização de código – como o *peephole* (MCKEEMAN, 1965) (AHO et al., 2006) – podem eliminar estas instruções do modelo de execução, aumentando a eficiência do código.

4.2.1.1 Casos especiais e limitações

Na arquitetura Intel® 32 bits as instruções de salto, desvio e chamada de procedimento podem conhecer o destino do desvio do fluxo de execução apenas no momento da execução desta instrução (ou seja, em tempo de execução) (DETMER, 2001) (INTEL, 2007) (INTEL N-Z, 2007). Por exemplo, ao utilizar o modo de endereçamento direto para indicar o destino de um desvio, o endereço de destino do desvio somente é, de fato, conhecido durante a execução da instrução, dado que ao longo da execução do código a informação armazenada na posição de memória que contém o endereço de destino pode ser modificada.

Em contrapartida a tal fato, no modelo de salto e desvio descrito anteriormente, o destino deve ser conhecido previamente. Esta informação é necessária para definir o estado destino da transição e, assim, montar o autômato.

O autômato de execução adaptativo prevê o tratamento dos casos em que o destino do desvio se modifica em tempo de execução por meio de rotinas semânticas, as

quais corrigem o destino da transição. Cita-se como exemplo o caso cujo operando do destino é passado de maneira indireta. Nesse caso, durante a interpretação desta instrução é ativada rotinas semânticas que alteram o destino da transição associada ao salto/desvio.

Para ilustrar o processo considere o caso da Figura 15(a). A instrução associada ao estado q_i desconhece o destino em tempo de execução, mas previamente foi associado como destino do salto o estado q_k, como pode ser visto pela transição (q_i, T) → q_k. Ao executar a instrução associada ao estado q_i, o interpretador obtém a informação que, naquele momento, o valor armazenado em x esta associado ao estado q_m. Neste momento, ativa-se a rotina semântica de correção do autômato, que modifica o destino da transição associada ao salto, passando de (q_i, T) → q_k para (q_i, T) → q_m, como pode ser observado na Figura 15(b).

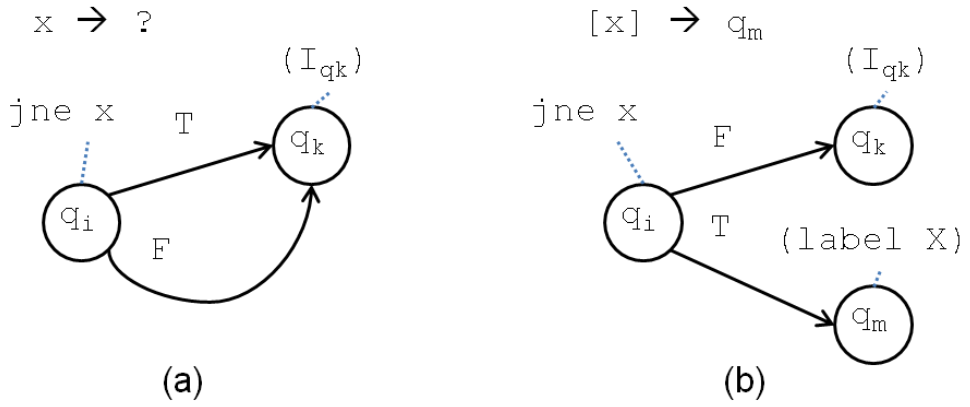


Figura 15 - Exemplo de cálculo do destino de um desvio/salto em tempo de execução

A única limitação deste processo de correção se deve ao fato de que o endereço de destino (ou seja, o estado para onde se vai saltar) deve, obrigatoriamente, pertencer à mesma sub-máquina. Esta restrição se deve a definição de transição interna (i.e. conforme definido em (NETO; PARIENTE, 2003), a transição interna possui o estado de origem e o estado de destino na mesma sub-máquina). Adicionalmente, também evita problemas de erro na pilha de sistema.

O caso da chamada de procedimento apresenta uma solução similar. Entretanto, ao invés da ação modificar o endereço de destino da transição de chamada de sub-máquina, a rotina semântica corrige sub-máquina chamada. Por modificar apenas a

chamada, o valor contido na posição deve estar associado a uma das sub-máquinas declaradas (ou seja, é possível chamar apenas as sub-máquinas declaradas).

4.2.2 Computação e Configuração

A computação de um autômato de execução inicia-se do estado inicial (q_0) da sub-máquina inicial (s_0). Partindo-se desse estado, a computação consiste nos seguintes passos: (i) passar a instrução associada ao estado para o interpretador de instruções, onde será executada; (ii) ler o símbolo existente na cadeia de trabalho, fornecido pelo interpretador de instruções após a execução da instrução fornecida no passo 1 (eventualmente, pode-se abdicar da leitura, transitando em vazio, como nos casos de chamada e retorno de procedimento); e (iii) transitar, em função do símbolo lido no passo 2, para o próximo estado do autômato de execução adaptativo.

Este processo se repete até atingir um dos estados finais da sub-máquina inicial (que corresponde à última instrução a ser executada). Caso, durante a execução, seja atingido um estado que não esteja definido no conjunto Q ou que ao estado esteja associado a uma instrução cuja definição não esteja declarada no conjunto I, a execução será abortada.

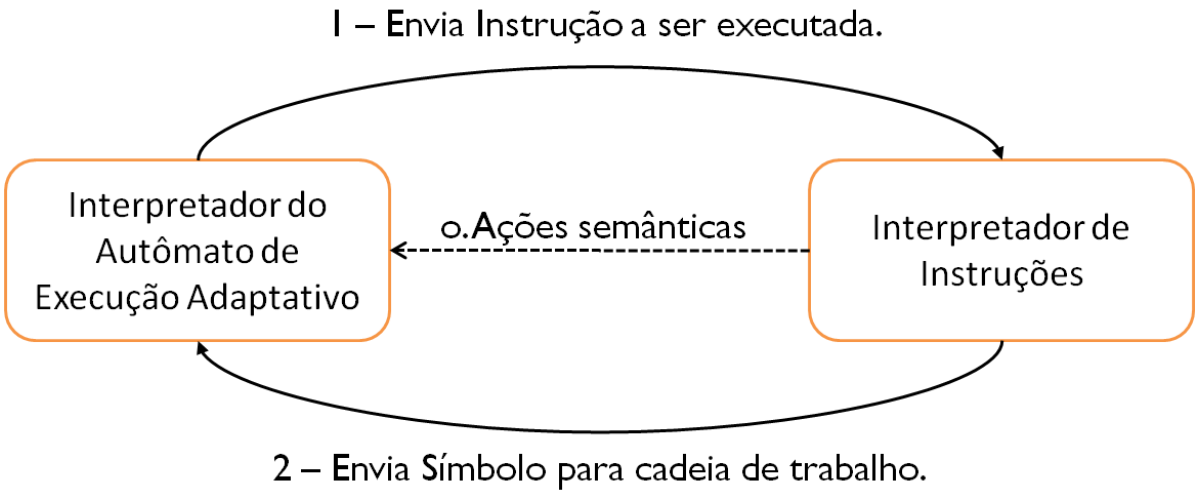


Figura 16 – Funcionamento da computação / transdução.

Este processo se assemelha ao modelo de execução do transdutor finito (NETO, 1987) e a máquina de Moore (LEWIS; PAPADIMITRIOU, 1981) (BABCSÁNYI, 2000), onde a instrução associada a cada estado constitui a saída e o símbolo fornecido pelo interpretador de instruções constitui a entrada (conforme esquematizado na Figura 16).

O algoritmo que descreve a execução pode ser visto em Algoritmo 1. Ressalta-se que a função `executa_instrução`, que consiste no acionamento do interpretador de instruções, além de ser responsável pela execução da instrução associada ao estado, também é responsável pelas rotinas semânticas de correção do autômato (para os casos listados no item 4.2.1.1) e pela função adaptativa (descrita no item 4.3).

Algoritmo 1: Interpretação do Autômato de Execução Adaptativo

Entrada: `Instância de um autômato de execução adaptativo`

Condição inicial:

```
estado_corrente = estado inicial da sub-máquina principal
sub-máquina_corrente = sub-máquina principal
estado_final = conjunto F da sub-máquina principal
```

Algoritmo:

```
while( (estado_corrente != estado_final) && (estado_corrente
∈ Qsub-máquinacorrente) )
{
  símbolo = executa_instrução(instrução(estado_corrente));
  executa_transição(estado_corrente,   sub-máquina_corrente,
  símbolo);
  atualiza(estado_corrente, sub-máquina_corrente);
}
If( (estado_corrente == estado_final ) && ( Γ == ε ))
{
  print("Execução Terminou com sucesso");
}else{
  print("Falha na execução");
}
End;
```

Para finalizar este tópico, define-se o conceito de configuração para os autômatos de execução adaptativo como sendo uma seqüência ordenada do tipo ((AEA),(q, s, i, Z),(α)).

O primeiro elemento da seqüência ordenada (AEA) é a própria declaração do autômato de execução adaptativo a ser executado, que pode sofrer modificações durante a execução. O segundo elemento, que consiste na 4-upla (q, s, i, Γ), representa a situação corrente do autômato, sendo que: q representa o estado corrente, s representa a sub-máquina corrente, i representa a instrução associada ao estado q e Γ representa a situação da pilha do autômato de execução. O terceiro elemento da seqüência ordenada (α) representa o símbolo escrito na cadeia de trabalho após a execução da instrução i (α pode ser T ou F) ou a cadeia vazia (simbolizado por ε), para os casos em que a instrução não edita a cadeia de trabalho.

Ressalta-se que o elemento (α) representa o símbolo da cadeia de entrada a ser utilizado pela próxima transição a ser executada. A separação do símbolo corrente da cadeia de trabalho do estado do autômato é motivada pelo fato de o mesmo ser gerado após a execução da instrução i.

Uma vez definida a configuração, a computação do autômato de execução adaptativo pode, eventualmente, ser descrita por uma seqüência de configurações do tipo: ((AEA),(q, s, i, Γ),(α)) ├ ((AEA')(q', s', i', Γ'),(α')), onde ├ representa a relação entre duas configurações após a execução de uma transição.

4.3 MODELO DE ALTERAÇÃO DO CÓDIGO

Como comentado no inicio deste capítulo, um dos motivos de propor um modelo de execução baseado em autômatos adaptativos foi o fato de este dispositivo possuir um mecanismo que provê a capacidade de se auto-modificar, o que permite aproveitar idéias e conceitos previamente validados para definir o processo de alteração do código.

Para explicitar o mecanismo adaptativo de modificação do código, este é dividido em duas componentes lógicas: o elemento básico de alteração e o mecanismo de alteração, descritos nos itens a seguir.

4.3.1 Elemento básico de alteração

O elemento básico de alteração associado ao autômato de execução adaptativo (também denominado trecho de código adaptativo) corresponde à representação, em autômato de execução adaptativo, dos trechos de códigos adaptativos definidos no item 3.1.2.1. Tal elemento consiste em um trecho de uma sub-máquina do autômato (ou seja, o processo de auto-modificação altera um pedaço do grafo que descreve uma sub-máquina), que contenha: (i) os estados associados as instruções `mark` e `emark` que delimitam um determinado trecho de código adaptativo, bem como os estados associados as instruções que, na representação em código de montagem adaptativo simbólico, estão contidas neste trecho; e (ii) as transições que partem dos estados acima especificados, mais a transição que parte do estado associado à instrução imediatamente anterior a instrução `mark`, na representação por código de montagem adaptativo simbólico. As transições que partem de estados que não pertencem ao trecho adaptativo e possuem como destino um estado interno ao trecho (com exceção do estado associado a instrução `mark`) não fazem parte do trecho que pode ser modificado

Esta escolha deve-se ao fato de que o autômato de execução adaptativo foi proposto com o intuito de ser utilizado para representar a forma de execução dos códigos escritos na linguagem AdaptCode. Portanto, nada mais natural que compartilhem o mesmo elemento básico de alteração, com ressalva em relação à forma de representação dos trechos de códigos adaptativos na linguagem AdaptCode e no autômato de execução adaptativo, que são diferentes.

Dado que o trecho a ser alterado é delimitado pelas instruções `mark` e `emark`, não é possível, em teoria, declarar novos procedimentos ou remover procedimentos existentes, dado que estas instruções são internas a uma sub-máquina. Esta restrição também se deve ao fato de que na definição do autômato adaptativo não

foi especificado a criação e remoção de sub-máquinas (apesar de algumas propostas descreverem o procedimento de cópia de uma sub-máquina).

Embora não seja permitido criar novos procedimentos com este modelo de alteração, não existe redução da expressividade, visto que sempre é possível definir inicialmente um procedimento contendo um comando de múltipla escolha (desvios), no qual cada escolha corresponderia a "um procedimento" diferente. Como cada elemento do comando de múltipla escolha pode ser criado ou removido dinamicamente, é possível, assim, simular a criação de tantos procedimentos quantos forem necessários.

As instruções de marcação dos trechos de códigos adaptativos (`mark` e `emark`) são usadas, pelo ambiente de execução, durante dois momentos distintos:

- Durante a montagem do programa (conversão do código escrito na linguagem AdaptCode para o formato de execução baseado em autômatos, realizada antes de sua execução), as instruções são usadas para formar uma tabela de controle dos trechos de códigos adaptativos, que será usada pelo interpretador durante a execução. Esta estrutura de controle tem como objetivo agilizar a localização do trecho, bem como tratar os casos de referência indireta aos trechos de códigos adaptativos (o indexador associado ao trecho marcado pode ser passado de forma indireta para a instrução de alteração de código).
- Em tempo de execução, estas instruções permitem que o ambiente de execução identifique que trechos de códigos adaptativos se encontram em "execução" (especificamente, dentro de quais trechos de códigos adaptativos a instrução em execução se encontra). Esta informação é particularmente importante para os casos em que a instrução de alteração do código esteja contida no trecho de código adaptativo a ser alterado. Não obstante, nos casos de chamadas de procedimento, as informações dos trechos de códigos adaptativos em "execução" são empilhadas pelo ambiente de execução, para caso estes trechos sejam modificados (por exemplo, caso o procedimento chamado altera o trecho de código que contém a instrução de retorno, que foi armazenada durante a chamada do procedimento), o ambiente de execução possa corrigir os endereços de retorno armazenados na pilha.

Um exemplo ilustrativo de marcação e de sua estrutura de controle do elemento básico de alteração é fornecido na Figura 17. O trecho que começa no estado 1 e vai até o 5 corresponde ao trecho de código adaptativo CA1 (indexado por 1). A estrutura de controle armazena:

- O nome simbólico do trecho (índice usado no código *assembly* adaptativo simbólico);
- O indexador do trecho, fornecido durante a montagem do código simbólico;
- A transição que parte da instrução anterior a instrução `mark` no código simbólico e que aponta para esta instrução (denominada de transição de entrada do trecho). No caso da Figura 17, a transição de entrada é (0, β) → 1.
- A transição que do estado associado instrução `emark` aponta a próxima instrução (denominada de transição de saída do trecho). No caso da Figura 17, a transição de saída é (5, β) → 6.

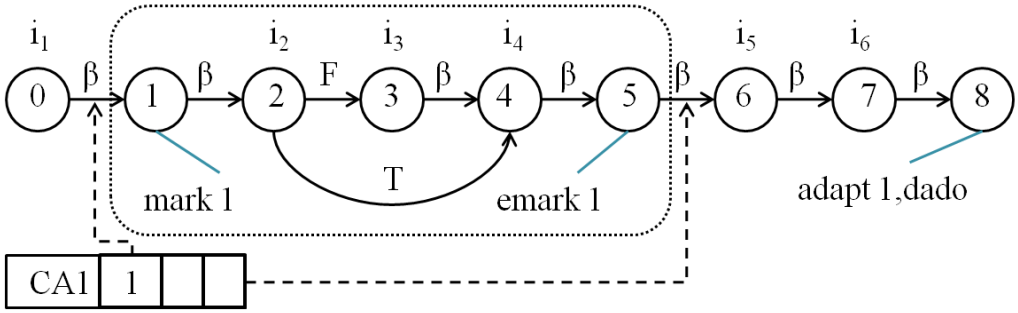


Figura 17 - Elemento de alteração e sua estrutura de controle - Trecho de código adaptativo CA1 (indexado pelo número 1).

Como restrição a esse modelo têm-se que os estados associados às instruções `mark` e `emark` podem ser estados de origem das transições que partem de apenas um estado (instrução imediatamente anterior a instrução `mark` ou `emark`). Caso o ambiente de execução encontre um caso que não atenda essa condição, durante a montagem é inserida uma instrução de `nop` antes da instrução `mark` ou `emark` em questão.

Caso a instrução `mark` e a `emark` de um determinado trecho de código adaptativo sejam, respectivamente, a primeira e a última instrução de um procedimento, um artifício é usado, durante a montagem, para que estas se tornem a segunda e a

penúltima instrução, com intuito de garantir a existência das transições de entrada e saída. Este artifício consiste na inserção de uma instrução `nop` (*no operation*) antes da instrução `mark` e outra depois da instrução `emark`.

4.3.2 Mecanismo de alteração - Função Adaptativa

O mecanismo responsável por executar a alteração do código baseia-se na camada adaptativa proposta para os autômatos adaptativos, ou seja, nas funções adaptativas. Ao contrário do autômato adaptativo, onde cada função adaptativa costuma ser declarada à parte do autômato, no modelo descrito neste artigo existe apenas uma função adaptativa declarada, de caráter generalista. Portanto, esta função adaptativa declarada é capaz de realizar todos os tipos permitidos de alteração de código, evitando, assim, a necessidade de declarar outras funções adaptativas (o apêndice 1 ilustra que ambos os modelos possuem a mesma expressividade).

Tal decisão é motivada pelo fato do formato de execução ser transparente ao programador e, portanto, não ter sentido atribuir a este a tarefa de criar as funções adaptativas associadas ao autômato de execução adaptativo.

Esta função adaptativa, descrita a seguir, é ativada por meio de uma instrução, pertencente ao conjunto I, com o seguinte formato: `adapt bloco,cod`. Durante a execução desta instrução (`adapt`), os operandos desta são passados como parâmetro para a função adaptativa. O primeiro argumento da função adaptativa (associado ao operando `bloco`) refere-se ao indexador do trecho de código adaptativo a ser alterado (a passagem do argumento pode ser de maneira direta – próprio identificador – ou indireta – posição de memória que contenha o identificador desejado). O segundo argumento (associado ao operando `cod`) refere-se à posição de memória da área de dados que contém o código a ser montado e inserido ao programa. Caso um desses dois argumentos não seja válido (como, por exemplo, o identificador do trecho não exista), a execução do programa é abortada.

O funcionamento desta ação adaptativa pode ser dividido em três passos:

- Primeiro Passo (ação adaptativa de consulta): Este passo possui um comportamento similar ao de uma ação adaptativa elementar de consulta indexada. Consiste em buscar as transições de entrada e saída do trecho de código adaptativo, a partir do parâmetro referente ao indexador do trecho de código adaptativo a ser alterado, como exemplificado na Figura 18.

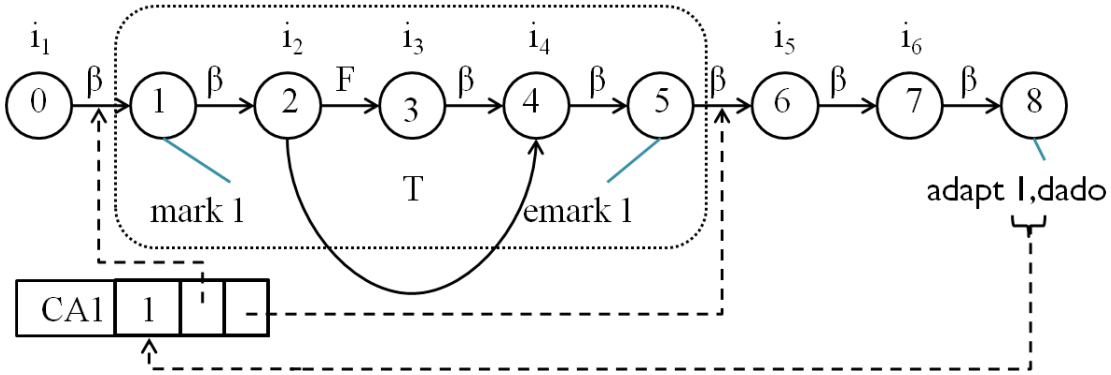


Figura 18 – Passo 1 - Procura do trecho de código adaptativo (no caso indexado por 1).

  - A busca as transições de entrada e saída é feito por meio da localização do indexador associado ao trecho de código adaptativo a ser modificado na estrutura de controle associado aos trechos de códigos adaptativos mantido no ambiente de execução (essa estrutura de controle consiste em uma tabela que associa o nome simbólico do trecho, o indexador do trecho utilizado internamente pelo ambiente de execução e as posições das transições de entrada e saída do trecho, conforme mencionado no item anterior).

- Segundo Passo (ação adaptativa de remoção): Remover o código existente no trecho que vai ser alterado (incluindo as instruções de marcação), salvando as informações das transições de entrada (*ts*) e a de saída (*te*) deste trecho (exemplo: Figura 19). Ao se remover o código, tanto às instruções do trecho quanto as informações de controle que o interpretador possui são destruídas (caso exista um ou mais trechos declarados dentro do trecho a serem removidos, estes também serão removidos);

  - Em relação às instruções de salto/desvio: No modelo teórico existem infinitos estados declarados (portanto Q é o conjunto de estados utilizado).

Caso algum estado pertencente ao trecho de código adaptativo removido (por removido, interprete que deixou de ser utilizado) seja destino de uma instrução de salto/desvio situada fora do trecho removido, ao ser executada esta instrução desviará para um estado que não pertencem ao conjunto Q e, portanto, a execução será abortada. Entretanto, para o caso prático, cuja quantidade de memória é limitada, essas transições são corrigidas por ações semânticas associadas ao ambiente de execução.

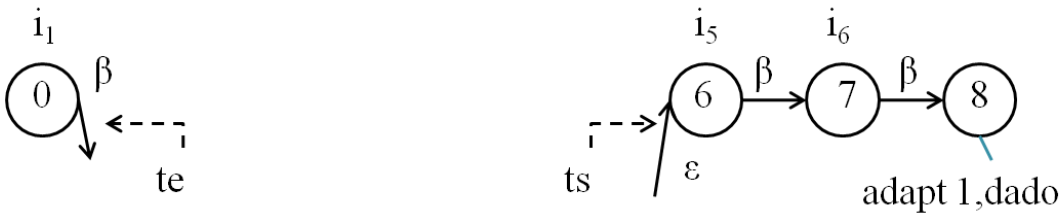


Figura 19 - Passo 2 - Remoção do trecho de código adaptativo que esta sendo alterado

- Terceiro Passo (ação adaptativa de inserção): O terceiro e último passo consiste na inserção de novas instruções no lugar do trecho apagado.
  Caso o parâmetro referente à posição de memória onde se encontra o trecho de código a ser inserido seja igual a zero, não será inserida nenhuma instrução (representação do processo de exclusão de um trecho). Neste caso, a transição salva como *te* é modificada de forma a ter como estado de destino o mesmo estado de destino da instrução *ts*. Em outras palavras, essa operação conecta a instrução imediatamente anterior ao bloco removido com a instrução imediatamente posterior, como ilustrado na Figura 20.

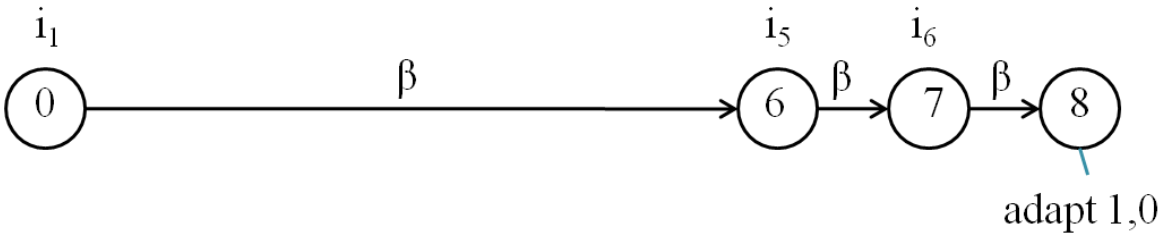


Figura 20 – Passo 3 – Programa após modificação caso no estado 8 existisse a instrução `adapt 1,0` no lugar de `adapt 1,dado`.

No caso de existir instruções que devem ser inseridas, esta etapa é subdivida em duas partes:

- A primeira parte consiste em montar o trecho de código armazenado na área de dados (segundo parâmetro da função adaptativa), conforme o ilustrado no exemplo da Figura 21. O código armazenado pode conter instruções que criam um ou mais trechos de códigos adaptativos. As instruções devem estar escritas na linguagem AdaptCode (para facilitar a escrita de códigos) e, portanto, deverão ser montadas e convertidas para a representação interna do ambiente de execução. Caso a montagem do código existente da área de dados falhe, o ambiente de execução associado à linguagem recebe um aviso de erro e efetua o devido tratamento (até o presente momento, este tratamento consiste no término da execução do código).

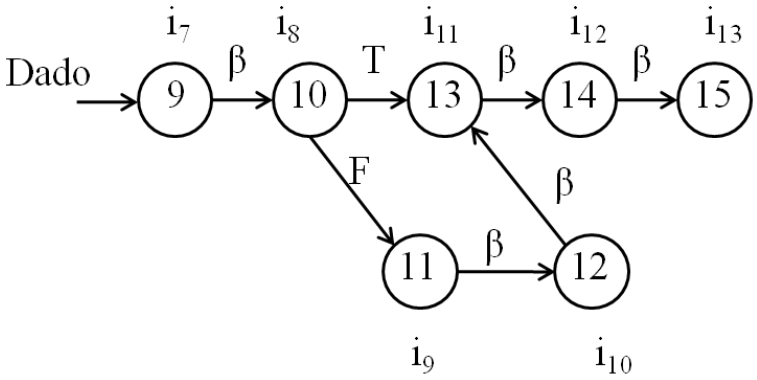


Figura 21 - Passo 3.1 - Programa escrito em dado (parâmetro) montado.

  - Durante a montagem da imagem existente na área de dados, adota-se a seguinte convenção em relação aos *labels*: (i) caso uma instrução de salto/desvio possua como operando um *label* que esteja declarado nesta mesma imagem, tal *label* é resolvido em função da declaração existente no próprio trecho a ser inserido (independentemente de este *label* ter sido utilizado previamente em outro ponto do código), ou seja, neste caso o *label* funciona como uma espécie de gerador. Um exemplo desta situação encontra-se no capitulo 6; e (ii) caso uma instrução de salto/desvio possua como operando um *label* que não esta declarado nesta mesma imagem, o montador consultará o ambiente de execução a procura de informações sobre este *label*

(armazenadas em estruturas de controle auxiliar). Caso tenha sido utilizado anteriormente, tal *label* é resolvido em função desta informação, caso contrário a montagem resulta em falha.

- A segunda parte remete à modificação em si: inserir o novo trecho no lugar do trecho removido durante o segundo passo (como no exemplo da Figura 22). Para realizar esta inserção, é atribuído um novo estado de destino (estado associado à primeira instrução do trecho montado) a transição salva como *ts* e um novo estado de origem (estado associado à última instrução montada) para a transição salva como *te*. Caso a instrução `adapt` esteja contida dentro do trecho que será alterado, a próxima instrução a ser executada é aquela contida no estado de destino da transição *ts* (por exemplo, para a situação descrita na Figura 18, a próxima instrução seria a referente ao estado 6).

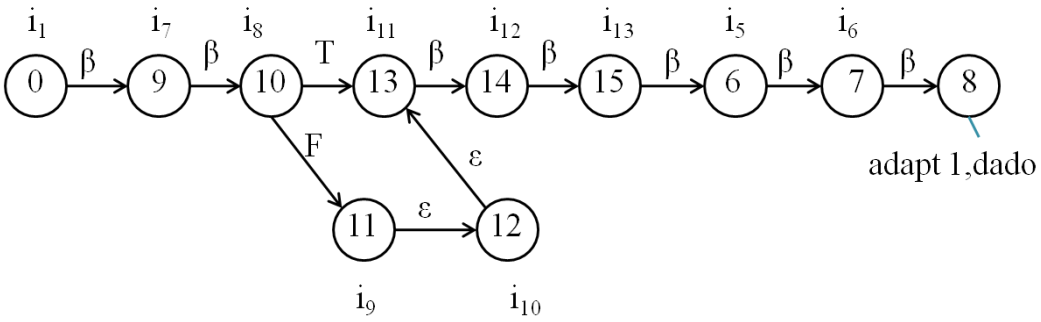


Figura 22 - Programa após alteração de código.

Todo este processo de alteração de código descrito não é visível ao programador, sendo visível apenas a chamada da função adaptativa, por meio da instrução `adapt`. Espera-se, com isso, facilitar o uso desta linguagem AdaptCode.

5 AMBIENTE DE EXECUÇÃO

Este capítulo complementa a descrição do ambiente de execução, iniciada com o detalhamento do autômato de execução adaptativo feito no capítulo anterior. Para completar tal descrição, esse capítulo é dividido em duas partes.

A primeira parte ilustra os passos necessários para a construção de uma máquina de Turing capaz de executar códigos adaptativos. Assim como o modelo de execução baseado em gramáticas adaptativas serviu de origem para o autômato de execução adaptativo, esse modelo definiu as linhas gerais de funcionamento e organização do ambiente de execução da linguagem AdaptCode.

Esse modelo é didático e permite a compreensão dos fundamentos por trás do ambiente de execução implementado, principalmente em relação ao modelo de alteração de código definido nessa dissertação, bem como ilustra como este ambiente pode ser representado via máquina de Turing.

Em contrapartida, a segunda parte desse capítulo faz considerações a respeito da arquitetura do ambiente de execução definido. Adicionalmente, procura-se detalhar os aspectos de implementação adotados para construir um ambiente de execução para programas adaptativos em um ambiente não-adaptativo.

5.1 MÁQUINA DE TURING PARA CÓDIGOS ADAPTATIVOS

A máquina de Turing universal apresenta duas características que são interessantes para essa dissertação: (i) a declaração da máquina de Turing a ser executada é feita na fita; e (ii) a fita pode ser editada ao longo da execução. Portanto, por (i) e (ii) constata-se a possibilidade de modificar, em tempo de execução, a própria declaração da máquina de Turing que está sendo executada pela máquina universal. Partindo-se dessa constatação, pode-se construir um ambiente de execução no qual o programa a ser executado é armazenado na fita da máquina de Turing, permitindo assim a auto-modificação desse.

Seguindo essa linha de pensamento, foi estabelecido um pseudo-modelo (a utilização desse termo se deve ao fato de sua descrição não ser formalmente completa) que inspirou o mecanismo de auto-modificação utilizado nas propostas da linguagem AdaptCode, do autômato de execução adaptativo e do ambiente de execução.

Esse tópico procura descrever esse pseudo-modelo, com o intuito de destacar as proposições que originaram o modelo, bem como o seu funcionamento. Destaca-se que a descrição apresentada não prima por completeza ou formalização do modelo. No entanto, fornece uma ilustração da equivalência do poder de expressão do ambiente de execução proposto e da máquina de Turing.

Para descrever esse ambiente de execução para códigos adaptativos baseado na máquina de Turing, faz-se necessário definir:

- Como especificar as operações suportadas pela linguagem e que podem ser utilizadas pelo código adaptativo;
- Como representar o armazenamento de informações (dados do programa);
- Como declarar as instruções que formam um código;
- Como declarar um código;
- A lógica de controle da máquina de Turing que implementa o ambiente de execução.

5.1.1 Definindo as operações suportadas

Para que a máquina de Turing possa executar um determinado código é necessário que esta saiba interpretar as operações suportadas pela linguagem. Tais operações, em essência, representam alguma função computável que é suportada pela linguagem, como por exemplo: somar dois valores, realizar a operação lógica E, atribuir um valor a um determinado registrador.

Estas podem ser representadas por meio da máquina de Turing (LEWIS; PAPADIMITRIOU, 1981), dado que são obrigatoriamente funções computáveis

(premissa). Uma vez definida a máquina de Turing associada a cada instrução, duas possíveis abordagens podem ser utilizadas para adicionar suporte a tais operações:

- Adicionar a máquina de estado de cada uma das operações suportadas à máquina de estados do ambiente de execução, de forma que o ambiente suporte a execução das operações da linguagem (i.e. incorporar a declaração das máquinas de Turing das operações na máquina de Turing que define o ambiente de execução).
- Codificar as máquinas de Turing associadas às operações suportadas e armazená-las na fita da máquina que implementa o ambiente de execução. Por ser tratar de uma máquina universal, o suporte a tais instruções é feita por meio da execução das máquinas contidas na fita no momento em que se deseja executar a operação (opção adotada).

Para suportar a linguagem é necessário tratar todo o conjunto de instruções associado à linguagem. Para cada operação I_x (com x representado valores de 1 a n, onde n representa a última instrução ao organizar os elementos do conjunto de instruções segundo alguma ordem, como por exemplo, a lexicográfica baseada nos nomes das operações) é especificada uma máquina de Turing MTI_x, que realiza o processamento associado a operação em questão (por exemplo, se a instrução I_k define a soma de dois números inteiros, MTI_k implementa a computação responsável por somar dois números inteiros).

Após a especificação das máquinas acima mencionada, essas são organizadas em uma fita específica, segundo a seqüência $\{(I^1, MTI_1); (I^2, MTI_2); \ldots ; (I^n, MTI_n)\}$, onde I^x representa um identificador da instrução I_x, definida pela máquina MTI_x ($1 \leq x \leq n$).

Fazendo uma analogia com a arquitetura Intel® 32 bits, uma determinada máquina MTI_x pode ser vista como o microcódigo da operação I_x e I^x pode ser visto como o código de operação (*opcode*). Baseado nessa analogia, a esta fita é dado o nome de Fita de microcódigo – $F_{\mu code}$.

Um exemplo de codificação da seqüência que declara uma operação é feita por meio de repetições de cadeias do tipo $\alpha I^x \beta MTI_x \alpha$, onde: α e β são caracteres delimitadores (para facilitar o *parsing* da cadeia), MTI_x é uma cadeia de símbolos que representa a codificação da declaração de máquina de Turing associada a

operação I_n e I^n (seqüência de n símbolos I) representa um indexador (código de operação) da máquina MTI_n. Esse exemplo encontra-se ilustrado na Figura 23 (o símbolo $B_\#$ é utilizado para demarcar o fim da cadeia à esquerda).

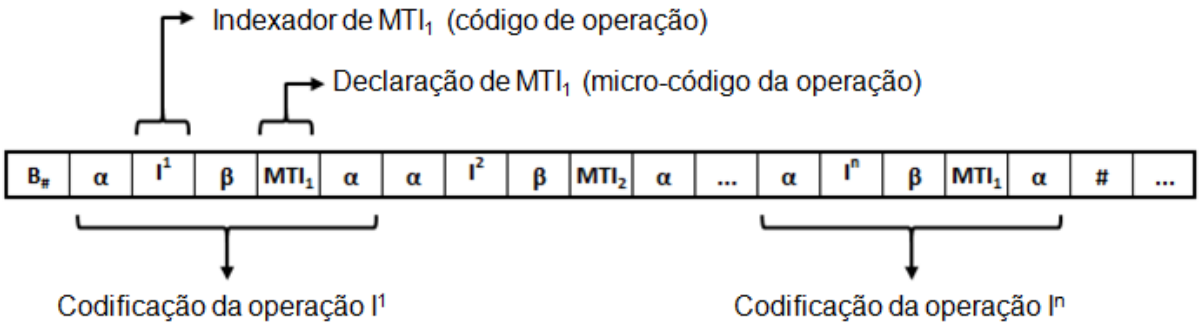


Figura 23 - Exemplo de fita $F_{\mu code}$.

5.1.2 Definindo a armazenagem de dados

Outro ponto importante a ser definido nesse ambiente de execução é a forma de armazenamento dos dados (no sentido de variável ou constante) de um programa. Como na máquina de Turing o armazenamento é feito por meio da fita, nada mais natural do que utilizá-la para armazenar os dados.

Por questões de organização do modelo (visando facilitar a localização de um determinado dado), bem como manter a clareza deste, é utilizado uma fita específica para armazenar dados, denominada F_{dados}.

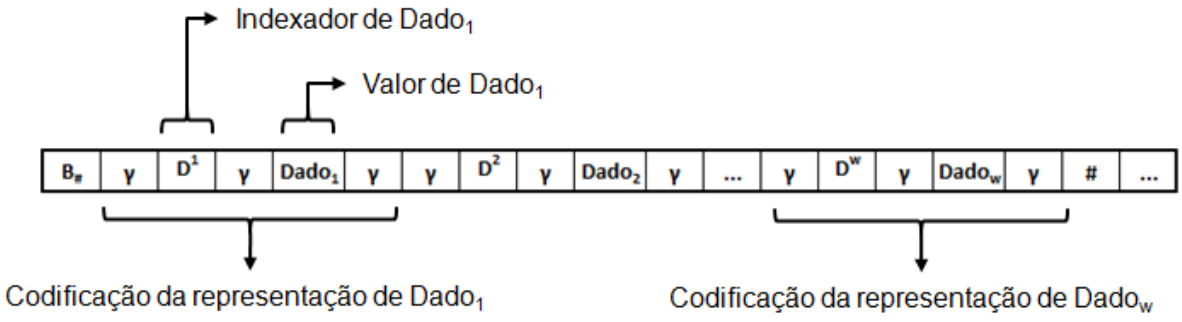


Figura 24 – Exemplo de Fita F_{dados}.

Os dados são armazenados por meio de uma codificação sobre um alfabeto previamente definido (como exemplo, o unário ou binário), permitindo assim a correta utilização por todas as máquinas declaradas na fita $F_{\mu code}$. Não obstante, cada dado é univocamente indexado, formando assim um esquema de endereçamento dos dados.

Um exemplo de possível codificação da fita F_{dados} (ilustrado na Figura 24) é composto por uma seqüência finita de cadeias do tipo $\gamma D^K \gamma Dado_k \gamma$, onde $Dado_k$ consiste em uma cadeia que representa o valor de um determinado dado, D^K (seqüência de k símbolos D) representa o indexador do dado $Dado_k$ e γ é um símbolo utilizado como delimitador da cadeia.

5.1.3 Representação de uma instrução

O código de um programa é formado por uma seqüência de instruções. Cada instrução é formada pelo código de uma operação suportada pela linguagem e dos dados a serem utilizados pela operação.

Este tópico ilustra como utilizar uma instrução nesse ambiente de execução (i.e. como instanciar uma operação com os devidos operandos – exemplo: a instrução `mov eax,5` é uma forma de utilizar a operação de movimentação de dados `mov`).

Neste ambiente de execução, a instrução pode ser resumida a executar uma das máquinas de Turing declaradas na fita $F_{\mu code}$ (operações definidas) utilizando como entrada um ou mais dados existentes na fita F_{dados} (operando).

Com intuito de preservar adequadamente os dados contidos na fita F_{dados}, bem como servir de apoio para a execução das máquinas declaradas em $F_{\mu code}$ é utilizado outra fita, denominada F_{rasc}, como rascunho de operação. Portanto, os dados utilizados por uma determinada instrução são copiados para essa fita, que funciona como fita de entrada durante a execução da operação associada à instrução.

Após a execução da máquina, caso exista a necessidade de armazenar algum dos dados contidos na fita F_{rasc}, esses serão copiados para F_{dados}.

Por exemplo, caso se deseje somar dois números ($dado_1$ e $dado_2$) e supondo que a instrução de soma seja indexada por I^3, antes de executar a máquina de Turing

associada a I^3, $dado_1$ e $dado_2$ são copiados para F_{rasc}, sendo essa fita utilizada como entrada e saída para a máquina MTI_3. Após a resolução do algoritmo (soma de inteiros), o resultado a ser armazenado é transportado para a fita F_{dados}.

A execução de uma instrução pode ser informalmente resumida como uma "chamada" a uma das máquinas de Turing declaradas na fita $F_{\mu code}$, parametrizada por meio de operandos transferidos para a fita F_{rasc}. Essa "chamada" de máquina de Turing pode ser feita, por exemplo, utilizando cadeias de símbolos que obedecem ao um dos seguintes padrões:

- $\Lambda I_k \Lambda$ – para os casos que a operação não utiliza nenhum operando. Exemplo, instrução `nop`;
- $\Lambda I_k \Xi D^x \Lambda$ – para casos que a operação utiliza 1 operando. Exemplo, instrução `call`;
- $\Lambda I_k \Xi D^x \Xi D^y \Lambda$ – para casos que a operação utiliza 2 operandos. Exemplo, instrução `add`.

Para os padrões acima citados, tem-se que: I_k é o indexador da máquina de Turing a ser executada (código da operação), D^x, D^y são indexadores de algum dado contido na fita F_{dados} (endereços dos dados a serem utilizados pela operação) e Λ, Ξ são símbolos delimitadores.

É possível notar semelhanças entre os padrões da cadeia utilizados e o formato das instruções associadas a arquitetura Intel® 32bits (DETMER, 2001) (INTEL, 2007) (INTEL N-Z, 2007). Esses últimos serviram de base para o estabelecimento do padrão de cadeias acima citado.

Destaca-se, por fim, que por questões de simplificação, nesse exemplo não foi definido processos de endereçamentos distintos, como imediato e indireto, adotando apenas o "endereçamento direto". Este modelo de endereçamento é chamado de "endereçamento direto" porque o indexador se comporta de maneira análoga à posição de memória do endereçamento direto definido na arquitetura Intel® 32 bits.

5.1.4 Representação de um código

Definido como instanciar uma instrução, o próximo passo da descrição desse ambiente de execução consiste em como representar o código.

Conforme discutido anteriormente, um código pode ser visualizado como uma seqüência ordenada de instruções. Para representar tal seqüência ordenada em uma máquina de Turing, faz-se necessário o uso de dois parâmetros.

O primeiro é um indexador responsável por indicar a posição de uma determinada instrução dentro da seqüência que define o código. Esse indexador desempenha o mesmo papel do endereço da instrução na arquitetura Intel® 32 bits.

O segundo é o indicador de próximo elemento. Tal parâmetro possui funcionalidade similar ao ponteiro de próximo elemento de uma lista ligada (LANGSAM; AUGENSTEIN; TENENBAUM, 1996), indicando a próxima instrução a ser executada. É responsável por manter a ordenação das instruções que formam o código.

Portanto, nesse esquema, o código é armazenado em uma fita dedicada a tal objetivo, denominada F_{code}. Para controlar a execução das instruções, outra fita, denominada $F_{cinstrução}$, é utilizada. Essa fita armazena o índice da instrução corrente, desempenhando a mesma função do registrador contador de instruções.

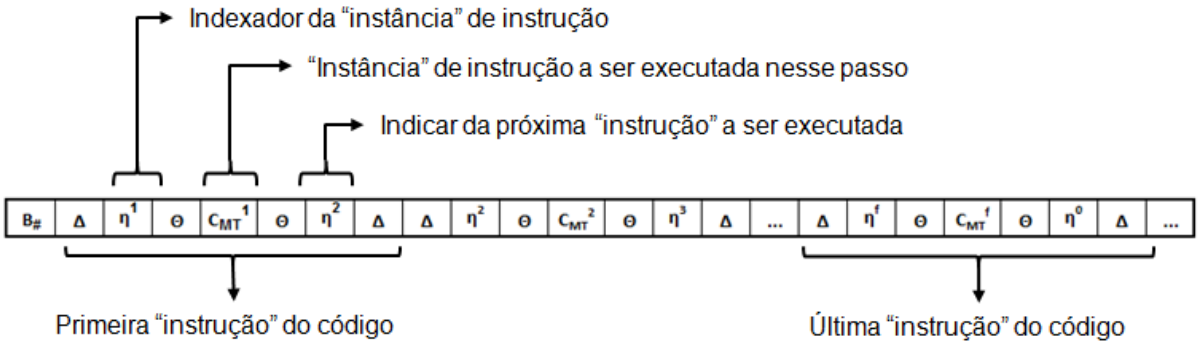


Figura 25 – Exemplo de Fita F_{code}.

O código do programa a ser executado pode ser representado por meio de uma seqüência de cadeia do tipo $\Delta\eta^{n}\Theta C_{MT}{}^{n}\Theta\eta^{k}\Delta$ (ilustrado na Figura 25), onde: Δ, Θ representam caracteres delimitadores; η^{n} representa a posição da instrução no

programa (“endereço” da instrução, exemplo: primeira instrução, segunda, última); C_{MT}^{n} (descrita em 5.1.3) é uma cadeia associada a descrição de uma instrução (ressalta-se que o índice n é referência a posição da instrução dentro do código e não a qual instrução será executada); e η^{k} representa a posição da próxima instrução a ser executada.

Este esquema é similar ao de composição de máquinas de Turing (LEWIS; PAPADIMITRIOU, 1981). No entanto, em vez das máquinas de Turing estarem diretamente ligadas, nesse esquema existe uma seqüência de índices, cada qual invocando uma determinada máquina de Turing e explicitando a próxima máquina a ser executada.

5.1.5 Lógica de funcionamento

Conforme visto no capítulo 2, um programa pode ser logicamente dividido em área de código e área de dados. Nesse modelo, essa divisão lógica é aproveitada, sendo o código do programa armazenado na fita F_{code}, e os dados do programa armazenados na fita F_{dados}, antes da execução do programa pelo ambiente. Portanto, o programa a ser executado por esse ambiente encontra-se explicitado nas fitas F_{code} e F_{dados}, sendo estas duas fitas a entrada a ser fornecida para a máquina de Turing universal que executa códigos adaptativos.

Esta execução é suportada pelas fitas: $F_{\mu code}$, que armazena as definições semânticas de cada operação (e nesse modelo não sofrerá alteração); $F_{cinstrução}$, responsável por controlar a informação da instrução corrente a ser executada; e F_{rasc}, responsável por servir de armazenamento temporário de informações associadas à execução da operação.

Algoritmo 2: Lógica de Controle

Entrada: Programa adaptativo codificado nas fitas F_{code} (área de código do programa) e F_{dados} (área de dados).

Condições iniciais:

- $F_{cinstrução}$ apontando para a primeira instrução de F_{code}.
- F_{rasc} em branco.
- $F_{\mu code}$ contendo a declaração de todas as instruções associada ao ambiente de execução.
- Para todas as fitas, o ponteiro encontra-se no símbolo $B_{\#}$ (primeiro símbolo de cada fita).

Passos de execução:

1. Ler o valor armazenado em $F_{cinstrução}$ (determinação da instrução corrente).
2. Localizar em F_{code} a instrução associada ao indexador lido no passo 1 (descobrir qual a instrução será executada).
3. Em F_{code}, ler o indexador da máquina de Turing (código de operação) a ser executada.
4. Localizar em $F_{\mu code}$ a declaração da máquina de Turing associada ao indexador lido (código de operação) no passo 3 (buscar o algoritmo que descreve a função da instrução a ser executada).
5. Caso a instrução a ser executada, especificada na fita F_{code}, referencie operandos (dados) armazenados da fita F_{dados}, para cada referência deve-se realizar os passos a e b descritos a seguir:
   a. Localizar, na fita F_{dados}, o indexador do operando a ser utilizado;
   b. Copiar tal indexador e o seu valor para a fita F_{rasc}.
6. Executa-se a máquina de Turing associada ao indexador lido no passo 3, utilizando como entrada os dados existentes na fita F_{rasc} (execução efetiva da instrução corrente). Esta execução deve prever, caso necessário, o salvamento do resultado da operação na fita F_{dados}. Para o caso do código de operação *halt*, o ambiente atinge o estado de *halt* (término da execução/computação).
7. Após a execução, o ambiente lê, na fita F_{code}, o valor do índice da próxima instrução a ser executada, copiando-o para a fita $F_{cinstruções}$. Caso a instrução que foi executada no passo 6 seja uma instrução que force o desvio de fluxo, o indexador relacionado a instrução apontada pelo desvio (passado como operando) será escrito na fita $F_{cinstruções}$. Esse passo descreve a obtenção do endereço da próxima instrução a ser executada.
8. Retorna ao passo 1, voltando os ponteiros de todas as fitas para as posições adequadas (símbolo $B_{\#}$ no inicio de cada fita) e limpando a fita F_{rasc}.

Para completar a descrição dessa máquina, é necessário explicitar a lógica de operação. Essa lógica, essencialmente, consiste nos seguintes passos detalhados em Algoritmo 2.

Para esclarecer o funcionamento desse ambiente, o processo da execução de uma determinada instrução do código descrito em F_{code} será ilustrado, utilizando os conteúdos das fitas representados na Figura 26 (para este exemplo, o ponteiro de uma determinada fita é simbolizado pelo campo preenchido na correspondente fita).

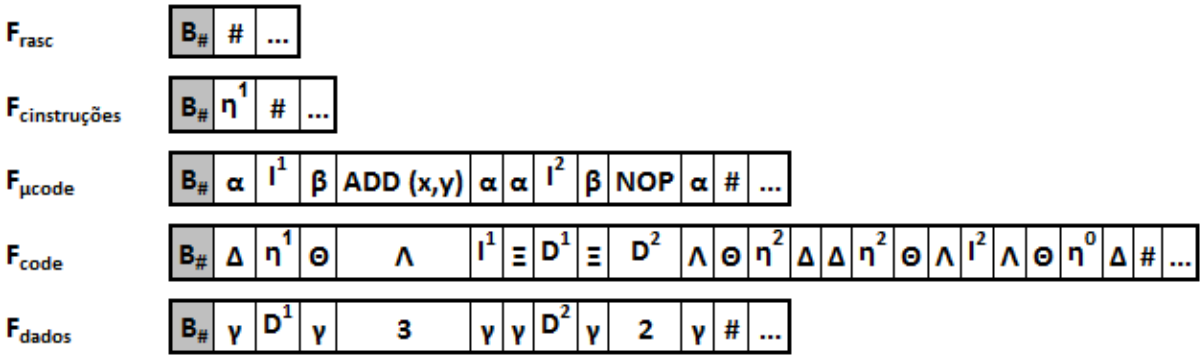


Figura 26 – Configuração inicial das fitas (exemplo).

Passo 1 - Obtenção do “endereço” da instrução a ser executada. Esse “endereço” consiste no índice associado à instrução do código a ser executada, explicitado na fita $F_{cinstruções}$. Para esse exemplo, será executado a instrução cujo indexador é η^1, como destacado na Figura 27.

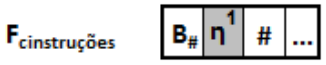


Figura 27 – Passo 1: determinando a instrução a ser executada.

Passo 2 – Localização da instrução a ser executada dentro do código. Nessa etapa, a lógica de funcionamento busca a cadeia que descreve a instrução do código a ser executada. Por meio do indexador (“endereço”) obtido no passo 1, a cadeia que descreve tal instrução é localizada em F_{code} (nesse exemplo, busca-se a cadeia associada ao índice η^1 na Fita F_{code} – ilustrado na Figura 28).

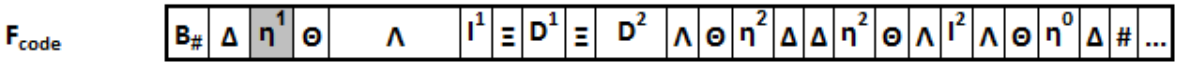


Figura 28 – Passo 2: localizando a instrução a ser executada.

Passo 3 – Determinando o "*opcode*" a ser executado. Esse passo busca determinar o código da operação a ser executada por essa instrução, por meio da localização do índice associado a uma das declarações de máquina de Turing feita na fita $F_{\mu code}$. No caso, a máquina de Turing a ser ativada é aquela cuja declaração está associada ao símbolo I^1 – conforme explicitado pela cadeia $\Delta\eta^1\Theta\Lambda I^1\Xi D^1\Xi D^2\Lambda\Theta\eta^2\Delta$, destacada na Figura 29).

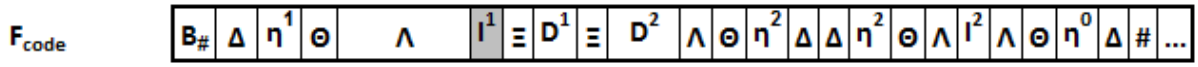


Figura 29 – Passo 3: índice da máquina de Turing universal a ser executada

Passo 4 – Localizar a declaração da operação (máquina de Turing) a ser executada. Procura-se a declaração das operações associadas à instrução I^1 dentre todas as máquinas de Turing declaradas em $F_{\mu code}$ (localiza a seqüência $\alpha I^1\beta MTI_1\alpha$ 4 existente na fita $F_{\mu code}$, exibida na Figura 30).

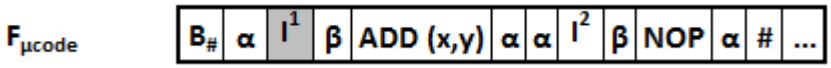


Figura 30 – Passo 4: Encontrando a declaração da máquina de Turing universal a ser executada.

Passo 5 – Posicionar o ponteiro da fita $F_{\mu code}$ adequadamente para futura execução da máquina de Turing. Dado que no passo anterior foi encontrado a cadeia de símbolos associado a máquina de Turing a ser executada, deve-se posicionar o ponteiro de $F_{\mu code}$ para a posição que inicie a declaração da máquina de Turing responsável por efetivamente executar a operação (no caso, a declaração de tal máquina é representada por `ADD (x,y)` – conforme ilustrado na Figura 31. Tal máquina é responsável por efetuar a soma o valor de `x` e `y`, armazenando o resultado em `x`).

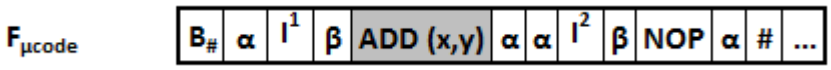


Figura 31 – Passo 5: Posicionando ponteiro para execução da máquina de Turing universal `ADD (x,y)`.

Passo 6 – Verificar a existência de operandos na instrução a ser executada. Antes de efetivamente executar a máquina `ADD(x,y)`, é obrigatório verificar se a instrução do código que esta sendo executada não possui operandos (que serão utilizados ao longo da execução dessa máquina).

Para tanto, a lógica de controle consulta a existência de operandos na cadeia da fita F_{code} que descreve a instrução a ser executada nesse passo. Ao analisar a cadeia que descreve a instrução associada ao símbolo η^1 (seqüência de símbolos ($\Delta\eta^1\Theta\Lambda I^1\Xi D^1\Xi D^2\Lambda\Theta\eta^2\Delta$ localizado em F_{code}), observa-se que essa instrução referencia dois operandos (D^1 e D^2). Seguindo a ordem de varredura da cadeia (da esquerda para a direita), o primeiro operando encontrado é o D^1, como ilustrado na Figura 32.

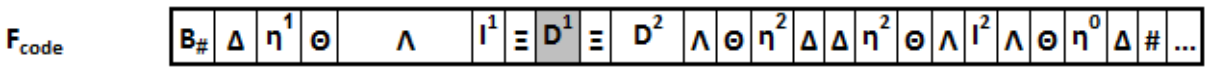


Figura 32 – Passo 6: Buscando operandos da instruções.

Passo 7 – Copiar o operando contido na fita F_{dados} para a fita F_{rasc}, para ser utilizado durante a execução da máquina `ADD`. Por meio do indexador do dado especificado na instrução (símbolo D^1), localiza-se o valor deste na fita F_{dados}. Após a busca desse valor, a cadeia que descreve tal dado é copiado para a fita F_{rasc} , conforme ilustrado na Figura 33.

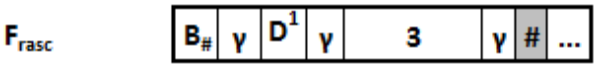


Figura 33 – Passo 7: Copiando dado D^1 para a fita F_{rasc}.

Passo 8 – Repetição do passo 7 para o segundo operando (D^2). Após a execução dos passos 7 e 8, a fita F_{rasc} adquire a configuração indicada na Figura 34.

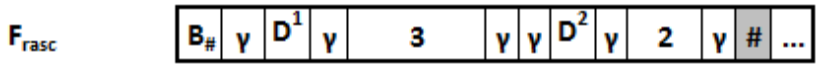


Figura 34 – Passo 8: Fita Frasc após a copia dos dados D^1 e D^2.

Ao término do tratamento dos operandos associado à instrução e com a fita de entrada preparada, a máquina de Turing universal que calcula `ADD (x,y)` pode ser executada.

Passo 9 – Execução da máquina `ADD (x,y)`. Nesse passo o ambiente de execução trabalha efetivamente como se fosse uma máquina de Turing universal, executando a declaração da máquina de Turing `ADD` e utilizando como entrada a fita F_{rasc}.

Ao término desse passo pode-se afirmar que a instrução foi efetivamente executada. A máquina executada durante o passo 9 deve tratar adequadamente o armazenamento da saída, copiando o resultado armazenado na fita F_{rasc} para a fita F_{dados}. Nesse exemplo, após a execução, o valor de saída (resultado da soma) é armazenado no "endereço" indexado por D^1, conforme indicado na Figura 35.

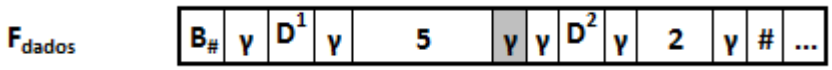


Figura 35 – Passo 9: Fita F_{dados} após a execução da instrução η^1.

Passo 10 – Determinação da próxima instrução a ser executada. O indexador ("endereço") da próxima instrução a ser executada encontra-se declarada na seqüência que descreve a instrução em execução (fita F_{code}). No caso da instrução η^1, a próxima instrução a ser executada é a η^2, conforme esquematizado na Figura 36.

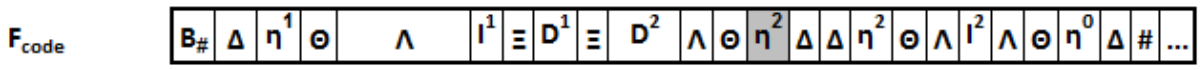


Figura 36 – Passo 10: Determinando a próxima instrução a ser executada (exceto para casos de desvio).

Passo 11 – Atualização do valor da fita $F_{cinstruções}$. Para terminar um passo da computação dessa máquina, o valor armazenado da fita $F_{cinstruções}$ (indexador da

instrução em ou a ser executada) é atualizado, como indicado na Figura 37. No caso da instrução em execução for um salto ou um desvio (que irá acontecer), o valor dessa fita pode ser atualizado pela própria máquina declarada na fita $F_{\mu code}$, sendo o índice contido na fita F_{code} desprezado.

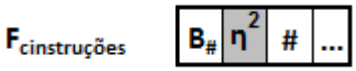


Figura 37 – Passo 11: Atualizando “contador de instruções”.

Adicionalmente, no passo 11 a fita F_{rasc} é limpa (voltando a configuração inicial vazia).

Após a execução dessa instrução, as fitas possuem a configuração descrita na Figura 38.

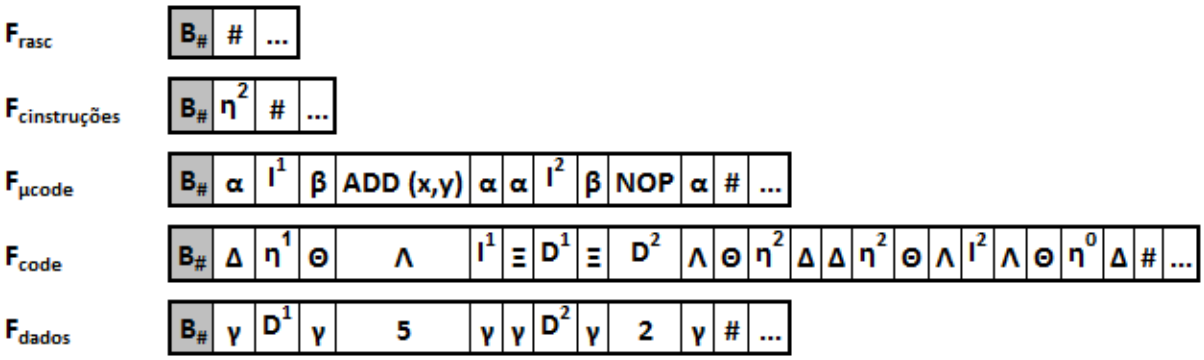


Figura 38 – Configuração das fitas após a execução da instrução indexada por η^1 (exemplo).

5.1.6 Modelo de alteração de código

O princípio de modificação do código no qual essa dissertação se baseia, consiste na substituição de uma seqüência de instruções justapostas por outra cuja imagem encontra-se armazenada na área de dados. Tais instruções justapostas são representadas pelas seqüências de símbolos do tipo $\Delta\eta^n\Theta C_{MT}{}^n\Theta\eta^k\Delta$ da fita F_{code}, cujo n vai de um determinado valor x a outro y, abrangendo todos os números

inteiros entre x e y (adicionalmente, x e y devem ser endereços de instruções existentes, para o correto funcionamento).

Portanto, modificar o código nesse ambiente resume-se em criar uma máquina de Turing na fita $F_{\mu code}$ que edite convenientemente as seqüências $\Delta\eta^{n}\Theta C_{MT}{}^{n}\Theta\eta^{k}\Delta$ existentes na fita F_{code}. Dado que tanto o código quanto os dados estão armazenados em fitas, estes podem ser convenientemente editados

Pelo modelo de alteração de código definido na linguagem AdaptCode, uma seqüência de símbolos $\Delta\eta^{n}\Theta C_{MT}{}^{n}\Theta\eta^{w}\Delta$ (com $x \leq n \leq y$) é substituída por outra seqüência de símbolos $\Delta\eta^{k}\Theta C_{MT}{}^{n}\Theta\eta^{l}\Delta$ existente na fita F_{dados}, ressaltando que k é tratado adequadamente ao ser inserido na fita F_{code}, com intuito de evitar sobreposições com valores existentes.

Para ilustrar esse processo, será adotada a seguinte simplificação de notação: $\Delta\eta^{n}\Theta C_{MT}{}^{n}\Theta\eta^{k}\Delta$ é equivalente a $(n, C_{MT}{}^{n}, k)$, onde n é o indexador da instrução, $C_{MT}{}^{n}$ é a "chamada" a uma determinada máquina de Turing declarada em $F_{\mu code}$ e k é o indexador da próxima instrução a ser executada.

Considere ainda que: (i) a fita F_{code} é formada por uma seqüência do tipo $\{ (1, C_{MT}{}^{1}, 2) ; (2, C_{MT}{}^{2}, 3) ; (3, C_{MT}{}^{3}, 4) ; ... ; (x, C_{MT}{}^{x}, x+1) ; ... ; (y, C_{MT}{}^{y}, y+1) ; .... (w, C_{MT}{}^{w}, 0)\}$; (ii) o bloco adaptativo a ser modificado corresponde ao trecho $\{ (x, C_{MT}{}^{x}, x+1) ; ... ; (y, C_{MT}{}^{y}, y+1) \}$; e (iii) $C_{MT}{}^{2}$ está associada a utilização da instrução adaptativa que deseja substituir o trecho indicado em (ii) pelo trecho $\{ (k, C_{MT}{}^{k}, k+1) ; (k+1, C_{MT}{}^{k+1}, 0) \}$.

Ao executar a instrução associada à $C_{MT}{}^{2}$, inicialmente o trecho a ser substituído é removido, sendo que a cadeia F_{code} passa a ser representada por $\{ (1, C_{MT}{}^{1}, 2) ; (2, C_{MT}{}^{2}, 3) ; (3, C_{MT}{}^{3}, 4) ; ... ; (x-1, C_{MT}{}^{x-1}, x) ; (y+1, C_{MT}{}^{y+1}, y+2) ; .... (w, C_{MT}{}^{w}, 0)\}$. Em seguida é feita a montagem e inserção do trecho, resultando na cadeia $\{ (1, C_{MT}{}^{1}, 2) ; (2, C_{MT}{}^{2}, 3) ; (3, C_{MT}{}^{3}, 4) ; ... ; (x-1, C_{MT}{}^{x-1}, k') ; (k', C_{MT}{}^{k'}, k'+1) ; (k'+1, C_{MT}{}^{k'+1}, y+1) ; (y+1, C_{MT}{}^{y+1}, y+2) ; .... (w, C_{MT}{}^{w} , 0)\}$.

Um aspecto curioso desse mecanismo remete a característica da própria declaração das instruções da linguagem ser armazenada em uma fita ($F_{\mu code}$), o que torna possível criar e remover instruções ao longo da execução de um código, contanto que a operação que realize tal atividade seja previamente definida. No entanto, esse tópico não será tratado nessa dissertação.

5.2 AMBIENTE DE EXECUÇÃO DA LINGUAGEM ADAPTCODE

Como dito na introdução desse capítulo, o ambiente descrito no item 5.1 como o serve como base didática para a compreensão do ambiente de execução da linguagem AdapCode. No entanto, o ambiente proposto não utiliza máquina de Turing pelos seguintes motivos: (i) complexidade de descrição e implementação dessa máquina, dado a simplicidade das operações definidas na máquina de Turing; e (ii) o foco é construir essa máquina sobre a arquitetura de computadores domésticos.

Como conseqüência, os processos definidos no ambiente previamente descrito foram adaptados de tal forma a permitir a construção sobre outra arquitetura. Em termos de implementação, o ambiente de execução associado à linguagem para programação adaptativa consiste em uma máquina virtual que faz o interfaceamento da execução do programa adaptativo com o computador doméstico. Em termos de representação do código, essa adaptação reflete-se no autômato de execução adaptativo.

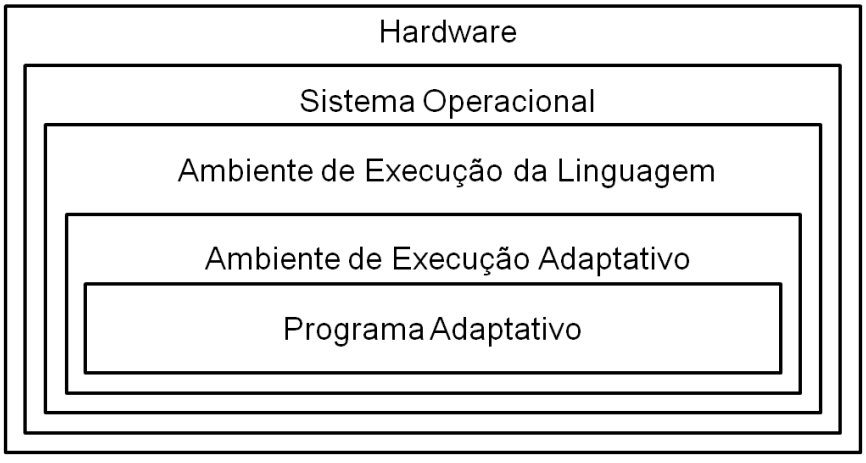


Figura 39 – Hierarquia de ambientes

A implementação desse ambiente de execução é feita sobre uma determinada linguagem de programação, o que resulta em um esquema de hierarquia de ambientes apresentado na Figura 39, onde: (i) o programa adaptativo é executado

sobre um ambiente de execução adaptativo; (ii) o ambiente de execução adaptativo é executado no ambiente de execução da linguagem em que foi implementado; (iii) o ambiente de execução da linguagem é um aplicativo que esta sendo executado sobre um sistema operacional; e (iv) o sistema operacional gerencia e está sendo executado sobre um determinado *hardware*.

Opcionalmente, o ambiente de execução que está sendo descrito nesse tópico pode ser construído: diretamente sobre um *hardware*; como parte de um sistema operacional; ou como um ambiente a ser executado sobre um sistema operacional. Eventualmente, este ambiente pode ser construído sobre um possível *hardware* adaptativo, dado que já existem alguns modelos de *hardware* com capacidade de se auto-reconfigurarem, como em (KEUNG; TYAGI, 2006), onde é proposto um *hardware* para a execução de *self-modifying finite automata* (RUBINSTEIN; SHUTT, 1995).

5.2.1 Arquitetura do ambiente de execução

O ambiente de execução proposto nessa dissertação, de maneira independente do nível em que é executado, é constituído de três blocos lógicos distintos: o de controle e interpretação, o de dados e o de código (Figura 40).

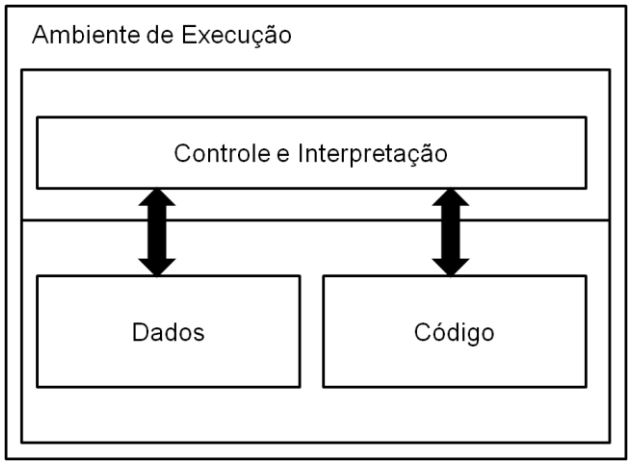


Figura 40 – Arquitetura do Ambiente de Execução

O bloco de interpretação e controle é responsável pelo controle do processo de execução do código, fornecendo funcionalidades necessárias para permitir a correta computação de um programa adaptativo. Dentre tais funcionalidades (aspectos de implementação), destacam-se:

- Interpretação de instruções: responsável pela execução propriamente dita das instruções da máquina (como por exemplo, uma instrução de soma). Ressalta-se que o mecanismo de modificação do código é uma instrução e, conseqüentemente, o mecanismo adaptativo encontra-se especificado neste módulo. O mecanismo adaptativo é responsável por prover operações de apoio à alteração do código, tais como:
    - Processar e executar as operações relacionadas à adaptatividade (`mark`, `emark`, `adapt` – que implementam o mecanismo de alteração do código executável), tratar casos de erros de alteração de código (exemplo: alterar um trecho inexistente);
    - Tratar os casos excepcionais (exemplo: alterar o trecho em execução);
    - Manter as informações que forem necessárias acerca dos trechos de código executável passíveis de serem alterados.
- Proteção à área de programa: protege o código executável do programa, permitindo apenas que a instrução de alteração adaptativa de código (`adapt`) modifique o seu conteúdo.
- Mecanismo de computação de autômato de execução adaptativo: responsável por implementar o algoritmo associado a computação do autômato de execução adaptativo. Pode-se citar como função: conhecer o estado corrente (conceito similar ao $F_{cinstruções}$), a sub-máquina ativa, armazenar a pilha de sub-máquinas, executar ações semânticas sobre o autômato e executar as transições.
- Interfaceamento com Sistema Operacional / Hardware: responsável por converter as operações do ambiente virtual em operações do ambiente físico.

Ao se fazer um paralelo com o modelo baseado em máquina de Turing, pode-se afirmar que este módulo possui as funções das fitas F_{rasc}, $F_{cinstrução}$, $F_{\mu code}$ e o da lógica de controle.

Os módulos de dados e código são responsáveis pelo gerenciamento de memória. Portanto, são responsáveis por controlar o processo de alocar memória e de reaver áreas de memória alocada, tanto para dados quanto para programas. Outra finalidade desta operação é a de controlar a quantidade de memória utilizada por um programa, interrompendo-o caso ultrapasse um tamanho máximo pré-estabelecido (preocupação inexistente em programas não-adaptativos, ao menos para a área de código executável). Registradores e dados de apoio são gerenciados por este módulo.

O módulo de dados é responsável por tratar o armazenamento das variáveis e das estruturas de dados utilizadas por um determinado programa adaptativo. Estabelece as rotinas necessárias para criar, remover, acessar ou gravar informações na área de dados de um programa adaptativo. Sua principal função é armazenar as variáveis do(s) programa(s) adaptativo(s) em execução. É o equivalente a função da fita F_{dados} (mas não aos dados contidos nessa fita).

Ainda em relação ao módulo de dados, a definição desse ambiente não especifica nenhum modelo de armazenagem de informação, deixando esse parâmetro livre para cada implementação. Como sugestão de modelo de armazenamento, pode-se citar: o armazenamento direto na memória, lidando com aspecto de endereçamento de memória como as posições a serem utilizadas e o tamanho de palavras; o mapeamento dos dados associados ao ambiente de execução em variáveis definida pela linguagem que implementa o ambiente.

Por fim, o módulo de código é o responsável pelo armazenamento do código adaptativo (especificamente, armazenar o autômato de execução adaptativo que representa o código a ser executado). Estabelecendo um paralelo com o ambiente de execução baseado em máquina de Turing, este módulo é o equivalente a Fita F_{code} (mas não aos dados contidos nessa fita).

Apesar de definir que o programa deve ser codificado como um autômato de execução adaptativo, a proposta desse ambiente não especifica a forma de implementar tal autômato (como por exemplo os tipos de estrutura de dados necessários).

5.2.2 Modelo de execução

A execução de um programa adaptativo nesse ambiente pode ser sub-dividida em dois momentos distintos.

O primeiro momento refere-se à montagem do programa adaptativo. Nesta etapa o código escrito na linguagem de programação AdaptCode é montado de tal forma que o código é representado por um autômato de execução adaptativo, e que as informações armazenados na área de dados são convertidos para a representação utilizada pelo ambiente. Durante essa etapa, informações importantes sobre o código, como endereços e *labels* de saltos, os nomes dos blocos adaptativos declarados, dentre outros, são aprendidos pelo ambiente de execução (aprendizado o qual é importante para o correto funcionamento da execução do programa).

Este processo é ilustrado na Figura 41. Observa-se que um programa adaptativo, cujo código está escrito na linguagem AdaptCode e, portanto, não apresenta clara divisão lógica entre código e dados, ao sofrer a montagem é dividido em um código adaptativo, representado por um autômato de execução adaptativo, e em uma área de dados, representado por um formato interno.

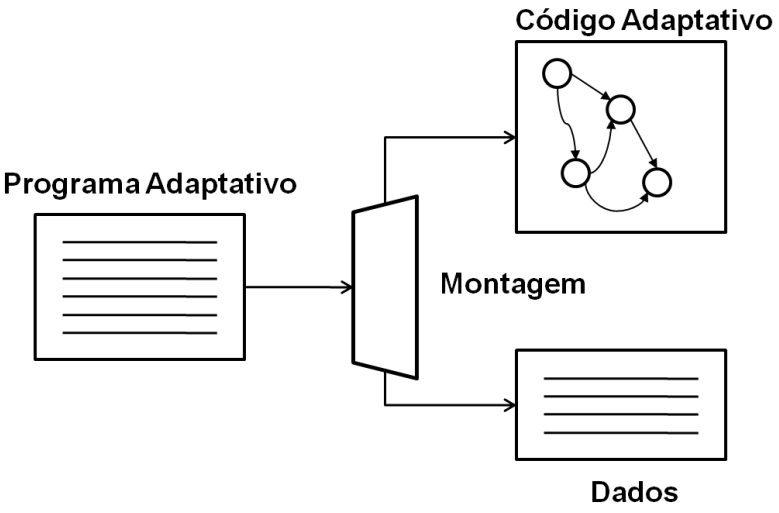


Figura 41 – Montagem de um programa adaptativo.

Após a etapa de montagem, entra-se na fase de execução. Como o próprio nome sugere, nesta etapa ocorre efetivamente à execução do código.

O funcionamento do ambiente de execução é similar ao funcionamento do autômato de execução adaptativo descrito em 4.2.2 e ao funcionamento do modelo de execução baseado em máquina de Turing detalhado no item 5.1.

Algoritmo 3: Execução do código

Entrada: Programa adaptativo devidamente codificado (Autômato de execução adaptativo + dados).

Condições iniciais:

- Valores correntes de estado_corrente e sub_máquina_corrente são os associados ao estado inicial da sub-máquina inicial (i.e. ponto onde começa a computação) do autômato de execução adaptativo.

Passos de execução:

- Enquanto o estado_corrente e a sub_máquina_corrente forem válidos (i.e. são estados declarados de sub-máquinas declaradas) e não forem finais (instrução halt – estado final da sub-máquina inicial), executam-se as seguintes operações:
  a. O interpretador busca, no módulo de código, a instrução que está associada ao estado_corrente da sub_máquina_corrente do autômato de execução adaptativo em execução;
  b. O interpretador busca (no módulo de dados) os operandos que estão associados à instrução buscada no item (a).
  c. Interpreta-se a instrução.
  d. Caso seja necessário, armazena-se na área de dados o resultado da instrução interpretada no item (c).
  e. Transita-se para o próximo estado a ser executado.

Conforme descrito no algoritmo 3, a etapa de execução é formada por um *loop* de execução de instruções. Enquanto o estado for válido (declarado no conjunto de estados do autômato de execução adaptativo) e não for final (estado final da sub-máquina inicial, também chamada de principal), o interpretador realiza as seguintes operações:

- Busca a instrução associada ao estado corrente do autômato de execução adaptativo;

- Caso a instrução referencie algum dado, busca estes para serem utilizados pela computação dessa instrução;
- Interpreta (executa) a instrução. Nesse passo, caso seja necessário, atualiza a descrição do autômato de execução ou ativa a função adaptativa;
- Caso seja necessário, armazena o resultado da instrução na área de dados;
- Transita para o próximo estado do autômato de execução adaptativo.

Essa execução pode ser abortada caso alguma anormalidade seja detectada. Cita-se como exemplo de tal situação: entrar em um estado não declarado e a tentativa de interpretação de uma a instrução não declarada no conjunto de instruções.

5.2.3 Técnica de modificação de código

Para encerrar com o tópico do ambiente de execução, descreve-se como o mesmo permite a computação de programas adaptativos ao ser implementado sobre uma máquina que não prevê a modificação de código.

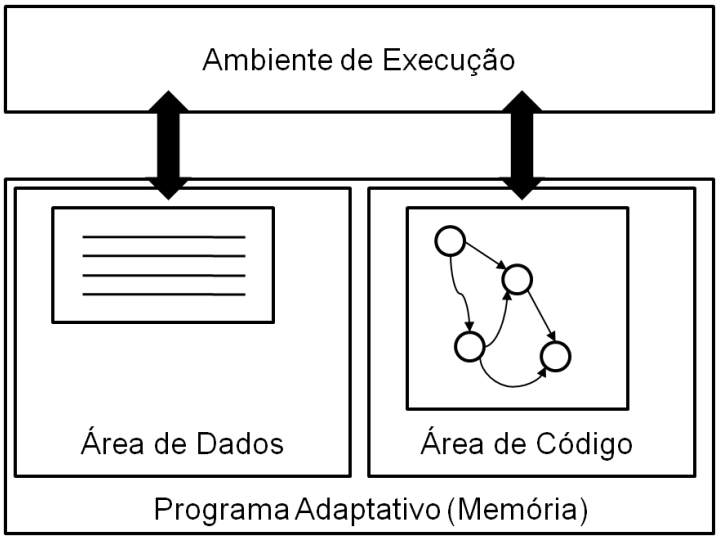


Figura 42 – Ambiente de execução - modelo de execução lógico

Ao analisar a execução de um código pela ótica do ambiente de execução, o programa adaptativo é constituído de duas áreas distintas de memória (código e

dados) – ilustrado na Figura 42, sendo que a área de código pode ser modificada apenas pela instrução `adapt`.

Nesse modelo a área de código é endereçada de maneira que o interpretador visualize o código como um autômato de execução adaptativo. Nesta situação, essa área de memória somente pode ser modificada em duas situações distintas.

A primeira refere-se ao momento de carregar o programa na memória. Imediatamente após a montagem do código, o código do programa adaptativo a ser executado é representado por meio de um autômato de execução adaptativo que é armazenado nessa área de memória.

A segunda refere-se a utilização da função adaptativa, invocada por meio da instrução `adapt`. Neste caso o ambiente de execução irá montar uma imagem contida na área de dados e irá modificar a área de código de tal forma que o trecho de código adaptativo indicado pela instrução `adapt` seja substituído pelo conteúdo dessa imagem devidamente montado.

A área de dados é tratada com um vetor de posições de memória cuja função é armazenar variáveis (sendo implementado o tratamento para armazenagem do tipo inteiro 32 bits e string).

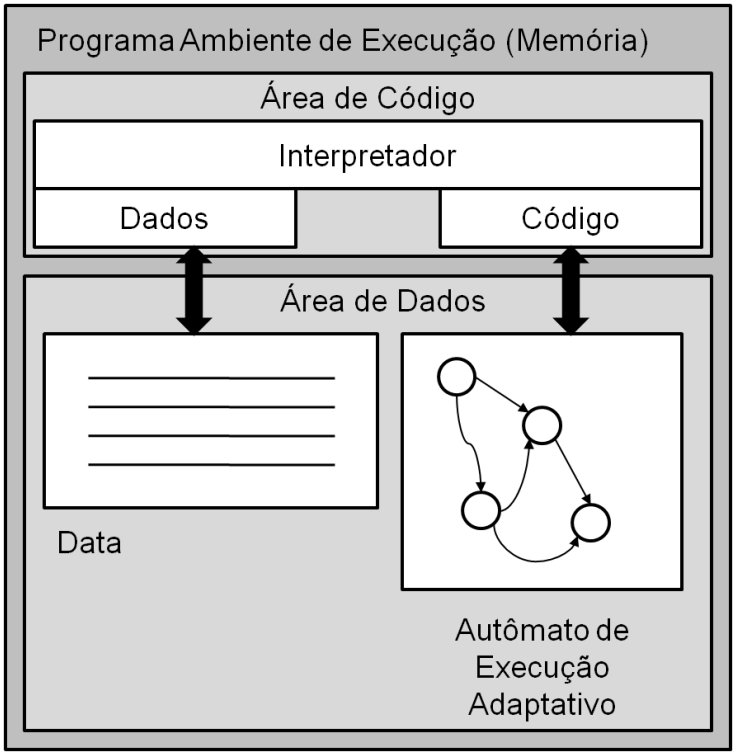


Figura 43 – Ambiente de execução - modelo de execução implementado (memória física)

Ao analisar esse ambiente por uma camada externa (por exemplo, pelo ponto de vista do sistema operacional ou do interpretador da linguagem no qual o ambiente foi escrito), pode-se visualizar que o ambiente de execução de códigos adaptativos mais o programa adaptativo tratam-se de um único programa não-adaptativo.

Nesse cenário (ilustrado na **Erro! Fonte de referência não encontrada.**(b)), o mbiente de execução forma o código a ser executado (área de código, que tipicamente não sofre alteração) enquanto os dados (data) e o código do programa adaptativo (autômato de execução adaptativo) são tratados como dados ou estruturas de dados associados ao programa ambiente de execução. Por ser considerado como área de dados, o autômato de execução adaptativo pode sofrer modificações livremente, o que permite o correto funcionamento do ambiente proposto.

O autômato de execução adaptativo é implementado por meio de três estruturas de dados:

- Uma lista ligada que armazena a estrutura de controle dos trechos de códigos adaptativos (i.e. armazena a declaração e informação de todos os trechos de códigos adaptativos declarados). Portanto, cada nó dessa lista representa a declaração de um trecho de código adaptativo e armazena as seguintes informações:
    - Nome associado ao trecho adaptativo e indexador fornecido pelo ambiente de execução.
    - Nome da sub-máquina no qual esse trecho foi criado.
    - Instrução de entrada e instrução de saída referente a esse trecho.
    - Informação do próximo elemento;
- Uma lista ligada que representa uma sub-máquina do autômato de execução adaptativo. Cada nó dessa lista representa uma instrução do procedimento associado a sub-máquina em questão e armazena as seguintes informações:
    - Número de estado
    - O código da instrução
    - O primeiro operador e o tipo do primeiro operador
    - O segundo operador e o tipo do segundo operador

    - As informações dos próximos elementos da lista. Neste caso, cada nó aponta para outros dois, sendo que um simboliza o caso de transição com o símbolo T e o outro de transição com o símbolo F.
- Uma lista ligada que representa o autômato de execução adaptativo. Cada nó dessa lista representa uma sub-máquina declarada do autômato de execução adaptativo em questão e armazena:
    - A informação do nome da sub-máquina
    - Uma lista ligada que modela a sub-máquina.
    - Informação de próximo elemento.

Essas estruturadas de dados foram definidas de forma a se aproveitar de duas características.

A primeira refere-se a armazenagem: independentemente do número de operandos de uma instrução, todas são armazenadas como se possuíssem dois operandos (instrução que ocupa maior quantidade de posições de memória), ou seja, todas as instruções ocupam o mesmo tamanho de memória. Apesar de resultar em um sub-aproveitamento da memória, simplifica o processo de gerenciamento da memória, uma vez que qualquer instrução pode ser escrita no espaço ocupado por outra instrução que está sendo apagada.

A segunda refere-se à forma de determinação da próxima instrução a ser executada, que é explicitada (informação contida no próprio nó). Essa característica, somada com a anterior, permite evitar a necessidade de realocação e de relocação do código. O preço de tal característica é o aumento de consumo de memória, uma vez que cada instrução armazena a informações da posição de outras duas instruções (essa implementação é extremamente similar ao esquema da fita F_{code} citado anteriormente).

Adicionalmente, para controlar os saltos e desvios, a cada sub-máquina do autômato de execução é associada uma lista de *labels* declarados (*labels* declarados no procedimento). Essa lista é responsável por associar os mesmos aos estados, permitindo o correto funcionamento das instruções de salto e desvios.

Como conseqüência desse modelo, existe a obrigatoriedade de trabalhar com um modelo de interpretação de instruções que estão armazenadas na área de dados (i.e. como uma linguagem interpretada, não compilada).

6 EXEMPLOS E CONSIDERAÇÕES

Esse capítulo apresenta a descrição de um exemplo simples e ilustrativo de código adaptativo escrito na linguagem AdaptCode. Seguido a esse exemplo, considerações a cerca da implementação de transdutores finitos adaptativos por meio dessa linguagem são descritas, com o intuito de explanar a escrita de procedimentos adaptativos em uma linguagem para programação adaptativa.
Por fim, são apresentados considerações sobre a linguagem em questão. A primeira descreve alguns aspectos adotados na implementação do protótipo que serve como prova de conceito. A segunda apresenta um estudo realizado em um protótipo anterior e apresentado no trabalho (PELEGRINI; NETO, 2008).

6.1 FATORIAL ADAPTATIVO

Para servir de apoio a explicação do processo de modificação de código associado à linguagem AdaptCode, bem como ilustrar o processo definido nos capítulos anteriores, este tópico apresenta um exemplo de código adaptativo.
Esse exemplo consiste na escrita da rotina fatorial utilizando métodos baseados na adaptatividade. A escolha desse exemplo é motivada pela simplicidade no entendimento do código e ao reduzido tamanho do código, o que facilita a compreensão do exemplo.
Por questões de organização, o exemplo foi dividido em três partes: a primeira apresenta o exemplo utilizando a linguagem de programação AdaptCode; a segunda desenvolve o mesmo via representação de autômato de execução adaptativo; e, por fim, a terceira parte apresenta considerações sobre tal exemplo.

6.1.1 Fatorial Adaptativo – Código AdaptCode

Para a criação de um código adaptativo capaz de calcular o fatorial de n, o funcionamento desse código foi abstratamente dividido em três fases distintas:

- Fase 1 – verificar o parâmetro n, com o intuito de garantir que n é um número inteiro maior que zero;
- Fase 2 – caso atenda ao critério especificado na fase 1, calcular o fatorial por meio de recursos de auto-modificação de código;
- Fase 3 – restaurar o código e retornar o valor calculado.

O código do procedimento fatorial é apresentando na Figura 44. O procedimento adotado na resolução do fatorial consiste em utilizar a auto-modificação de código para criar a série de multiplicações que calculam o fatorial do número n (fat(n) = $\prod_{x=1}^{n} x$ para todo n ≥ 1, e 1 para n igual a 0). Para calcular o produtório, nesse exemplo, foram criadas duas ações adaptativas com propósitos distintos de modificação de código.

```
FAT proc          //procedimento factorial
  pop n           //desempilha n (fat(n))
  mov eax,1       //move o valor 1 para eax
  cmp n,0         //compara se n é igual a zero
  jle X           //para os casos que n <= 0, desvia para o
                  //label (X), o que faz o procedimento
                  //retornar o valor 1.
  adapt 1,step    //adapta trecho 1 (chamada adaptativa)
  mark 2          //inicio do trecho adaptativo 2
  mark 1          //inicio do trecho adaptativo 1
  emark 1         //fim do trecho adaptativo 1
  emark 2         //fim do trecho adaptativo 2
label X           //label X
  mov res,eax     //armazena o valor de eax em res
  adapt 2,base    //adapta trecho 2
  ret             //retorna do procedimento
FAT endproc       //fim de procedimento
```

Figura 44 – Procedimento fatorial de n adaptativo (área de código)

A ação adaptativa relacionada à instrução do código `adapt 1,step` (responsável por modificar o trecho de código adaptativo `1`) é responsável por construir a seqüência de instruções associada ao produtório que calcula o fatorial. A área de dados apontada por `step` descreve a imagem do trecho de código responsável por multiplicar o resultado parcial do produtório (exemplo: n * (n-1)) pelo próximo valor

(exemplo: (n-2)) e, em seguida, decrementar n. Também descreve um teste que verifica a necessidade de outra auto-modificação do código (caso n seja maior que 0), por meio da instrução `adapt 1,step` e da definição de um novo trecho adaptativo, indexado por 1. O trecho de código a ser inserido por essa auto-modificação encontra-se descrito na Figura 45.

```
  mul eax,n      //eax ← eax*n
  dec n          //n ← n - 1
  cmp n,0        //compara se n é igual a 0
  jz Y           //desvia para Y se n == 0
  adapt 1,step   //adapta trecho 1 (chamada adaptativa)
  mark 1         //inicio do trecho adaptativo 1
  emark 1        //fim do trecho adaptativo 1
label Y          //label Y
  endp           //fim de trecho de código
```

Figura 45 – Dado step (área de dados)

Após o cálculo do fatorial, a ação adaptativa associada à instrução `adapt 2,base` (auto-modificação do trecho de código adaptativo `2`) remove todas as instruções inseridas pela aplicação da outra ação adaptativa e retorna o código para o seu estado inicial, viabilizando o seu reuso futuro. O trecho de código a ser inserido por essa modificação encontra-se ilustrado na Figura 46.

```
  mark 2         //inicio do trecho adaptativo 2
  mark 1         //inicio do trecho adaptativo 1
  emark 1        //fim do trecho adaptativo 1
  emark 2        //fim do trecho adaptativo 2
  endp           //fim de trecho de código
```

Figura 46 – Dado base (área de dados)

Para detalhar o funcionamento desse código, o mesmo é explicitado para o caso do cálculo do fatorial de 3 (Figura 47 à Figura 51).

Inicialmente, `eax` recebe o valor `1`. Como `n` é maior que zero, a instrução `adapt 1,step` é executada, gerando a multiplicação (eax ← 3*1 = 3) e decrementando `n` (n=2), conforme ilustrado na Figura 47.

```
FAT proc         //procedimento fatorial(n)
  pop n          //desempilha n (fat (n))
```

```
  mov eax,1                  //eax ← 1
  cmp n,0                    //compara n com 0
  jle X                      //desvia para X se n <= 0
  adapt 1,step               //adapta
  mark 2
    mul eax,n
    dec n
    cmp n,0
    jz Y'
    adapt 1,step
    mark 1
    emark 1
   label Y'                  // Y' 1ª instância de Y
  emark 2
label X
  mov res,eax
  adapt 2,base
  ret
FAT endproc
```

Figura 47 – Calculando o fatorial de 3 (após 1ª Modificação).

Como `n` é diferente de 0, o código inserido no programa faz com que o mesmo se modifique novamente, inserindo os códigos responsáveis pela multiplicação (eax ← 3*2 = 6) e decrementando `n` (n=1) - Figura 48.

```
FAT proc                     //procedimento fatorial(n)
  pop n                      //desempilha n (fat (n))
  mov eax,1                  //eax ← 1
  cmp n,0                    //compara n com 0
  jle X                      //desvia para X se n <= 0
  adapt 1,step               //adaptou
  mark 2
    mul eax,n                //eax ← 1*3 = 3
    dec n                    //n ← 3-1=2
    cmp n,0
    jz Y'                    //n != 0, não desvia
    adapt 1,step             //adapta
       mul eax,n
       dec n
       cmp n,0
       jz Y''
       adapt 1,step
       mark 1
       emark 1
     label Y''               // Y'' 2ª instância de Y
  label Y'                   // Y' 1ª instância de Y
```

```
  emark 2
label X
  mov res,eax
  adapt 2,base
  ret
FAT endproc
```

Figura 48 – Calculando o fatorial de 3 (após 2ª modificação de código)

Novamente, `n` ainda é diferente de zero e, portanto, o programa se modificará uma terceira vez (Figura 49), o que completa o cálculo do valor do fatorial.

```
FAT proc                    //procedimento fatorial(n)
  pop n                     //desempilha n (fat (n))
  mov eax,1                 //eax ← 1
  cmp n,0                   //compara n com 0
  jle X                     //desvia para X se n <= 0
  adapt 1,step              //adaptou
  mark 2
    mul eax,n               //eax ← 1*3 = 3
    dec n                   //n ← 3-1=2
    cmp n,0
    jz Y'                   // n! = 0, não desvia
    adapt 1,step            //adaptou
       mul eax,n            //eax ← 3*2 = 6
       dec n                //n ← 2-1=1
       cmp n,0
       jz Y''               //n != 0, não desvia
       adapt 1,step         //adapta
         mul eax,n
         dec n
         cmp n,0
         jz Y'''
         adapt 1,step
         mark 1
         emark 1
       label Y'''           // Y''' 3ª instância de Y
     label Y''              // Y'' 2ª instância de Y
  label Y'                  // Y' 1ª instância de Y
  emark 2
label X
  mov res,eax
  adapt 2,base
  ret
FAT endproc
```

Figura 49 – Calculando o fatorial de 3 (após 3ª modificação de código)

Seguindo o cálculo do produtório, o seu valor é armazenando-o no acumulador (eax ← 6*1) e decrementando `n` (n=0). Como agora `n` é igual a zero, a instrução `jz Y'''` faz com o que a execução salte para o endereço associado a `Y'''`, interrompendo processo de modificação e encerrando o cálculo do fatorial (Figura 50).

```
FAT proc                          //procedimento fatorial(n)
  pop n                           //desempilha n (fat (n))
  mov eax,1                       //eax ← 1
  cmp n,0                         //compara n com 0
  jle X                           //desvia para X se n <= 0
  adapt 1,step                    //adaptou
  mark 2
    mul eax,n                     //eax ← 1*3 = 3
    dec n                         //n ← 3-1=2
    cmp n,0
    jz Y'                         //n != 0, não desvia
    adapt 1,step                  //adaptou
       mul eax,n                  //eax ← 3*2 = 6
       dec n                      //n ← 2-1=1
       cmp n,0
       jz Y''                     //n != 0, não desvia
       adapt 1,step               //adaptou
         mul eax,n                //eax ← 6*1 = 6
         dec n                    //n ← 1-1=0
         cmp n,0
         jz Y'''                  //n == 0, desvia para Y'''
         adapt 1,step             //não executa
         mark 1
         emark 1
       label Y'''
     label Y''                    // Y''' 3ª instância de Y
  label Y'                        // Y'' 2ª instância de Y
  emark 2                         // Y' 1ª instância de Y
label X:
  mov res,eax
  adapt 2,base                    //res ← eax = 6
  ret                             //adapta
FAT endproc
```

Figura 50 – Calculando o fatorial de 3 (antes da 4ª modificação de código)

Após armazenar o valor do fatorial em `res` (instrução `mov res,eax`), o procedimento realiza uma última auto-modificação (instrução `adapt 2,base`). Esta auto-modificação faz com que o procedimento `FAT` retorne à sua forma original (vide

Figura 51), removendo as instruções inseridas pelas instruções `adapt 1,base` (Figura 44 e Figura 50), de forma que em uma próxima chamada possa funcionar corretamente e com o intuito de evitar o consumo desnecessário de memória.

```
FAT proc                    //procedimento fatorial(n)
  pop n                     //desempilha n (fat (n))
  mov eax,1                 //eax ← 1
  cmp n,0                   //compara n com 0
  jle X                     //desvia para X se n <= 0
  adapt 1,step              //adaptou
      mark 2
      mark 1
      emark 1
      emark 2
label X
  mov res,eax               //res ← eax = 6
  adapt 2,base              //adaptou
  ret                       //retorno
FAT endproc
```

Figura 51 – Calculando o fatorial de 3 (após a 4ª modificação de código)

É importante destacar os seguintes pontos deste exemplo:

- Enquanto a instrução `adapt 1,step` remove o trecho marcado, indexado por 1 (1 não é posição de memória), também cria um novo trecho marcado, com o mesmo indexador.
- Durante a montagem do código que começa em `step`, o *`label Y`* é resolvido em função da declaração do mesmo existente no próprio trecho a ser inserido, agindo dessa forma como um gerador de *labels*.

6.1.2 Fatorial Adaptativo – Autômato de Execução Adaptativo

Esse tópico descreve o funcionamento do exemplo citado no item anterior, com a representação do código via autômato de execução adaptativo, explicitando assim como o processo de execução do exemplo é visto pelo ambiente de execução.

O código do exemplo, descrito na Figura 44, é representado via formato de autômato de execução adaptativo conforme a ilustração da Figura 52.

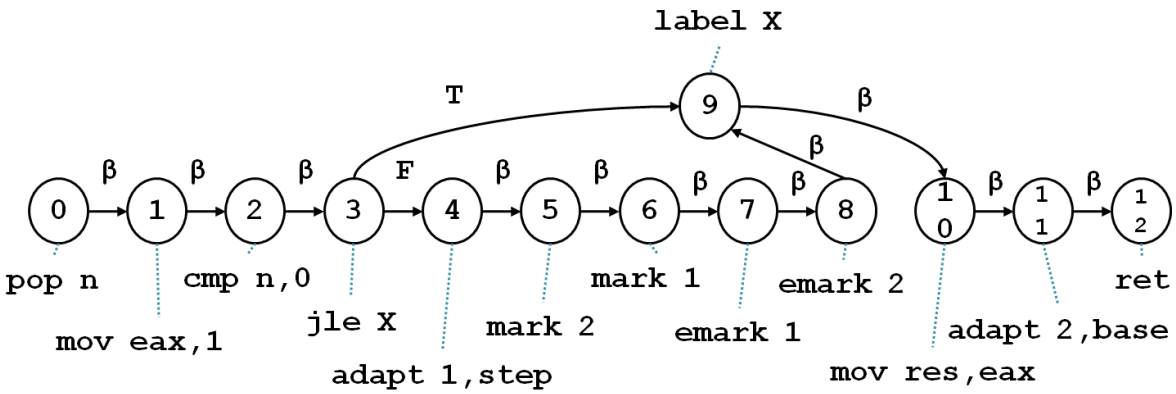


Figura 52 – Código a ser executado no formato de autômato de execução adaptativo

Com o intuito de facilitar a compreensão desse exemplo, as figuras Figura 53 e Figura 54 apresentam a forma de autômatos dos dados `step` (descrito na Figura 45) e `base` (descrito na Figura 46), respectivamente.

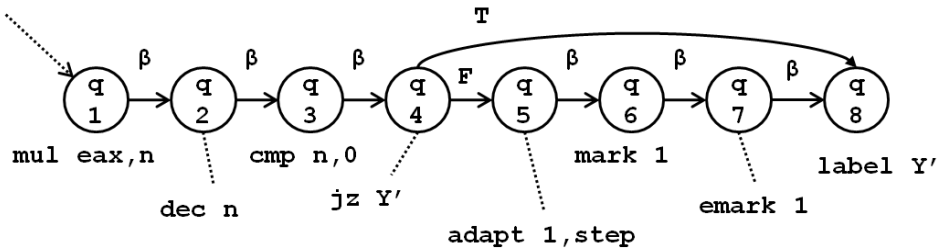


Figura 53 - Dados `step` – formato de autômato de execução adaptativo.

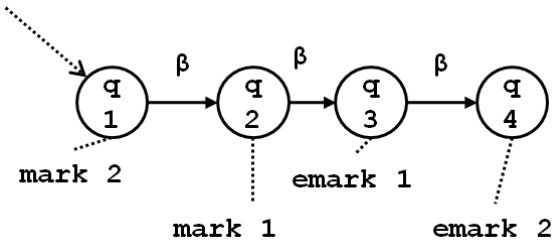


Figura 54 – Dados `base` – formato de autômato de execução adaptativo.

Ressalta-se que, em termos de funcionamento do sistema, a conversão de representação da imagem do código armazenado nesses dados para o formato de autômato de execução adaptativo ocorre apenas durante a execução da instrução

`adapt` (portanto, a numeração do estado é calculada apenas durante a ação adaptativa – para evitar confusões, a Figura 53 e a Figura 54 utilizam a designação de estados na forma q_x).

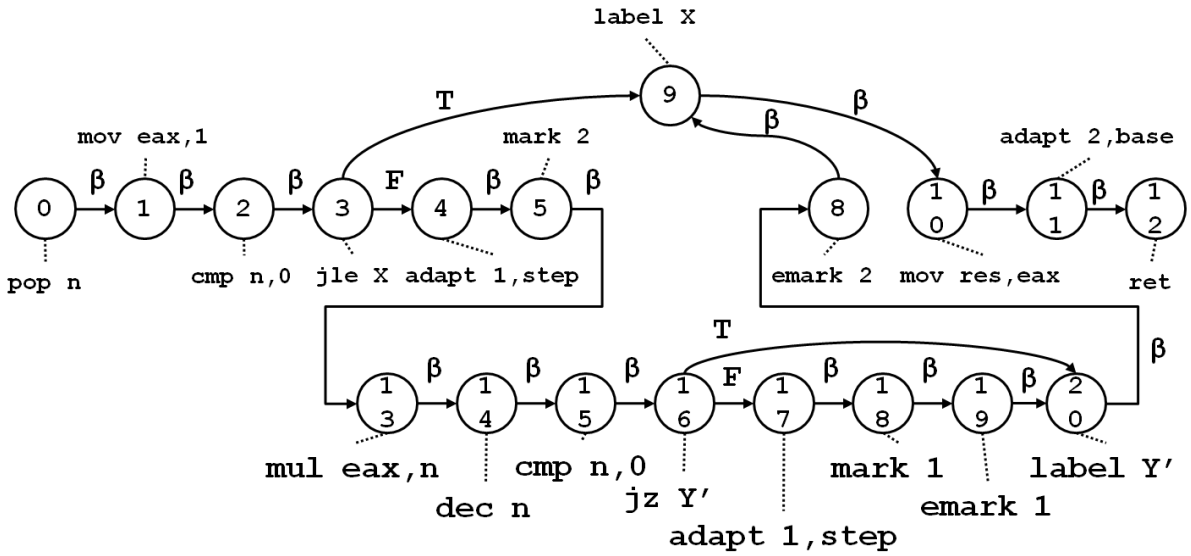


Figura 55 – Código após a 1ª modificação – formato de autômato de execução adaptativo.

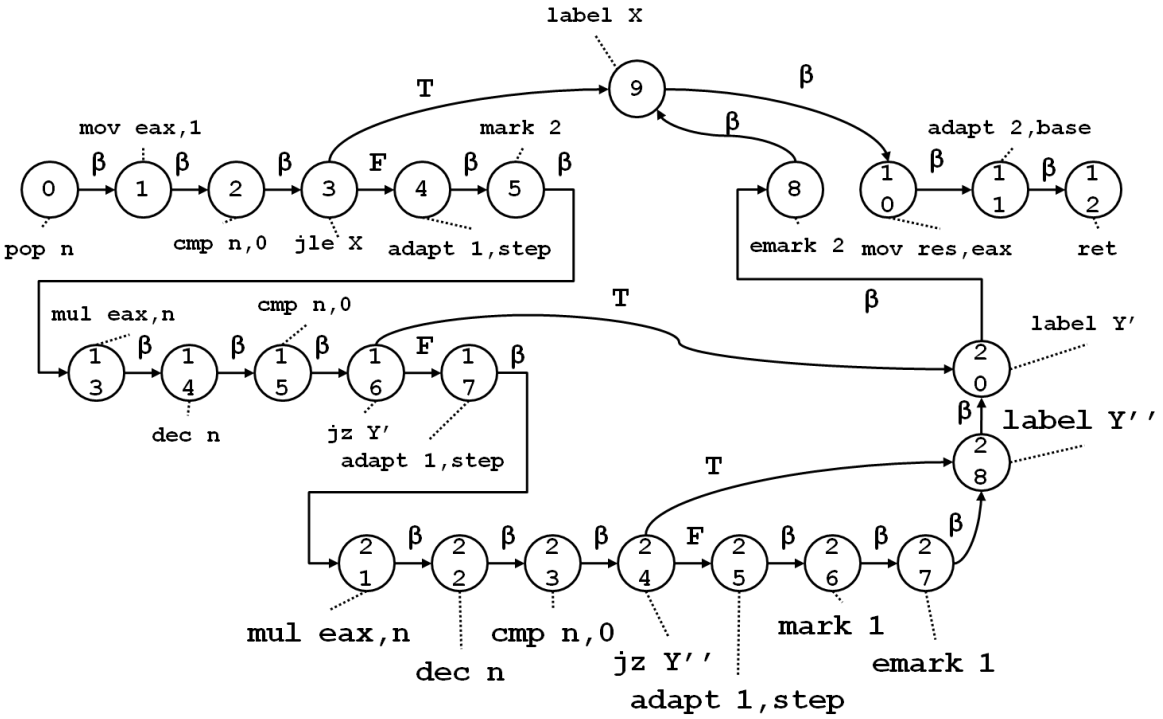


Figura 56 – Código após a 2ª modificação – formato de autômato de execução adaptativo.

Durante a primeira modificação (ativada no momento da execução da instrução associada ao estado 4), a instrução `adapt` chama a função adaptativa homônima, que por sua vez realiza a modificação do autômato de execução adaptativo. Essa modificação consiste na remoção das transições associadas aos estados 6 e 7 (bem como a eliminação desses estados) e da montagem e inserção do trecho de autômato associado ao dado `step`, devidamente parametrizado (processo ilustrado na Figura 55).

Alterações similares ocorrem ao atingir o estado 17 (modificação 2, ilustrado na Figura 56) e no estado 25 (modificação 3, ilustrado na Figura 57).

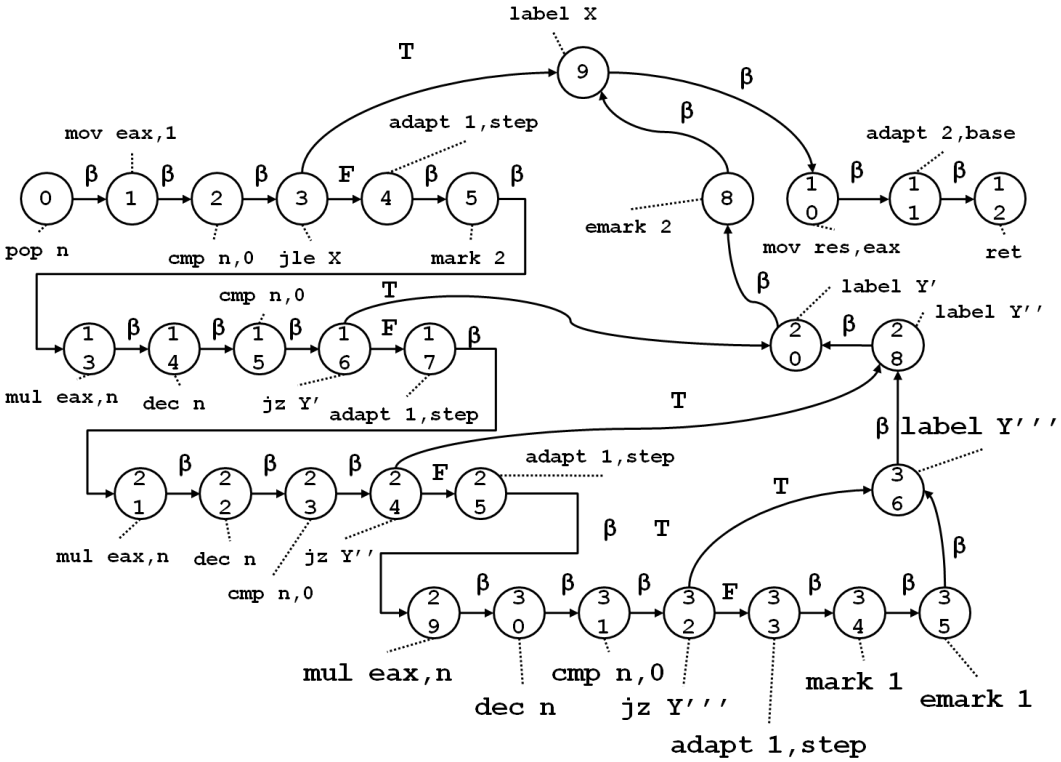


Figura 57 – Código após a 3ª modificação – formato de autômato de execução adaptativo.

Ao atingir o estado 32 (vide Figura 57), como n é igual a 0, significa que o produtório que calcula o fatorial foi devidamente calculado. Ao executar a instrução associada a esse estado (`jz Y'''`), o interpretador escreve T na cadeia de trabalho e executa a transição (32, T) → 36 (estado associado ao `label Y'''`).

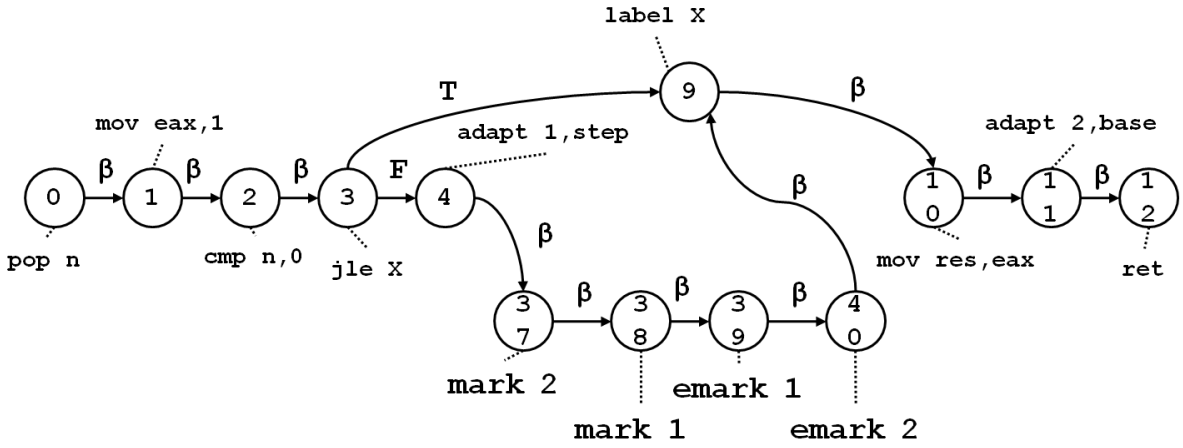


Figura 58 – Código após a 4ª modificação – formato de autômato de execução adaptativo.

Continuando a execução, ao atingir o estado 11, o autômato sofre outra auto-modificação (modificação 4), a qual é responsável por retornar o código ao estado inicial, como indicado pela Figura 58.

6.1.3 Considerações a respeito do exemplo.

A metodologia utilizada para a escrita do código exemplo é baseada na premissa que a imagem do trecho a ser inserido é sempre igual. Portanto, os dados `step` e `base` (posições de memória que armazenam tais imagens) não se modificam durante a execução do programa adaptativo.

Ressalta-se, no entanto, que existem outras formas de resolver este problema por meio de auto-modificação de códigos. Por exemplo, é possível gerar, em uma área de dados (como a iniciada no endereço `calc`), a imagem da seqüência de instruções de multiplicações necessárias para resolver o fatorial e, em seguida, inserir e executar tal código.

Nesse caso, para calcular o fatorial de 3, este procedimento atribuiria o valor 1 para eax, em seguida, escreveria na área de dado apontada por `calc`, inicialmente vazia, o trecho de código descrito na Figura 59.

Assim como no exemplo anterior, após o término do cálculo, o procedimento deve armazenar o resultado e realizar outra modificação para retornar o código do

procedimento a sua forma original, bem como limpar a área de dados apontada por `calc`.

```
Área de dados
Dado calc
// endereço calc aponta para o local
// onde está armazenada a primeira
// instrução (mul eax,3) do dado abaixo.
mul eax,3  //eax ← 3*1
mul eax,2  //eax ← 3*2
mul eax,1  //eax ← 6*1
endp       //endp fim de montagem
```

Figura 59 – Instruções na área de dados (`calc`) que calcula o fatorial de 3.

6.2 TRANSDUTOR FINITO ADAPTATIVO – CONSIDERAÇÕES DE CODIFICAÇÃO

Este tópico procura ilustrar uma situação onde a programação adaptativa torna-se um ferramental interessante: a implementação de dispositivos adaptativos.

Em (PELEGRINI; NETO, 2007), é definido o conceito de transdutor finito adaptativo, o qual consiste em uma espécie de autômato finito adaptativo no qual o símbolo de entrada (utilizado na transição) é transformado em um símbolo de saída.

Por questões de simplificação, o transdutor finito adaptativo a ser implementado trabalha com chamadas às funções adaptativas do tipo A (depois da execução da transição) e que a função saída – responsável por converter o símbolo de entrada no de saída - é comum ao estado (i.e. todas as transições que se originam de um mesmo estado utilizam a mesma função saída). As ações adaptativas podem modificar o estado de destino de uma transição, a função saída do estado ou a chamada a função adaptativa (PELEGRINI; NETO, 2007).

A implementação consiste em, após a inicialização do estado corrente (como o valor do estado inicial), ocorre a execução de um *loop* infinito que consiste nos seguintes passos:

- Passo 1: Leitura do símbolo da cadeia de entrada

- Passo 2: Testar o valor de estado corrente, comparando com os valores de 0 a n' (todos os estados declarados), com intuito de determinar o ponto do código que o descreve.
- Passo 3: Aplicar a função saída associada ao estado sobre o símbolo lido.
- Passo 4: Comparar o valor de símbolo lido, um a um, com os valores especificados no alfabeto, de forma a determinar o ponto do código que descreve a transição a ser executada.
- Passo 5: Atribuir o valor do estado destino da transição ao estado corrente;
- Passo 6: Executar a função adaptativa associada a transição;
- Passo 7: Retornar ao passo 1.

```
estado_corrente = 0
enquanto (!!) //loop infinito
símbolo_lido = obtem_simbolo();
se estado_corrente = 0 then
         símbolo_saída = f0(símbolo_lido)
         se símbolo_lido = a then
          estado_corrente = 2;
          funçãoadaptativa(x);
          ...
     se símbolo_lido = z then
          estado_corrente = 4;
          funçãoadaptativa(x);
se estado_corrente = 1 then
         símbolo_saída = f1(símbolo_lido)
         se símbolo_lido = a then
          estado_corrente = 3;
          funçãoadaptativa(x);
          ...
     se símbolo_lido = z then
          estado_corrente = 6;
          funçãoadaptativa(x);
...
se estado_corrente = n' then
         símbolo_saída = fn(símbolo_lido)
         se símbolo_lido = a then
          estado_corrente = 7;
          funçãoadaptativa(x);
          ...
     se símbolo_lido = z then
          estado_corrente = n';
          funçãoadaptativa(x);
```

Figura 60 – Lógica não adaptativa – Ilustração da implementação.

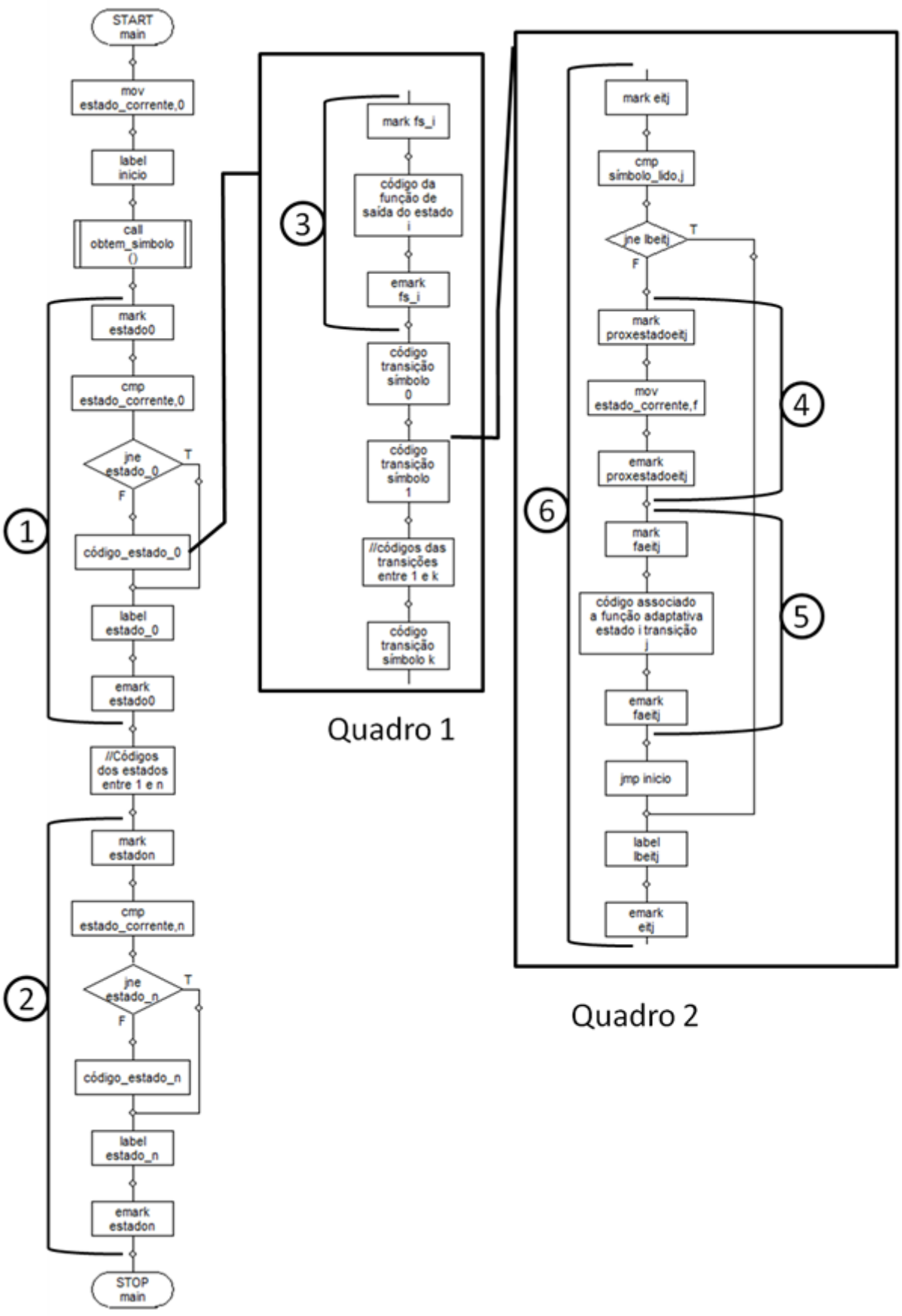


Figura 61 – Fluxograma do modelo de construção do transdutor finito adaptativo

A Figura 60 ilustra a lógica associada à implementação do transdutor adaptativo sem a utilização dos trechos de blocos adaptativos. O esquema geral de implementação desta lógica utilizando os trechos de blocos adaptativos é definido pelo fluxograma apresentado na Figura 61.

Para permitir a modificação do transdutor, têm-se que:

- A implementação de cada estado via linguagem AdaptCode é feita dentro de um trecho de código adaptativo, o que possibilita a criação e remoção de estados por meio da modificação desses trechos, como os ilustrados pelas chaves 1 e 2 da Figura 61 (nessa mesma figura, têm se que a codificação do estado 0 é feita dentro do trecho de código adaptativo `estado0`). Destaca-se, no entanto, que tal ação adaptativa não é suportada pelo dispositivo descrito.
- O código responsável por implementar a função de saída para cada estado é escrito dentro de um trecho de código adaptativo, o que permite modificar a função de saída livremente. No quadro 1 da Figura 61 é ilustrado o código associado a função de saída do estado 0, cujo trecho adaptativo é indexado por `fs_i` (chave 3 da Figura 61).
- Na codificação de cada uma das transições que se originam de um determinado estado (vide quadro 2 da Figura 61), são definidos três trechos de códigos adaptativos distintos:
    - O primeiro contém a instrução que faz a atribuição do novo estado corrente, o que permite modificar o estado destino dessa transição (exemplo: chave 4 da Figura 61, trecho de código adaptativo `proxestadoeitj`).
    - O segundo esta associado ao código que implementa a chamada da função adaptativa a ser feita após a execução da transição (exemplo: chave 5 da Figura 61, trecho de código adaptativo `faeitj`).
    - Por fim, o terceiro trecho de código adaptativo (exemplo: chave 6 da Figura 61, trecho de código adaptativo `eitj`) refere-se a implementação da transição como um todo, o que permitiria modificá-la totalmente (incluindo os trechos de códigos adaptativos listados nos dois itens acima). Assim como no caso do estado, a camada adaptativa desse

dispositivo não define tal operação, mas esta é explicitada para descrever a potencialidade do método.

A implementação de um estado do transdutor finito escrito na linguagem AdaptCode encontra-se parcialmente ilustrado no exemplo de código descrito na Figura 62.

```
mark estadoi                    //bloco adaptativo associado ao
                                //estado 0
cmp estado_corrente,i           //comparação
jne estado_i                    //salto para estado_0 se não for o
                                //estado 0
                                //código associado ao estado 0
mark fs_i                       // bloco adaptativo associado à
                                //lógica da função de saída
//logica função de              //codificação da função de saída
//saída

emark fs_i                      //término do bloco associado à
                                //função de saída
mark eitj                       //bloco adaptativo associado à
                                // transição com o símbolo j.
cmp símbolo_lido,j              //determina se o símbolo lido é j
jne lbeitj                      //caso não o seje salta para
                                //lbeitj (e testar outros valores)

mark proxestadoeitj             //bloco adaptativo associado à
                                determinação do próximo_estado
mov estado_corrente,f           //estado_corrente ← f (estado
                                //destino da transição)
emark proxestadoeitj            //término do bloco adaptativo


mark faeitj                     //bloco adaptativo associado à
                                //chamada da função adaptativa.
//código da função              //Código da função adaptativo
//adaptativa

emark faeitj                    //término do bloco função
                                //adaptativa
jmp inicio                      //salto para inicio - transitou
label lbeitj                    //label lbeitj

emark eitj                      //fim do bloco adaptativo associado
                                //a transição
//códigos das demais
//transições
label estado_0                  //label para salto
```

emark estado0	*//final do bloco adaptativo //associado ao estado*

Figura 62 – Código associado a um estado i do transdutor finito adaptativo.

6.3 CONSIDERAÇÕES SOBRE A IMPLEMENTAÇÃO

Esse tópico procura destacar as principais características do protótipo do ambiente de execução adaptativa. São elas:

- Interpretação de instrução baseada em comandos de linguagens de alto nível. Como o ambiente é construído visando a prova de conceito, a interpretação das instruções da linguagem AdaptCode são feitas por meio dos comandos de alto nível da linguagem no qual o ambiente foi construído. No entanto, as semânticas das instruções são baseadas nas existentes na arquitetura Intel® 32 bits.
- Suporte a tipos de dados inteiros (32 bits) e strings, armazenadas em estruturas de dados baseadas em variáveis similares suportadas pela linguagem de programação no qual o ambiente foi construído.
- Possibilita apenas a execução de um único programa por vez (mono-programado).
- Suporte a instruções dedicadas ao tratamento de variáveis do tipo string.

6.3.1 Controle de estruturas

Ao se implementar o autômato de execução adaptativo, um dos aspectos de maior complexidade refere-se ao controle das estruturas auxiliares necessárias para o correto funcionamento da auto-modificação do código.

Tais estruturas originam-se da necessidade de adicionar ou remover declarações de *labels* e trechos de código adaptativo. Por exemplo, ao modificar um determinado trecho adaptativo T1, deve-se tanto remover as informações de outros trechos

adaptativos existentes dentro do trecho T1, quanto às informações dos *labels* declaradas dentro do trecho T1.

Para controlar tais informações foram definidas duas metodologias. A primeira consiste na declaração explícita dos trechos adaptativos quanto dos *labels* no próprio código. Nesse caso, ao se apagar o trecho T1, varre-se, uma a uma, as instruções que pertencem a esse trecho e, ao encontrar informações de *labels* (instrução `label`) ou de blocos adaptativos (instruções `mark` e `emark`), um tratamento adequado é aplicado (no caso remover as declarações). Essa abordagem foi adotada na implementação.

A segunda abordagem consiste em criar estruturas de dados que auxiliam no armazenamento das informações necessárias, tais que as mesmas permitam mapear as relações de aninhamento entre os diversos trechos de códigos adaptativos e *labels* declarados.

6.4 DISCUSSÃO SOBRE DESEMPENHO

Concluindo esse capítulo, este tópico procura discutir alguns aspectos de desempenho dessa linguagem. Estes estudos foram feitos com um protótipo associado a uma versão anterior da linguagem AdaptCode, denominada de AdaptCod, apresentado no artigo (PELEGRINI; NETO, 2008). Esse ambiente de execução protótipo foi implementado como máquina virtual denominada de *AdaptCod Machine*, responsável por processar o código (montar programa na forma de autômato, de maneira semelhante ao descrito nas seções anteriores) e o executar.

Para analisar o desempenho da *AdaptCod Machine* em relação à máquina física, foram utilizados três programas. Devido ao fato de a *AdaptCod Machine* ser um protótipo experimental, com quantidade limitada de instruções, optou-se por desenvolver ensaios simples em lugar de usar programas pré-existentes.

Para garantir que os programas fossem escritos de maneira similar, foi montado um pequeno compilador que gera o programa tanto para a linguagem de máquina do PC quanto para a do interpretador (portanto, tais programas são estáticos e não

realizam auto-modificações). Os programas foram compilados tanto para a linguagem *Assembly* do PC (sendo montados via MASM32 (MASM32, 2006)) quanto para o *AdaptCod*.

O primeiro programa (Teste 1) calcula os fatoriais de maneira recursiva, para os números de 1 a 10, apresentando os resultados na tela; o segundo (Teste 2) realiza um milhão de loops, cada qual consiste de cinco operações: duas de soma (somando uma variável a ela mesma), um decremento (da variável que controla o salto) e uma operação de apresentação de dados na tela; e o terceiro programa (Teste 3) é similar ao anterior (Teste 2), exceto pelo fato de não fazer a apresentação de dados na tela.

Os programas foram executados em um computador com a seguinte configuração: AMD 64, 3.2 Ghz, com 1 GB de RAM. Foram coletados: o tempo de execução total – o tempo de carregar e executar o programa, até o seu término, denotado por T_{total} – e o tempo de ocupação da CPU pelo programa, denotado por T_{proc}.

Para garantir a consistência dos resultados, foram realizadas dez medições para cada programa. Os valores médios dos tempos coletados estão apresentados na Tabela 1.

Tabela 1 – Desempenho de Execução

	Assembly		*AdaptCod*	
	T_{total} (ms)	T_{proc} (ms)	T_{total} (ms)	T_{proc} (ms)
Teste 1	40,60	15,63	42,20	42,20
Teste 2	69235,50	3651,56	71517,10	31525,00
Teste 3	62,50	50,00	12907,70	12773,44

Para os programas que realizam operações de saída (exibem dados na tela), Teste 1 e Teste 2, o tempo de execução total do *AdaptCod* tende a se aproximar do obtido na versão *assembly*. Isto se deve ao fato de que as operações de entrada/saída costumam ser lentas, quando comparadas aquelas exclusivamente dependentes do processador (HENNESSY; PATTERSON, 2003), bem como ao fato de estas operações serem tratadas de maneira relativamente similar (a máquina virtual que interpreta o *AdaptCod* não trabalha com nenhum esquema sofisticado relacionado à entrada/saída, tal como virtualização de periféricos). Entretanto, ao se analisar o tempo que estes programas ocupam no processador, é possível perceber um

acréscimo de mais de 60% no tempo de utilização, causado pelo processo de interpretação de instruções e das transições entre os estados do autômato.

Esta situação se confirma nos resultados referentes ao Teste 3 (programa que faz uso intensivo de processador). Observa-se que o tempo total referente à execução deste programa na versão *AdaptCod* é significativamente maior que na versão *assembly* do PC. A elevada utilização do processador é esperada, devido ao *overhead* ocasionado pela máquina virtual, destacando-se o fato de que o interpretador de instruções foi construído em linguagens de alto nível. Outro fator importante que degrada o desempenho é a ausência de mecanismo de cache na implementação da máquina virtual.

No intuito de averiguar se tal queda de desempenho é efeito colateral do uso de um interpretador de instruções da máquina virtual ou do tempo associado à determinação da próxima instrução (transição do autômato referente ao modelo de execução), o código da máquina virtual foi alterado, para não interpretar as instruções, apenas percorrer o mesmo caminho (i.e. passar pelo mesmo número de estados do autômato). O resultado foi uma queda no tempo de execução total, que caiu de aproximadamente 13 mil milisegundos para 856 milisegundos, o que indica a necessidade de uma otimização da implementação do interpretador de instruções.

Um último teste de desempenho foi realizado, com intuito de averiguar o desempenho da operação de auto-modicação / adaptação. Em um novo programa, com 135 instruções, foi feito uma substituição de um trecho de código adaptativo composto de 32 instruções por outro trecho de código composto de outras 32 instruções. O processo de alteração demora, aproximadamente, 87 milisegundos, sobre um tempo de execução de 193,8 milisegundos. Ou seja, 45% do tempo de execução foi gasto para alterar 23,7% do código. Este elevado percentual deve-se ao processo de montagem do código (conversão da linguagem simbólica para a representação interna) do código a ser inserido, antes de aplicar a modificação.

Por fim, ressalta-se que: (i) as implementações desenvolvidas (tanto a da linguagem AdaptCod quanto a da AdaptCode) não visam desempenho, apenas provar os conceitos apresentados; e (ii) o desenvolvimento de uma versão eficiente e completa exige programadores experientes.

7 CONSIDERAÇÕES FINAIS

Neste capítulo descrevem-se as contribuições advindas desta pesquisa, destacando-se os resultados obtidos, bem como sugestões de futuros trabalhos acerca do tópico discutido ao longo dessa dissertação.

7.1 CONTRIBUIÇÕES

Ao longo dessa dissertação, procurou-se expandir o assunto iniciado em (FREITAS, 2008): as linguagens para programação adaptativa. Como resultados desse trabalho têm-se:

- A consolidação de alguns conceitos inicialmente propostos em (FREITAS, 2008), bem como a introdução de novos conceitos associado ao tópico linguagens para programação adaptativa, como trecho de código adaptativo, que eventualmente podem ser explorados em outras propostas.
- A definição de um novo dispositivo adaptativo de processamento, o autômato de execução adaptativo, que reúne características dos autômatos adaptativos e dos modelos de execuções utilizados por computadores;
- A proposição da linguagem para programação adaptativa AdaptCode, baseada em linguagens *assembly*. Adicionalmente a essa contribuição, têm-se a definição de um processo de auto-modificação de código que é baseado na marcação de trechos a serem modificados e na instrução de auto-modificação do código. Apesar de estar definido para linguagens simbólicas, podem ser aproveitadas em outras classes de linguagens de programação, como as linguagens estruturadas.
- Ao ramo da engenharia de software e programação, esse trabalho procura contribuir por meio da proposição de linguagens para programação adaptativa que contornam algumas dificuldades enfrentadas ao se escrever um código que se auto-modifica na linguagem *assembly* tradicional, e nas quais o processo de

auto-modificação, em si, é delegado ao ambiente de execução, visando facilitar e simplificar a utilização desse recurso.

7.2 TRABALHOS FUTUROS

O trabalho de Freitas (FREITAS, 2008), seguido da presente dissertação, são os primeiros trabalhos de dois novos tópicos associado à tecnologia adaptativa: as linguagens para programação adaptativa e os códigos adaptativos. Ainda existem tópicos nessa pesquisa que necessitam de contribuições e avanços, como:

- A proposição de *hardwares* para a execução de código adaptativo. Pode-se citar, como proposta, a criação de hardwares dedicados para a execução do autômato de execução adaptativo (similar ao feito em (KEUNG; TYAGI, 2006) para o *self-modifying finite automata*). Tal pesquisa aumentaria a eficiência dos códigos adaptativos e facilitaria o desenvolvimento de linguagens para programação adaptativa.
- Estudos comparativos de complexidade entre escrever códigos utilizando os recursos adaptativos e os não adaptativos. Por exemplo, comparar a complexidade de tempo e espaço entre o fatorial resolvido por adaptatividade e o resolvido de forma recursiva e o resolvido de forma interativa.
- Em relação à proposta da linguagem AdaptCode, para o desenvolvimento de códigos comerciais ou como base para o desenvolvimento linguagens de alto nível para programação adaptativa, ainda existem trabalhos a serem feitos. Pode-se destacar como trabalhos a serem realizados, a implementação de um ambiente de execução eficiente e completo, bem como o estudo aprofundado do desempenho do mesmo (por exemplo, em relação aos mecanismos de cache, perfil do programa).
- A proposição de linguagens de alto nível para programação adaptativa que utilizem o mecanismo descrito nesse trabalho como substrato. Essas linguagens de alto nível poderiam mapear os comandos associados a modificação de códigos nos descritos para a linguagem AdaptCode.

- O estudo do processo de auto-modificação proposto nessa dissertação sobre a ótica da engenharia de *software*.

7.3 CONCLUSÃO

Este dissertação apresenta a proposta de código de montagem adaptativo, descrito pela linguagem AdaptCode, bem como seu mecanismo de execução. Tal proposta procura contornar algumas dificuldades enfrentadas ao se escrever um código que se auto-modifica na linguagem de montagem tradicional por meio da utilização dos conceitos da Tecnologia Adaptativa. Para facilitar ao programador utilizar os mecanismos de alteração de código, disponibiliza-se para ele três operações, que implementam as mais complexas dificuldades da tarefa de alteração do código.

Foi proposta uma nova forma de execução de código, baseada em dispositivos adaptativos, denominado de autômato de execução adaptativo. Esse dispositivo adaptativo, associado a um ambiente de execução de linguagens, procura esconder as dificuldades do processo de alteração em si do programador, permitindo que este altere o código baseado em um mecanismo simples de delimitação de trechos de seu programa e no uso da instrução de modificação do código.

Espera-se que a proposta dessa linguagem sirva como base para o desenvolvimento de linguagens de alto nível para programação adaptativa, que por sua servirão para implementações de programas adaptativos.

Não obstante, também é esperado que esta linguagem seja usada para a escrita de códigos auto-modificáveis que possuam outras finalidades, como as previamente citadas (por exemplo, programas que desejem ocultar detalhes internos).

REFERÊNCIAS BIBLIOGRÀFICAS

ACAR, U. A.; BLELLOCH, G. E; HARPER, R. **Adaptive functional programming.** ACM Transactions on Programming Languages and Systems (TOPLAS), vol. 28, nº 6, 2006. pp. 990-1034.

AHO, A. V. et al. **Compilers: Principles, Techniques, and Tools**. 2ª ed. Addison Wesley, 2006. 1009 p.

ANCKAERT, B.; MADOU, M.; BOSSCHERE, K. De. **A model for self-modifying code**, Information Hiding, vol. 4437/2007, 2006, pp. 232-248. Springer-Verlag Berlin Heidelberg 2007.

AYCOCK, J. **A Program Execution Model Based on Generative Dynamic Grammars**. IASTED International Conference on Computer Science and Technology (CST 2003). 2003. Cancun, Mexico. pp. 411-416. ACTA Press.

BABCSÁNYI, I. **Equivalence of Mealy and Moore automata**. Acta Cybernetica. Vol **14**(4). 2000. p. 541-552.

BARHAM, P et al. **Xen and the Art of Virtualization**. 19th ACM Symposium on Operating Systems Principles (SOSP 2003). Bolton Landing, NY, USA: ACM Press. 2003. p. 164 -177.

BLASS, A.; GUREVICH, Y. **Algorithms: A Quest for Absolute Definitions**. Bulletin of the European Association for Theoretical Computer Science, 2003(81): p. 195 - 225. 2003.

BRINGSJORD, S.; FERRUCI, D. **Brutus and the Narrational Case Against Church's Thesis**. Narrative Intelligence Symposium AAAI 1999 Fall Symposium: p. 105 - 111. North Falmouth, Massachusetts. 1999.

CHERANSOFT, Visual Turing V1.0. Disponível em: < http://www.inf.pucrs.br/~calazans/graduate/teo_comp/vturing.exe>. Acessado em: 04 mar. 2006.

DEITEL, H.M., DEITEL, P.J.. **Java Como Programar**. 4ª Edição. Bookman, 2003. 1386p.

DETMER, R. C. **Introduction to 80X86 Assembly Language and Computer Architecture.** Jones & Bartlett Publishers, 2001. 499 p.

ENGLER, D. R.; HSIEH, W. C.; KAASHOEK, M.F. **C: a language for high-level, efficient, and machine-independent dynamic code generation.** *Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 1996, ACM: St. Petersburg Beach, Florida, United States. pp. 131 – 144.

FREITAS, A. V. de; NETO, J. J. **Conception of adaptive programming languages.** Proceedings of the 17th IASTED international conference on Modelling and simulation, 2006. pp. 453-458. ACTA Press.

FREITAS, A. V. de; NETO, J. J. **Adaptive Device With Underlying Mechanism Defined By a Programming Language**. Proceedings of the 4th WSEAS International Conference on Information Security, 2005, Tenerife, Spain. pp. 423-428.

FREITAS, A. V. de; NETO, J. J. **WTA 2007 - II.1 - Programming Languages adherent to the Adaptive Paradigm**. Revista IEEE América Latina. Vol. 5, Num. 7, ISSN: 1548-0992, Novembro 2007. pp. 522-526.

FREITAS, A. V. de, **Considerações sobre o Desenvolvimento de Linguagens Adaptativas de Programação.** 2008. 122 p. Tese (Doutorado) Departamento de Computação e Sistemas Digitais (PCS), Escola Politécnica, Universidade de São Paulo, São Paulo, 2008.

GIFFIN, J. T.; CHRISTODORESCU M.; KRUGER L. **Strengthening Software Self-Checksumming via Self-Modifying Code**. Proceedings of the 21st Annual Computer Security Applications Conference (ACSAC'05), IEEE Computer Society Washington, DC, USA, 2005. pp. 23-32.

HENNESSY, J. L.; PATTERSON, D. A. **Arquitetura de Computadores: Uma Abordagem Quantitativa**. 3ª ed. Editora Campus, 2003. 827 p.

HONGXU, C.; ZHONG, S.; ALEXANDER, V. **Certified self-modifying code**. SIGPLAN Notices. Vol. 42(6), 2007. pp. 66-77.

INTEL. **Intel® 64 and IA-32 Architectures Software Developer's Manual in Volume 2A: Instruction Set Reference, A-M.** 2006. 744 p. Disponível em:

<http://www.intel.com/design/processor/manuals/253666.pdf>. Acessado em: 02 mar. 2007.

INTEL. **Intel® 64 and IA-32 Architectures Software Developer's Manual in Volume 2B: Instruction Set Reference, N-Z**. 2006. 612 p. Disponível em: <http://www.intel.com/design/processor/manuals/253667.pdf>. Acessado em: 02 mar. 2007.

IWAI, M. K. **Um formalismo gramatical adaptativo para linguagens dependentes de contexto.** 2000. 191 p. Tese (Doutorado) - Departamento de Computação e Sistemas Digitais (PCS), Escola Politécnica, Universidade de São Paulo, São Paulo, 2000.

KANZAKI, Y. et al. **Exploiting self-modification mechanism for program protection**. Proceedings of the 27th Annual international Conference on Computer Software and Applications (November 03 - 06, 2003). COMPSAC. IEEE Computer Society, pp. 170-179.

KEUNG, K. M.; TYAGI, A. **State space reconfigurability: an implementation architecture for self modifying finite automata.** Proceedings of the 2006 international conference on Compilers, architecture and synthesis for embedded systems. 2006, ACM: Seoul, Korea. p. 83-92.

LANGSAM, Y.; AUGENSTEIN, M.J.; TENENBAUM, A.M. **Data Structure using C and C++**. 2º ed. Prentice-Hall, Inc. 1996. 672 p.

LEWIS, H. R.; PAPADIMITRIOU, C.H. **Elements of the Theory of Computation**. 1st ed. Prentice-Hall Inc., 1981. 466 p.

LIEBERHERR, K. J. **Adaptive Object-Oriented Software: The Demeter Method**. PWS Publishing Company Boston, 1996.

LIEBERHERR, K.; ORLEANS, D.; OVLINGER, J. **Aspect-oriented programming with adaptive methods.** Communications of the ACM, vol. 44(10), 2001. pp. 39-41.

LIU, Y.; PRADHAN, S. **The Demeter method: an efficient way to build adaptive software.** ACM SIGICE Bulletin, vol 22, 1996. pp. 7-19.

LTA. Laboratório de Linguagens e Técnicas Adaptativas - LTA Home Page. Disponível em <http://www.pcs.usp.br/~lta/>. Acessado em 10 fev. 2006.

MADOU, M., et al., **Software protection through dynamic code mutation**. Information Security Applications, vol. 3786/2006, LNCS, 2005. pp. 194-206. Springer-Verlag Berlin Heidelberg 2006.

MASM32. Montador MASM 32. Disponível em: <http://www.masm32.com/>. Acessado em: 18 jun. 2006.

MASSALIN, H. **Synthesis: An Efficient Implementation of Fundamental Operating System Services**. 1992, 141 p. PhD Thesis dissertion - Columbia University, New York, NY, 1992. Disponível em: <http://portal.acm.org/citation.cfm?id=143219>

MCKEEMAN, W.M. **Peephole optimization**. Communications of the ACM. Vol **8**(7). 1965. p. 443-444. Available in <http://doi.acm.org/10.1145/364995.365000>

MICROSOFT. Overview of the .NET Framework. Disponível em: <http://msdn.microsoft.com/en-us/library/zw4w595w(VS.71).aspx/>. Acessado em: 02 jan. 2009.

MIZRAHI, V.V. **Treinamento em Linguagem C Modulo 1**. McGraw-Hill, 1990. 241 p.

NAIR, R.; SMITH, J. E. **Virtual Machines: Versatile Platforms for Systems and Processes**. 1st ed. Morgan Kaufmann, 2005. 656 p.

NETO, J.J. **Introdução à Compilação**. Engenharia de Computação. LTC - Livros Técnicos e Científicos Editoras, 1987, 222 p.

NETO, J.J. **Contribuição à metodologia de construção de compiladores**. 1993. 272 p. Tese (Livre Docência) - Departamento de Computação e Sistemas Digitais (PCS), Escola Politécnica, Universidade de São Paulo, São Paulo, 1993.

NETO, J.J., **Adaptive Automata for Context-Sensitive Languages**. SIGPLAN NOTICES, vol. 29, nº 9, 1994. pp. 115-124.

NETO, J. J. **Adaptive Rule-Driven Devices - General Formulation and Case Study.** Implementation and Application of Automata *6th International Conference, CIAA*. 2001. Pretoria, South Africa: Lecture Notes in Computer Science, Springer-Verlag.

NETO, J. J. **WTA 2007 - I.1 - A small survey of the evolution of Adaptivity and Adaptive Technology.** Revista IEEE América Latina. Vol. 5, Num. 7, ISSN: 1548-0992, Novembro 2007. pp. 496-505.

NETO, J. J.; MAGALHÃES M. E. ***Reconhecedores Sintáticos - Uma Alternativa Didática para Uso em Cursos de Engenharia.*** XIV Congresso Nacional de Informática. 1981. São Paulo. pp. 140-148

NETO, J. J.; PARIENTE, C.A.B. **Adaptive Automata - a Revisited Proposal.** Implementation and Application of Automata 7th International Conference, CIAA 2002, LNCS, vol 2608/2003. pp. 158-168. Springer-Verlag Berlin Heidelberg 2003.

PARIENTE, C.A.B. **Gramáticas Livres de Contexto Adaptativas com Verificação de Aparência**. 2004. 261 p. Tese (Doutorado) - Departamento de Computação e Sistemas Digitais (PCS), Escola Politécnica, Universidade de São Paulo, São Paulo, 2004.

PEDRAZZI, T.; TCHEMRA, A. H.; ROCHA R. L. A. **Adaptive Decision Tables - a Case Study of their Application to Decision-Taking Problems.** Proceedings of International Conference on Adaptive and Natural Computing Algorithms - ICANNGA 2005. 2005. Coimbra, Portugal.

PELEGRINI, E. J.; NETO, J. J. **Um modelo de execução para código dinamicamente variável, baseado em autômatos adaptativos**. Revista IEEE América Latina (a ser publicado). Vol. **6**(5), 2008.

PELEGRINI, E. J; NETO, J. J. **Applying Adaptive Technology in Data Security. in Peruvian Computer Week.** Anais das VI Jornadas Peruanas de Computación (JPC 2007), Trujillo, Peru, 2007. pp. 31-40. 2007.

PISTORI, H. **Tecnologia Adaptativa em Engenharia de Computação: Estado da Arte e Aplicações.** 2003. 171 p. Tese (Doutorado) - Departamento de Computação e Sistemas Digitais (PCS), Escola Politécnica, Universidade de São Paulo, São Paulo, 2003.

PRESSMAN, R.S. **Engenharia de Software**. 5ª ed. McGraw-Hill, 2002. 843 p.

ROCHA, R. L. A.; NETO, J. J. **Autômato adaptativo, limites e complexidade em comparação com máquina de Turing.** *Second Congress of Logic Applied to Technology - LAPTEC'2000*. São Paulo: Faculdade SENAC de Ciências Exatas e Tecnologia, 2000. pp. 33-48.

ROSE, R. **Survey of System Virtualization Techniques**. Available from: <http://www.robertwrose.com/vita/rose-virtualization.pdf>. Acessado em 11/11/2006.

RUBINSTEIN, R.; SHUTT, J.N. **Self-Modifying Finite Automata: Basic Definitions and Results**. Techincal Report WPI-CS-TR-95-2. 1995, Worcester Polytechnic Institute: Worcester, Massachusetts, 1995. Available in: <ftp://ftp.cs.wpi.edu/pub/techreports/pdf/99-3.pdf>

SCHEIBLER, K. Using self modifying code under Linux (site). Disponível em <http://asm.sourceforge.net//articles/smc.html>. Acessado em 6 jan. 2008.

SHUTT, J.N. **Recursive Adaptable Grammars**. 1993. 149 p.. M.S. Thesis. Computer Science Department, Worcester Polytechnic Institute, Worcester, Massachusetts, 1993. (Emended 16 December 2003.)

SILBERSCHATZ, A.; GALVIN, P. B.; GAGNE, G. **Operating Systems Concepts**. 7th ed. John Wiley & Sons, 2005. 944 p.

STEELE, G. L. **- Common Lisp The Language**; 2nd ed. Digital Press, 1990, 1029 p.

TANENBAUM, A. S. **Sistemas Operacionais Modernos.** Prentice-Hall, 2003. 707 p.

TSCHUDIN, C.; YAMAMOTO L. **Harnessing Self-modifying Code for Resilient Software**. Innovative Concepts for Autonomic and Agent-Based Systems, vol. 3825/2006, LNCS, 2006, pp. 197-204. Springer-Verlag Berlin Heidelberg 2006.

VMWARE. Virtualization Basics. Disponível em < http://www.vmware.com/technology/virtualization.html>. Acessado em 5 jan. 2009.

YODER, J. W.; JOHNSON, R. **The Adaptive Object-Model Architectural Style.** Proceedings of the IFIP 17th World Computer Congress - Tc2 Stream / 3rd IEEE/IFIP Conference on Software Architecture: System Design, Development and

Maintenance, Deventer, The Netherlands. 2007. pp. 3-27. Available from: <http://www.adaptiveobjectmodel.com/WICSA3/ArchitectureOfAOMsWICSA3.pdf>.

APÊNDICE 1 – EQUIVALÊNCIA DE MODELOS DE CHAMADAS DE FUNÇÃO ADAPTATIVA

Ao longo dessa dissertação foi afirmado que realizar a chamada de funções adaptativas no estado é equivalente a realizar a chamada nas transições.
Antes de provar tal equivalência, suponha que:

- q_i, q_j designam dois estados (i.e. elementos) qualquer pertencentes ao conjunto de estados do autômato adaptativo (Q). Eventualmente, q_i e q_j podem denotar o mesmo estado ($q_i = q_j$);
- α designa um determinado símbolo pertencente ao alfabeto do autômato ou a cadeia vazia (ε);
- F é o nome-referência de uma determinada função adaptativa, F' é o nome-referência de outra função adaptativa que realiza a mesma ação adaptativa que F;
- $(q_i, \alpha) \rightarrow (q_j)$ [F●] é a representação de uma transição onde a chamada adaptativa ocorre antes da execução da transição (chamada da função adaptativa do tipo B). Deve ser interpreta como: de um determinado estado q_i do autômato, lê-se o símbolo α, executa-se a função adaptativa F e, em seguida, transita-se para o estado q_j (ou para outro estado pertencente a Q caso a função adaptativa modifique a declaração da transição);
- $(q_i, F')(\alpha) \rightarrow (q_j)$ é a representação de uma transição no modelo em que as chamadas as funções adaptativas ocorrem no estado (chamada da função adaptativa do tipo S). Deve ser interpretada como: no estado q_i, executa-se a ação adaptativa F' e, lendo-se o símbolo α, transita-se para o estado q_j (ou para outro estado pertencente a Q caso a função adaptativa modifique a declaração da transição). Ressalta-se que esse tipo de transição não é definido no autômato adaptativo.

Por questão de simplificação, estão sendo omitidas as informações de pilha e de sub-máquina corrente. Essa simplificação é adotada dado que chamadas as funções

adaptativas não são utilizadas nas transições de chamada e retorno de sub-máquina (NETO; PARIENTE, 2002).

Lema 1: Uma determinada transição onde a chamada a função adaptativa ocorre na transição (antes da execução da transição - tipo B) pode ser simulada por meio de uma ou mais transições onde a chamada ocorre no estado (tipo S).

Prova:

Provar tal equivalência remete a prova de que os modelos de máquinas finitas de Mealy e Moore são equivalentes, o que é apresentado em (BABCSÁNYI, 2000). A prova do Lema 1 é feito por meio da demonstração do processo de conversão de um modelo de notação, sem perdas de expressão.

Analisando a descrição do funcionamento de uma transição do tipo B com uma do tipo S observa-se que a diferença entre esses dois processos resume-se a ordem de execução da função adaptativa em relação à determinação da transição a ser executada.

Nas transições do tipo B determina-se a transição a ser executada e, em seguida, a função adaptativa é chamada. Já no caso das transições do tipo S, primeiramente chama-se a função adaptativa e, em seguida, determina-se a transição a ser executada.

Para converter do modelo de chamada de função adaptativa do tipo B, no caso (q_i, α) $\rightarrow$ (q_j) [F●], para transições do tipo S são necessários duas transições.

A primeira transita-se do estado q_i para um estado intermediário, criado com o propósito de realizar a conversão de modelos (denominado $q_{ij\alpha}$'). Essa transição simula o processo de determinar a transição antes de executar a transição.

A esse estado intermediário encontra-se associado à chamada a função adaptativa F', que realiza a mesma ação adaptativa associada a F, mas devidamente convertida para tratar adequadamente a topologia diferenciada.

O estado intermediário é o estado de origem de uma transição, vazia, que possui como estado destino q_j (estado destino da transição com chamada tipo B).

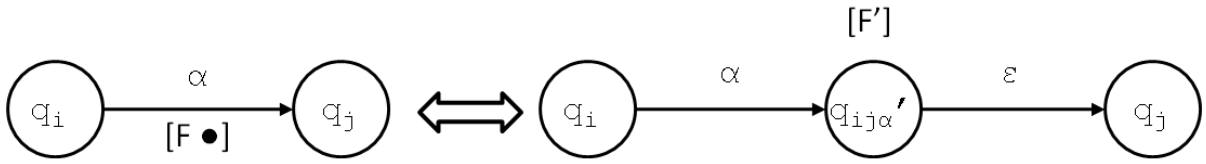


Figura 63 – Convertendo modelos de chamada de funções adaptativas: Chamada do tipo A → chamada no estado (tipo S).

Resumindo, uma determinada transição (q_i, α) → (q_j) [F●] (chamada a função adaptativa antes de executar a transição) pode ser simulada por meio de uma seqüência de transições (q_i, ε)(α) → ($q_{ij\alpha}$') ; ($q_{ij\alpha}$', F')(ε) → q_j (chamada a função adaptativa no estado), onde q_i' é um estado criado durante a conversão de modelos e F' representa a mesma ação adaptativa associada a F, ajustada para o modelo pós-conversão (esquema ilustrado na Figura 63).
Nota: Cada transição convertida utiliza um estado intermediário próprio (não compartilhado).

Lema 2: Uma determinada transição onde a chamada a função adaptativa ocorre no estado (tipo S) pode ser simulada por meio de uma ou mais transições onde a chamada ocorre na transição (antes da execução da transição – tipo B).

Prova:
Similar ao desenvolvido no Lema 1, a prova do lema 2 consiste em provar que é possível simular o comportamento de um modelo no outro modelo.
No entanto, diferentemente do desenvolvido na prova do Lema 1, ao converter do modelo de chamada de função adaptativa do tipo S para o tipo B é necessário primeiro executar a função adaptativa (convertida para tratar a diferença de topologia do autômato), por meio de uma transição a um estado intermediário e, em seguida, transitar com o símbolo em questão.
Portanto, uma transição do tipo (q_i, F)(α) → (q_j) – chamada do tipo S – pode ser simulado por meio da seqüência de instruções do tipo B (q_i, ε) → (q_i') [F' ●]; (q_i', α) → q_j, conforme ilustrado na Figura 64.
Nota: O estado intermediário (q_i') é comum a todas as transições convertidas que possuam o mesmo estado de origem.

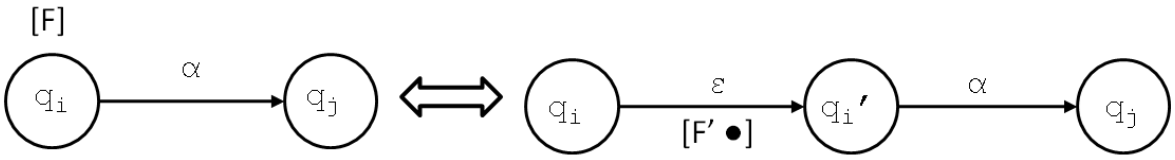


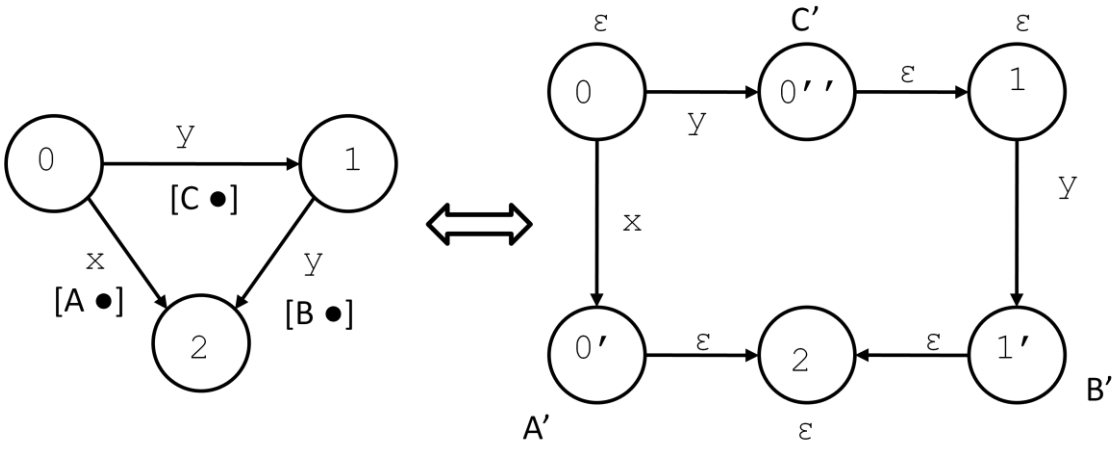
Figura 64 – Convertendo modelos de chamada de funções adaptativas: Chamada no estado (tipo S) → chamada do tipo B.

Lema 3: Caso a proposição dos autômatos adaptativos fosse modificada para admitir a chamada de função adaptativa dentro dos estados (chamadas a funções adaptativos tipo S), a mesma possuiria o mesmo poder de expressão do modelo com chamada a funções adaptativas nas transições.

Prova:

Para os autômatos adaptativos, é provado que utilizar apenas chamadas as funções adaptativas antes da execução da transição (freqüentemente chamadas de tipo B) é equivalente a realizar chamadas antes e/ou depois das transições (IWAI, 2000).

Para converter um modelo que utilize apenas chamadas do tipo B para um modelo que utilize apenas chamadas do tipo S, basta utilizar as propriedades de conversão definidos na prova do Lema 1 para cada uma das transições declaradas.



(a) (b)

Figura 65 - Esquema de demonstração de equivalência (a execução de (a) é equivalente a de (b)). (a) Modelo Original de execução: A transição (0, x)→2 [A●] representa transição interna com o símbolo x, invocando a ação adaptativa A após a execução da transição. (b) Modelo de invocação das ações adaptativas nos estados.

Para converter um modelo que utilize apenas chamadas do tipo S para um modelo que utilize apenas chamadas do tipo B, basta utilizar as propriedades de conversão definidos na prova do Lema 2 para cada uma das transições declaradas.
Um exemplo prático da conversão encontra-se ilustrado na Figura 65 (Figura 65 (a) é equivalente a Figura 65 (b)).

APÊNDICE 2 – PODER DE EXPRESSÃO DA LINGUAGEM ADAPTCODE

Lema 4: A máquina capaz de executar a linguagem AdaptCode possui o mesmo poder de expressão de uma máquina de Turing com as seguintes limitações: fita finita, tamanho do conjunto de estados (K) e do conjunto de símbolos (Σ) finitos, contanto que a declaração dessa máquina de Turing possa ser alocada na memória disponível no ambiente de execução.

Prova:

Para realizar tal demonstração, esse anexo descreve como criar um programa que simula o funcionamento de uma máquina de Turing com as restrições acima citadas.

Para simplificar a descrição, é adotada a hipótese que $| K | < 2^{32}-4$ e $| \Sigma | < 2^{32}-4$ (i.e. um estado ou um símbolo pode ser representado por uma palavra Double Word, utilizada para a representação de números inteiros de 32 bits).

Seguindo a hipótese acima definida, a fita pode ser representada como um vetor de palavras Double Word. Neste exemplo, a fita é representada pela variável inteira `fita[MAX]`, onde `MAX` é um inteiro que define o número máximo de posições que a fita possui. O ponteiro que indica a posição da fita é representado por meio de uma palavra Double Word (variável do tipo inteiro), nomeado de `pointer`, que armazena o índice apontado.

O conjunto de estados declarados $K = \{q_1, q_2, ..., q_n\}$ é codificado nos números inteiros pares e maior que 2. Portanto, na representação via linguagem AdaptCode, o conjunto K é declarado como $\{ 4, 6, ... , (4 + 2^n) \}$, contanto que $(4 + 2^n)$ não seja maior que o valor 2^{32} -1 (vide Tabela 2).

Tabela 2 – Codificação de símbolos

q_i	$(4+2^i)$
a_w	$(3+2^w)$
h	0
R	2
L	1

Similar ao feito para o conjunto K, o conjunto de símbolos $\Sigma = \{ a_1, a_2, ... a_w \}$ é codificado por meio de números inteiros impares maior que 1. Portanto, na representação via linguagem AdaptCode, o conjunto Σ é declarado como { 3, 5, ... , $(3 + 2^w)$ }, contanto que $(3 + 2^w)$ não seja maior que o valor 2^{32} -1 (vide Tabela 2).
O estado de *halt*, L (operação de mover o ponteiro para a esquerda) e R (operação de mover o ponteiro para a direita) são representados por 0, 1 e 2, respectivamente (conforme ilustrado na Tabela 2).
As operações especificadas pela máquina de Turing são traduzidas como:

- *halt* da máquina de Turing é traduzido pela instrução `halt` ou por um *loop* infinito;
- Movimentação do ponteiro para a esquerda é traduzido pela instrução **`sub`** `pointer,1`;
- Movimentação do ponteiro para a direita é traduzido pela instrução **`add`** `pointer,1`;
- A leitura do símbolo da fita é traduzido pela instrução **`mov`** `temp,fita[pointer]`;
- A escrita de um símbolo na fita é traduzido pela instrução **`mov`** `fita[pointer],<símbolo>`.

Adicionalmente, mais duas variáveis inteiras (posições de memória Double Word) são utilizadas, uma para armazenar o estado corrente, denominada `estado`, e outra, denominada de `temp`, para armazenar o símbolo lido.
Um elemento da δ, representado por $\delta (q_i,a_w) = (q_j, A)$, onde: q_i é um estado pertencente ao conjunto K; q_j ou é um estado pertencente ao conjunto K ou o estado h; a_w é um símbolo pertencente ao conjunto em Σ; A é utilizado para denotar ou as ações de movimentação do ponteiro da fita L e R ou a ação de escrita de um símbolo (declarado em Σ) na fita, é representado pelo código que descreve a seguinte lógica:

```
Se estado = qi então
     Se temp = aw então
```

```
Estado ← qj
Executa ação(L,R ou escrita)
```

Aplicando as regras acima descritas é possível escrever um programa que simula o comportamento de máquina de Turing no ambiente de execução da linguagem AdaptCode.

Uma vez demonstrado que é possível escrever um programa que simula o funcionamento da máquina de Turing, pode-se afirmar que o ambiente de execução tem poder de expressão de máquina de Turing.

Ilustração da demonstração do Lema 4:

Para ilustrar esse processo, considere o seguinte exemplo: a codificação da seguinte máquina de Turing: K = { q_0, q_1 }, Σ = { #, a }, s = q_0 e δ é declarado na Tabela 3.

Tabela 3 – Descrição da função de transição da máquina de Turing exemplo

K	Σ	δ(K, Σ)
q_0	a	q_1,#
q_0	#	h,#
q_1	a	q_0,a
q_1	#	q_0,R

Adicionalmente, é pressuposto que a fita contenha os seguintes símbolos #aaa#, sendo que o ponteiro encontra-se inicialmente localizado no primeiro branco (destacado pelo sublinhado).

Na linguagem AdaptCode, essa máquina é declarada como: K = { 4, 6 } ; Σ = { 3, 5 }, s = 4. Para esse caso, inicialmente a fita possui a seguinte configuração: `F[0] = 3, F[1 a 3] = 5 e F[4 ... MAX] = 3`.

```
mov estado,4   //estado ← s (estado inicial 4)
//pointer ← 0 (posição do ponteiro no primeiro
//elemento da fita
mov pointer,0
//label ciclo, utilizado para repetir o passo de computação
label ciclo
// verifica se o estado é halt
cmp estado,0
```

```
// caso seja, salta para halt_trat
je halt_trat
// verifica se o estado é igual a 4 (q0)
cmp estado,4
// caso não o seja, salta para e6
jne e6
//Le o símbolo da fita indicado por pointer e armazena em temp
mov temp,fita[pointer]
//Verifica se o símbolo lido é 3 (#)
cmp temp,3
//caso não o seja salta para e4s5
jne e4s5
//estado ← 6 (q1)
mov estado,6
// escreve 3 (#) na fita[pointer]
mov fita[pointer],3
//salta para label ciclo para próximo passo da computação
jmp ciclo
//label e4s5 – tratamento estado 4 símbolo 5
label e4s5
// verifica se o símbolo lido é 5 (a)
cmp temp,5
// caso não seja, salta para error
jne error
//δ(4,5)=0,3
//estado ← 0 (h)
mov estado,0
// escreve 3 (#) na fita[pointer]
mov fita[pointer],3
//salta para label ciclo para próximo passo da computação
jmp ciclo
//label e6 – tratamento caso estado 6
label e6
// compara se o estado = 6 (q1)
cmp estado,6
// caso não o seja salta para error
jne error
//Le o símbolo da fita indicado por pointer e armazena em temp
mov temp,fita[pointer]
//Verifica se o símbolo lido é 3 (#)
cmp temp,3
//caso não o seja salta para e6s5
jne e6s5
//δ(6,3)=4,5
//estado ← 4 (q0)
mov estado,4
// escreve 5 (a) na fita[pointer]
mov fita[pointer],5
//salta para label ciclo para próximo passo da computação
jmp ciclo
```

```
//label e6s5 – tratamento estado 6 símbolo 5
label e6s5
// verifica se o símbolo lido é 5 (a)
cmp temp,5
// caso não o seja, salta para error
jne error
//δ(6,5)=4,1
//estado ← 4 (q0)
mov estado,4
// ação 1, movimenta ponteiro para a direita
add pointer,1
//salta para label ciclo para próximo passo da computação
jmp ciclo
//label error, loop infinito
label error
jmp error
//label halt_trat, loop infinito
label halt_trat
jmp halt_trat
```