

Avaliação Parcial de Programas usando CMIX/II

Fernando M. Q. Pereira, Roberto S. Bigonha, Vladimir Di Iorio, Mariza A. S. Bigonha

Departamento de Ciência da Computação
Universidade Federal de Minas Gerais
Belo Horizonte - MG (Brazil)

Resumo

O processo de Avaliação Parcial é uma forma de tentar otimizar programas em relação a partes de sua entrada, criando programas mais específicos e portanto, geralmente mais eficientes. Este artigo mostra os resultados obtidos da aplicação da avaliação parcial de diversos algoritmos bem conhecidos no campo da Análise Numérica e do reconhecimento de cadeias de caracteres. Para a criação dos programas especializados, foi utilizado o avaliador parcial CMIX/II, que processa programas escritos na linguagem C. **Palavras Chaves:** Avaliação parcial, geração automática de programas, otimização, CMIX, análise Numérica, autônomos finitos determinísticos.

Abstract

Partial evaluation is a technique for program specialization. Programs are specialized with respect to parts of their input data, generating more efficient residual programs. In this work, we present results of partial evaluation of several numerical analysis and pattern matching algorithms. To generate the specialized programs, it was used a partial evaluator for the C language, known as CMIX/II. **Keywords:** partial evaluation, automatic program generation, optimization, CMIX, numerical analysis, pattern matching.

1 Introdução

Este artigo procura demonstrar a aplicabilidade das técnicas de avaliação parcial usando para isto o programa CMIX/II [2] e uma série de algoritmos, apresentados na Seção 1.1, bem conhecidos no campo da análise numérica e no reconhecimento de padrões léxicos. Por questões de espaço não será possível incluir o código de todos eles no artigo, mas o leitor interessado pode consultar [6].

A avaliação parcial de programas é uma técnica de geração automática de código cujo objetivo é tornar mais eficientes os programas que lhe são submetidos. Suponha um programa P que tenha duas entradas, identificadas como in_1 e in_2 . O resultado da avaliação parcial de P em relação à entrada in_1 é um novo programa P_{in_1} , designado por *residual* ou *especializado*. O programa P_{in_1} , quando executado sobre a entrada restante in_2 , produz o mesmo resultado que a execução de P sobre ambas as entradas. Avaliação parcial é um tipo de *especialização de programas*. A entrada in_1 é designada entrada estática e in_2 , *entrada dinâmica*. Dessa forma, o principal objetivo da avaliação parcial é o ganho em eficiência, pois, se parte dos dados de entrada de um programa é conhecida, as estruturas do programa que dependam apenas dessa parte podem ser previamente computadas e o programa especializado conterá apenas o código necessário para processar os dados ainda não conhecidos.

CMIX/II [2] é um programa *gerador de extensões de geração*, que processa programas escrito na linguagem C [5] e é implementado nessa mesma linguagem. Um gerador de extensões de geração não produz um programa especializado diretamente. A partir do programa alvo P e de uma especificação de sua entrada, CMIX/II produz como saída um avaliador parcial específico para P . Este último é chamado de *extensão de geração* para P . Uma extensão de geração de um programa P é um programa P_{gen} que, quando executado com um valor in_1 para a primeira entrada de P , gera um programa residual P_{in_1} . O programa P_{in_1} , é o resultado da avaliação parcial de P com valor in_1 para a primeira entrada.

```

double interp (int m, double* x, double* y, double z)
{int i, j; double p, r = 0;
  for (i=0; i<m; i++) { p = y[i];
    for (j=0; j<m; j++) if (i != j) p = p * ( (z-x[j]) / (x[i]-x[j]) );
    r = r + p; }
  return r; }

```

Figura 1: Algoritmo de interpolação de Lagrange, implementado em C

1.1 Programas Óbvios e Não Óbvios

Um programa P é chamado *óbvio* em relação à divisão de sua entrada entre estática e dinâmica, se seu fluxo de controle depende apenas dos valores da sua entrada estática. Isto significa que nunca são feitos testes de controle de fluxo sobre variáveis dinâmicas. As expressões condicionais são exemplos de testes de controle de fluxo. Estes programas constituem excelentes alvos para a avaliação parcial, desde que a entrada estática esteja corretamente definida. Exemplos de programas óbvios, bem como os dados que definem os seus fluxos de controle, neste artigo são:

- *algoritmos interpoladores* em relação ao número de pontos utilizados na interpolação;
- *algoritmo para resolução de sistemas lineares* em relação às dimensões da matriz principal;
- *algoritmo para multiplicação matricial* em relação às dimensões matriciais;
- *algoritmo para integração numérica* em relação ao grau do polinômio de integração.

Em geral, pode-se definir uma parte da entrada que torna óbvia a maior parte dos programas de natureza numérico-científica e com isto, os torna ótimos candidatos à avaliação parcial, uma vez que todo o esforço computacional sobre a entrada estática pode ser resolvido em tempo de sua especialização. Como exemplos têm-se a residualização de laços e o cálculo antecipado de endereços de memória. No entanto, muitos programas *não são óbvios* para a avaliação parcial, e isto pode levar a resultados imprevisíveis durante o processo de especialização. Por exemplo, como alguns testes de fluxo de controle são resolvidos por variáveis dinâmicas, o programa pode tomar diversos caminhos diferentes, o que leva geralmente à explosão de código no programa residual obtido. O código final pode se tornar muito grande, ou mesmo infinito, uma vez que todas as combinações possíveis de variáveis precisam ser levadas em consideração, mesmo que apenas alguns estados possam ser atingidos de fato. Mas, ainda assim, a avaliação parcial de programas não óbvios pode produzir bons resultados como será mostrado nas Seções 6 e 7. São exemplos de programas não óbvios neste artigo:

- *A máquina simuladora de autônomos finitos determinísticos* em relação à tabela de transições;
- *O reconhecedor de expressões regulares* em relação à expressão regular que o controla.

2 Algoritmos Interpoladores

Algoritmos interpoladores surgem da necessidade de se obter valores intermediários não constantes em uma tabela e geralmente se baseiam em polinômios interpoladores. Dados experimentais, tabelas estatísticas e de funções complexas são exemplos desta situação. É sabido que qualquer função real contínua pode ser aproximada por uma função polinomial. Esta técnica é útil para aproximar pontos com base em um conjunto discreto de valores de ordenadas e abscissas já conhecidos. Assim, a maior parte dos programas de interpolação, com base em uma tabela de pares ordenados tenta construir uma função polinomial que contém os pontos dados. Existe uma grande variedade de técnicas de interpolação polinomial, as mais conhecidas são aquelas que usam as fórmulas de *Lagrange* ou *Newton*.

2.1 Algoritmo de Lagrange

O polinômio de Lagrange é obtido pela fórmula: $L_n(x) = \sum_{i=0}^n y_i \prod_{j=0, j \neq i}^n \left(\frac{x-x_j}{x_i-x_j} \right)$.

Um programa interpolador, precisa, pois, receber como entrada um conjunto de ordenadas e abscissas, o valor da quantidade de pontos constantes nesta tabela e o valor a ser interpolado. A saída consiste na ordenada do ponto interpolado na tabela. Figura 1 ilustra uma implementação em linguagem C do algoritmo interpolador baseado no polinômio de Lagrange e a Figura 2 mostra o mesmo programa, transcrito em FCL, uma linguagem de fluxograma [1].

```

(m, x, y, z)
b0: r = 0; i = 0; goto b1
b1: if (i < m) goto b2 else goto b9
b2: p = y[i]; goto b3
b3: j = 0; goto b4
b4: if (j < m) goto b5 else goto b8
b5: if (i != j) goto b6 else goto b7
b6: p = p * ((z-x[j]) / (x[i] - x[j])); goto b7
b7: j = j + 1; goto b4
b8: r = r + p; i = i + 1; goto b1
b9: return r;

```

Figura 2: Algoritmo de interpolação de Lagrange escrito em FCL.

```

static double lagrange (double *x, double *y, double z)
{ double p; double r; r = (double) 0;
  p = y[0];
  p = p * ((z - x[1]) / (x[0] - x[1])); p = p * ((z - x[2]) / (x[0] - x[2]));
  p = p * ((z - x[3]) / (x[0] - x[3])); p = p * ((z - x[4]) / (x[0] - x[4]));
  r = r + p; p = y[1];
  p = p * ((z - x[0]) / (x[1] - x[0])); p = p * ((z - x[2]) / (x[1] - x[2]));
  p = p * ((z - x[3]) / (x[1] - x[3])); p = p * ((z - x[4]) / (x[1] - x[4]));
  r = r + p; p = y[2];
  p = p * ((z - x[0]) / (x[2] - x[0])); p = p * ((z - x[1]) / (x[2] - x[1]));
  p = p * ((z - x[3]) / (x[2] - x[3])); p = p * ((z - x[4]) / (x[2] - x[4]));
  r = r + p; p = y[3];
  p = p * ((z - x[0]) / (x[3] - x[0])); p = p * ((z - x[1]) / (x[3] - x[1]));
  p = p * ((z - x[2]) / (x[3] - x[2])); p = p * ((z - x[4]) / (x[3] - x[4]));
  r = r + p; p = y[4];
  p = p * ((z - x[0]) / (x[4] - x[0])); p = p * ((z - x[1]) / (x[4] - x[1]));
  p = p * ((z - x[2]) / (x[4] - x[2])); p = p * ((z - x[3]) / (x[4] - x[3]));
  r = r + p; return r; }
double lagrange_res (double *residual_arg, double *residual_arg1, double residual_arg2)
{ double r; r = lagrange (residual_arg, residual_arg1, residual_arg2);
  return r; }

```

Figura 3: Algoritmo de Lagrange especializado para uma tabela de cinco pontos.

Em muitas aplicações práticas a tabela permanece fixa para diversos pontos diferentes, portanto ela parece ser a melhor candidata à especialização automática. Esta, no entanto, pode ser uma opção errônea, ao se analisar o código mais friamente. No caso da especialização acontecer em relação à tabela de entrada, e em relação ao seu tamanho, naturalmente, apenas as computações no bloco **b6** e **b8**, da Figura 2 não poderão ser resolvidas em tempo de especialização. Os pontos presentes na tabela, contudo, são usados somente no bloco **b6**. Ou seja, o conteúdo da tabela é útil unicamente na eliminação do denominador $(x[i] - x[j])$ no bloco **b6**. Além do mais, o algoritmo resultante serve apenas para interpolar pontos daquela tabela passada em tempo de especialização, tendo, portanto, pouca relevância prática. Por outro lado, se a especialização acontecer em relação ao tamanho da tabela, o valor m , então o algoritmo resultante é muito mais geral e os ganhos computacionais são praticamente iguais ao que se teria conseguido com a especialização completa do código. A melhor opção é, portanto, que o tamanho da tabela seja escolhido como a entrada estática durante o processo de especialização do algoritmo. Isto posto, especializando o programa exibido na Figura 1 em relação ao número de pontos que deverão ser usados na geração do polinômio interpolador, chega-se ao código da Figura 3, gerado pelo avaliador parcial CMIX/II. Uma vez que a maior parte das computações, principalmente aquelas decorrentes de laços, foram completamente residualizadas, o programa final é bem mais eficiente que o programa original, como pôde ser comprovado via simulações com o programa `time-prog`, implementado junto com o avaliador parcial CMIX/II [2]. A Tabela 1 apresenta as

CMIX/II resultados da especialização			
Tamanho da tabela	tempo original	tempo residual	speedup
3	2.59s	1.55s	1.7
4	5.06s	2.7s	1.9
5	8.58s	4.51s	1.9
6	13.2s	6.77s	1.9

Tabela 1: Comparação entre desempenho dos programas original e especializado.

b4: s	b1: s
b5: s	b2: d
b6: 4d	b3: s
b7: s	<i>li</i> : (3s + 4d)m
	b8: s + d
(a)	(b)

Tabela 2: Computações estáticas e dinâmicas:(a)Interpolador de Lagrange, (b)Laço de Lagrange.

medidas obtidas pela execução de ambos os programas: residual e especializado em uma máquina SPARC executando sobre o sistema operacional Solaris 2.5.1. O *tempo original* se refere ao período gasto pelo programa original durante sua execução, ao passo que o *tempo residual* se refere ao período gasto pelo programa gerado pelo especializador em sua execução. O tamanho da tabela é dado em números de pontos necessários para construí-la e as medidas de tempo foram obtidas com a função `times()` definida na biblioteca `sys_times.h` da linguagem C [5]. O ganho de desempenho do programa residual sobre o programa especializado pode ser estimado por meio de uma análise quantitativa do código a ser residualizado. A codificação em FCL, apresentada na Figura 2, nesse caso possibilita uma boa visão a respeito do papel de cada ponto do programa em seu desempenho global. Uma análise deste tipo é feita de forma a ilustrar o que foi dito. A fim de separar computações dinâmicas das estáticas, são usadas duas notações de significado óbvio: **s** e **d** para estático e dinâmico, respectivamente. Assim, **bx: ns + md**, significa que no bloco *b_x* do programa em questão são executadas **n + m** operações, das quais **n** são estáticas e **m** são dinâmicas. Dessa forma, no laço mais interno do programa, que vai do bloco **b4** ao bloco **b7**, as operações estão divididas conforme mostra a Tabela 2(a). Observe que, nessa computação, está sendo considerado o número de operações e não a quantidade de linhas de código ou atribuições, portanto, no bloco **b6** ocorrem 4 operações que dependem de alguma forma de uma das variáveis dinâmicas. O laço mais externo, que vai do bloco **b1** até **b8** gera, por sua vez, as computações exibidas na Tabela 2(b) na qual, *li* é uma abreviação para laço interno e corresponde às operações calculadas na primeira computação da Tabela 2(a).

É interessante notar que o número de iterações em laços nesse programa é controlado pela variável **m**, ou seja, o tempo de execução do programa depende do número de pontos que constam na tabela de interpolação. Caso este valor se torne arbitrariamente grande, as operações que não fazem parte do laço tornam-se irrelevantes para o desempenho final do programa e, portanto, podem ser desconsideradas. A seguir vêm-se as fórmulas para os custos estáticos e dinâmicos. A ordem de complexidade quadrática se deve aos dois laços no programa: o primeiro que engloba os blocos **b4** a **b7** e o mais externo, que é formado pelos blocos de **b1** a **b8**: $C_s(P) = 3m^2s + 3ms$ e $C_d(P) = 4m^2d + 2md$

Computações estáticas no programa original podem ser resolvidas em tempo de especialização e não influem no desempenho do programa residual. O *speedup* relativo entre um programa residual e o programa especializado obtido dele pode ser então estimado pela equação: $SU(P) = \frac{C_s(P)+C_d(P)}{C_d(P)}$.

Para o exemplo apresentado nesta seção tem-se então a seguinte fórmula geral para o *speedup* estimado: $SU(P) = \frac{3m^2s+3ms+4m^2d+2md}{4m^2d+2md}$. Se for considerado ainda que **s** e **d** tenham o mesmo peso relativo, tem-se para a estimativa final: $SU(P) = \frac{7m^2+5m}{4m^2+2m}$ e, caso o tamanho da entrada estática tenda ao infinito, tem-se por fim: $\lim_{m \rightarrow \infty} SU(P) = \frac{7}{4} = 1.75$.

```

/* Parametros de entrada: * m: numero de pontos constantes na tabela
                          * x: matriz de doubles, abscissas dos pontos na tabela
                          * y: matriz ordenadas dos pontos na tabela
                          * z: valor da abscissa do ponto a ser interpolado. */
double interp (int m, double *x, double *y, double z)
{ int i, k; double r=0, *dely;
  dely = (double*) malloc (m * sizeof(double));
  for (i=0; i<m; i++) dely[i] = y[i];
  /* contrucao da tabela de diferencas divididas */
  for (k=1; k<m; k++)
    for (i=m-1; i>=k; i--) dely[i] = (dely[i] - dely[i-1]) / (x[i] - x[i-k]);
  /* avaliacao do polinomio de Newton atraves do metodo de Horner */
  r = dely[m-1]; for (i = m-2; i>=0; i--) r = r * (z - x[i]) + dely[i]; return r; }

```

Figura 4: Algoritmo de Newton implementado na linguagem C.

```

double interp_res (double *residual_arg, double *residual_arg1, double residual_arg2)
{ double r; double *x; double *y; double z; double r1; x = residual_arg;
  y=residual_arg1; z=residual_arg2; r1=(double) 0; cmixHeap4=y[0];
  cmixHeap3=y[1]; cmixHeap2 = y[2]; cmixHeap1 = y[3]; cmixHeap = y[4];
  cmixHeap = (cmixHeap - cmixHeap1) / (x[4] - x[3]);
  cmixHeap1 = (cmixHeap1 - cmixHeap2) / (x[3] - x[2]);
  cmixHeap2 = (cmixHeap2 - cmixHeap3) / (x[2] - x[1]);
  cmixHeap3 = (cmixHeap3 - cmixHeap4) / (x[1] - x[0]);
  cmixHeap = (cmixHeap - cmixHeap1) / (x[4] - x[2]);
  cmixHeap1 = (cmixHeap1 - cmixHeap2) / (x[3] - x[1]);
  cmixHeap2 = (cmixHeap2 - cmixHeap3) / (x[2] - x[0]);
  cmixHeap = (cmixHeap - cmixHeap1) / (x[4] - x[1]);
  cmixHeap1 = (cmixHeap1 - cmixHeap2) / (x[3] - x[0]);
  cmixHeap = (cmixHeap - cmixHeap1) / (x[4] - x[0]); r1 = cmixHeap;
  r1=r1 * (z - x[3]) + cmixHeap1; r1=r1 * (z - x[2]) + cmixHeap2;
  r1=r1 * (z - x[1]) + cmixHeap3; r1=r1 * (z - x[0]) + cmixHeap4; r=r1; return r; }

```

Figura 5: Algoritmo de Newton, especializado para uma tabela de cinco pontos.

2.2 Algoritmo de Newton

Assim como o polinômio de Lagrange, o polinômio de Newton também constitui um modo de interpolar pontos em uma tabela sem a necessidade de resolver um sistema de equações lineares. A fórmula deste polinômio envolve o operador de diferença dividida, definido recursivamente como: *ordem 0* : $\Delta^0 y_i = y_i = [x_i]$, *ordem n* : $\Delta^n y_i = \frac{\Delta^{n-1} y_{i+1} - \Delta^{n-1} y_i}{x_{i+n} - x_i} = [x_i, x_{i+1}, \dots, x_{i+n}]$

Usando este operador, tem-se para os polinômios de Newton: $P_n(x) = y_0 + \sum_{i=0}^n \Delta^i y_0 \prod_{j=0}^{i-1} (x - x_j)$. A Figura 4 mostra esta fórmula transcrita para um programa em linguagem C. Neste programa o polinômio é avaliado usando-se a técnica de Horner. Esta técnica se baseia na equação:

$P(x) = (((\dots (c_n x + c_{n-1} x) + c_{n-2}) x + \dots + c_2) x + c_1) x + c_0$, conhecida como *Fórmula de Horner*, cuja principal vantagem é reduzir o número de multiplicações necessárias para gerar os valores polinômiais. Como o exemplo da Seção 2.1, vê-se logo que o mais interessante, em relação à avaliação parcial, é que a especialização se dê sobre o tamanho da tabela, ficando o seu conteúdo na condição de entrada dinâmica.

Definidas as entradas estáticas e dinâmicas, após o programa da Figura 4 ser submetido ao avaliador parcial CMIX/II chega-se ao código especializado da Figura 5 para a entrada estática $m = 5$.

Afim de melhorar o desempenho, o avaliador parcial substituiu o vetor `dely`, Figura 4, alocado dinamicamente por variáveis temporárias simples de modo a evitar os custos da indexação de arranjos. Como o polinômio de Lagrange, este programa é mais eficiente que o programa que lhe deu origem.

CMIX/II resultados da especialização			
Tamanho da tabela	tempo original	tempo residual	speedup
2	1.25s	4.42s	1.9
3	1.96s	6.01s	1.7
4	2.86s	8.87s	1.9
5	4.19s	13.15s	1.9

Tabela 3: Comparação entre desempenhos dos programas original e especializado

```

int resolve (matriz a, vetor B, int m) {
    int pivot, x, y, z;
    real aux_a, aux_b, t;
    for (x = 0; x < m; x++) {
        for (y = pivot = x; y < m; y++)
            if (fabs(a[y][x]) > fabs(a[pivot][x])) pivot = y;
        if (!a[pivot][x]) return 0;
        if (pivot != x) {
            for (y = 0; y < m; y++) { aux_a = a[pivot][y]; a[pivot][y] = a[x][y]; a[x][y] = aux_a; }
            aux_b = B[pivot]; B[pivot] = B[x]; B[x] = aux_b; }
        for (y = x + 1; y < m; y++)
            if (a[y][x]) { t = a[y][x] / a[x][x];
                for (z = x; z < m; z++) a[y][z] = a[y][z] - t * a[x][z];
                B[y] = B[y] - t * B[x]; }
        for (y = x - 1; y >= 0; y--)
            if (a[y][x]) { t = a[y][x] / a[x][x];
                for (z = x; z < m; z++) a[y][z] = a[y][z] - t * a[x][z];
                B[y] = B[y] - t * B[x]; }
        for (x = 0; x < m; x++) if (a[x][x]) B[x] /= a[x][x]; return 1; }

```

Figura 6: Implementação do algoritmo para Sistemas Lineares na Linguagem C

A simulação revelou um ganho substancial no seu desempenho, pois as interpolações foram resolvidas em tempo quase 2 vezes menor que o programa original. A Tabela 3 mostra alguns resultados obtidos variando-se o tamanho da tabela que foi passado como entrada estática para o especializador.

Caso o ganho no desempenho do programa especializado sobre o programa residual seja estimado, como foi feito para o algoritmo de Lagrange, encontrar-se-á um valor de aproximadamente 1.7 para o *speedup* se o tamanho do parâmetro m , que é passado em tempo de especialização, for infinito. A equação $SU_s(m) = \frac{m^2+6m-5}{\frac{3}{2}m^2+\frac{5}{2}m-3} + 1$ fornece a razão entre as expressões de complexidade dos programas especializado e original. Note, no entanto, que para pequenos valores de m , o *speedup* pode ser bem diferente, pois os termos lineares dessa função terão uma influência maior no cálculo dos resultados. Ainda assim, a fim de justificar a diferença entre os resultados previstos e estimados, acredita-se que a quebra do vetor `dely` em variáveis independentes seja responsável por uma substancial parte dos ganhos no desempenho do programa especializado sobre o programa original.

3 Sistemas Lineares

Sistemas lineares são constantes em diversos ramos da Física e da Matemática. Problemas que envolvem a resolução de equações matriciais da forma $AX = B$ geralmente se originam de situações concretas, como da análise de circuitos elétricos e de sistemas mecânicos. Como estas equações podem tornar-se por demais complicadas, dado a quantidade de elementos presentes na matriz A , diversos algoritmos foram desenvolvidos, no sentido de resolver tais equações em tempo polinomial.

Um algoritmo para problemas deste tipo poderia receber, por exemplo, como parâmetros de entrada uma matriz quadrada A , um vetor de termos independentes B e um número inteiro especificando a ordem da matriz A , que deve ser igual ao tamanho da coluna do vetor B . Uma implementação deste algoritmo, na linguagem C, Figura 6, procura escalonar a matriz A , por meio de operações lineares: somando duas linhas, multiplicando uma linha por constante ou trocando a posição de duas linhas

```

int main (int argc, char **argv)
{ double a[15][15]; double b[15]; int i, j, m;
  FILE *fp;
  if (argc < 2) { fprintf(stderr, "Sintaxe: mat arquivo.\n"); exit (1); }
  if (!(fp = fopen(argv[1], "r")))
  { fprintf(stderr, "Erro, nao foi possivel abrir o arquivo %s.\n", argv[1]); exit (2); }
  fscanf(fp, "%d", &m);
  for (i = 0; i < m; i++)
    for (j = 0; j < m; j++) fscanf(fp, "%lf", &a[i][j]);
  for (i = 0; i < m; i++) fscanf(fp, "%lf", &b[i]);
  cmixGenInit(); matsis(a, m); cmixGenExit(stdout);
  return 0; }

```

Figura 7: Trecho de código inserido no programa gerado automaticamente por CMIX

```

int resolve_res(double *residual_arg)
{ int r; double *b; double aux_b;
  b = residual_arg; aux_b = b[2];
  b[2] = b[0]; b[0] = aux_b;
  b[1] = b[1] - -0.5000000000000000 * b[0];
  b[2] = b[2] - 0.2500000000000000 * b[0];
  b[2] = b[2] - -0.2999999999999999 * b[1];
  b[0] = b[0] - -1.2000000000000000 * b[1];
  b[1] = b[1] - 1.2500000000000000 * b[2];
  b[0] = b[0] - 5.6666666666666670 * b[2];
  b[0] = b[0] / 4.0000000000000000;
  b[1] = b[1] / 5.0000000000000000;
  b[2] = b[2] / 1.2000000000000000;
  r = 1; return r; }

```

Figura 8: Resolução de Sistemas Lineares: programa residual para uma matriz A de ordem 3 (3×3)

quaisquer na matriz. Estas operações são também aplicadas sobre o vetor B, ao final do que, estando a matriz A escalonada, tem-se no vetor B a solução do sistema linear.

Uma característica interessante desse tipo de problema é que freqüentemente o vetor de termos independentes sofre variações e a matriz A permanece imutável. Este fato torna os problemas lineares excelentes candidatos à especialização automática. Como a parte constante geralmente é constituída pela matriz principal, o mais indicado é que o programa seja especializado em relação a esta matriz. Note que, tendo em vista a posterior especialização do programa original, não é necessário que este prime pela eficiência, mas é importante que as partes estáticas e dinâmicas estejam bem definidas e separadas. Por isso, é necessário que sejam definidas duas variáveis auxiliares, uma para tratar os elementos da matriz estática A e outra para manipular os elementos do vetor B. Caso isso não seja feito, a especialização monovariante realizada pelo CMIX/II acaba definindo a matriz A como um elemento residual no programa especializado e a especialização não tem sentido.

Outro ponto importante a salientar é o fato de que o especializador CMIX/II não pode receber diretamente estruturas mais complexas, como matrizes, que representem a entrada estática do programa a ser especializado. Com efeito, este avaliador parcial aceita somente cadeias de caracteres ou números inteiros simples como parte da entrada estática dos programas que serão especializados. Assim, para que o programa pudesse ser especializado em relação a matriz A, foi preciso editar manualmente o código do gerador de extensões produzido pelo CMIX/II, inserindo a função `main` da Figura 7 no seu escopo. Nesse código foram incluídas as funções predefinidas da biblioteca CMIX `cmixGenInit()` e `cmixGenExit()`, e uma rotina para a geração do programa executável propriamente dito.

Estando pronta a extensão de geração, obtida via a especialização automática do algoritmo de resolução de sistemas lineares acima descrito, sua execução resultou no código apresentado na Figura 8, mais conciso e totalmente livre de qualquer referência à matriz A. Este código foi obtido pela execução

```

void mult_mat(double **matp, double **mat1, double** mat2, int m, int n, int p)
{#pragma cmix residual: mult_mat::i
  int i, /* varia as linhas da matriz mat1 */
      j, /* varia as colunas da matriz mat2 */
      k; /* varia as colunas da matriz mat1 */
  /* preenchimento da matriz matp com zeros */
  for (i = 0; i < m; i++) for (j = 0; j < p; j++) matp[i][j] = 0;
  /* multiplicacao das matrizes mat1 e mat2 */
  for (i = 0; i < m; i++) for (j = 0; j < p; j++)
    for (k = 0; k < n; k++) matp[i][j] = matp[i][j] + (mat1[i][k]) * (mat2[k][j]); }

```

Figura 9: Algoritmo de multiplicação matricial implementado na Linguagem C

da extensão de geração para a matriz A, a seguir, e para $m = 3$:

$$\begin{bmatrix} 1 & -3 & 2 \\ -2 & 8 & -1 \\ 4 & -6 & 5 \end{bmatrix}$$

Nas simulações o programa especializado provou ser cerca de 4.1 vezes mais rápido que o programa original. Este substancial ganho em eficiência é intuitivamente percebido quando se compara os dois códigos, original e especializado, pois no último não há qualquer referência à matriz A, uma vez que esta fora completamente residualizada durante o processo de avaliação parcial.

4 Multiplicação Matricial

O produto da matriz A de dimensão $n \times p$ por uma matriz B ($p \times m$) é uma matriz $C = AB$ ($n \times m$) tal que: $c_{ij} = \sum_{k=1}^p a_{ik}b_{kj}$ e $1 \leq j \leq m$. Esta operação tem amplo uso, principalmente no relacionamento entre tabelas. Note que o produto matricial é definido apenas para matrizes de dimensões compatíveis, onde o número de colunas da primeira matriz é igual ao número de linhas da segunda.

A implementação de um algoritmo para multiplicar matrizes cujas dimensões são compatíveis é trivial. Ele recebe como parâmetros de entrada as 3 dimensões matriciais relevantes na multiplicação e as 3 matrizes de números reais: a matriz que conterà o resultado da multiplicação matricial; a primeira matriz fator na multiplicação; e a segunda matriz fator na multiplicação. Isto faz com que estes parâmetros sejam particularmente propícios à especialização dada a grande quantidade de computações que poderiam ser realizadas durante a obtenção do programa residual. Espera-se, dessa forma, que o programa residual contenha somente referências às posições matriciais, não havendo variáveis auxiliares para controlar seu fluxo. Submetido o algoritmo descrito à especialização, com as dimensões matriciais: $m = 2$, $n = 2$, $p = 2$, Figura 9, foi obtido o programa residual 10. Como era esperado, a residualização promoveu a computação dos laços em tempo de especialização. Esta computação prévia é a grande responsável pelos ganhos em eficiência do programa residual, pois este se atém apenas aos cálculos aritméticos envolvendo as próprias matrizes, e, obviamente, aos cálculos de endereços de posições matriciais. A fim de comparar o desempenho de ambos os programas: residual e original, foi criado um gerador de matrizes aleatórias, Figura 11. Neste gerador existe uma função que gera números aleatórios do tipo real compreendidos entre o intervalo especificado como seu argumento; e uma função que serve apenas para inicializar o gerador randômico. Na fase de testes, as matrizes foram geradas e multiplicadas 50.000 vezes, para isto usou-se a ferramenta `time-prog` [2], incluída no pacote *CMIX/II*. Os dados coletados estão dispostos na Tabela 4. Nesta tabela os valores m , n , p referem-se, respectivamente: m , o número de linhas da primeira matriz parcela na multiplicação; n , o número de colunas da primeira matriz e número de linhas da segunda; e, p , o número de colunas da segunda matriz. Como pode ser observado, o desempenho do programa residual é de 2 a 3 vezes superior ao desempenho do programa original. Isto se deu porque grande parte do esforço computacional aconteceu durante a especialização do programa. Nesta etapa cálculos referentes aos laços foram realizados, e variáveis de controle foram completamente residualizadas, desaparecendo do programa final.


```

static void mult_mat (double **matp, double **mat1, double **mat2)
{ matp[0][0] = (double) 0; matp[0][1] = (double) 0;
  matp[1][0] = (double) 0;   matp[1][1] = (double) 0;
  matp[0][0] = matp[0][0] + mat1[0][0] * mat2[0][0];
  matp[0][0] = matp[0][0] + mat1[0][1] * mat2[1][0];
  matp[0][1] = matp[0][1] + mat1[0][0] * mat2[0][1];
  matp[0][1] = matp[0][1] + mat1[0][1] * mat2[1][1];
  matp[1][0] = matp[1][0] + mat1[1][0] * mat2[0][0];
  matp[1][0] = matp[1][0] + mat1[1][1] * mat2[1][0];
  matp[1][1] = matp[1][1] + mat1[1][0] * mat2[0][1];
  matp[1][1] = matp[1][1] + mat1[1][1] * mat2[1][1];
  return;
}
void mult_mat_res (double **residual_arg, double **residual_arg1, double **residual_arg2)
{ mult_mat (residual_arg, residual_arg1, residual_arg2); return; }

```

Figura 10: Multiplicação de Matrizes especializado em relação às dimensões matriciais

```

double **monta_matriz(int seed, int m, int n)
{ int i, j; double **mat;
  mat = (double**) malloc (m * sizeof(double*));
  for (i=0; i<m; i++) mat[i] = (double*) malloc (n * sizeof(double)); random_reset (seed);
  for (i = 0; i < m; i++) for (j = 0; j < n; j++) mat[i][j] = rand_double (0, 32767);
  return mat;}

```

Figura 11: Gerador de Matrizes Randômicas

<i>CMIX/II resultados da especialização</i>					
<i>m</i>	<i>n</i>	<i>p</i>	tempo original	tempo residual	speedup
13	4	4	6.13s	2.7s	2.9
5	5	5	3.53s	1.23s	2.3
5	4	3	0.64s	1.83s	2.9
8	8	8	13.07s	6.68s	2.0

Tabela 4: Desempenho dos programas original e especializado para diversas dimensões matriciais

```

double integ (double a, double b, int n)
{ char p, cm, im, i; double h, x, y, s = 0, integral;
  int ioup, ck, c[24], d[8] = {2, 6, 8, 90, 288, 840, 17280, 28350};
  c[0] = 1;      c[1] = 1; c[2] = 4; c[3] = 1;   c[4] = 3;
  c[5] = 7; c[6] = 32; c[7] = 12;
  c[8] = 19; c[9] = 75; c[10] = 50;
  c[11] = 41; c[12] = 216; c[13] = 27; c[14] = 272;
  c[15] = 751; c[16] = 3577; c[17] = 1323; c[18] = 2989;
  c[19] = 989; c[20] = 5888; c[21] = -928; c[22] = 10496; c[23] = -4540;
  p = ( n*(n+2) + n%2 ) / 4 - 1;
  im = (char) n/2; cm = p + (im);
  if (a > b) { x = a; a = b; b = x; }
  h = (b - a)/n; x = a; ioup = !((n+1)%2);
  for (i=0; i <= n; i++)
  { y = f(x); ck = c[cm - (char) fabs(im + (i/(im + 1))*ioup - i)];
    s = s + (y * ck); x = x + h; }
  integral = n * (h/d[n-1]) * s;
  return integral; }

```

Figura 12: Implementação do Método de Newton-Cotes de Integração Numérica.

5 Integração Numérica

Integração é uma das mais importantes ferramentas matemáticas, sendo vastamente empregada no cálculo de grandezas físicas para as quais é possível criar um modelo numérico. Assim, para uma certa função $f(x)$, integrável no intervalo $[a, b]$, tem-se: $\int_a^b f(x)dx = F(b) - F(a)$, sendo $F'(x) = f(x)$. Porém, pode ser que a forma analítica de $F(x)$ seja difícil de se obter, ou talvez sejam conhecidos apenas valores discretos de $f(x)$. Nesse caso, é preciso usar métodos numéricos para avaliar a integral de $f(x)$. Esses métodos consistem em aproximar a função $f(x)$ por um polinômio interpolador e determinar analiticamente a integral desse polinômio no intervalo $[a, b]$. Dentre os algoritmos de integração mais conhecidos, existe o que usa a equação $I_n = \frac{nh}{d_n} \sum_{i=0}^n c_i y_i$, conhecida por *Fórmula de Newtons-Cotes*, onde n é o grau do polinômio interpolador usado na integração e os valores c_i são chamados coeficientes de Cotes, sendo tabelados de acordo com o grau do polinômio usado. O valor d_n é conhecido na tabela, havendo um valor diferente para cada n . Por fim, os valores y_i são as ordenadas obtidas da função que se quer integrar, apenas verificando-se o seu valor para abscissas conhecidas.

Um algoritmo para calcular integrais usando a equação dada se presta a calcular a integral da função definida por $f(x)$, no intervalo $[a, b]$, por uma fórmula de Newton-Cotes gerada a partir de um polinômio de grau n , $1 \leq n \leq 8$. Os coeficientes de Cotes estão dispostos em um arranjo, de forma a economizar algum espaço, já que alguns deles se repetem simetricamente. Dado o grau do polinômio, um apontador indica a posição a partir da qual estão os coeficientes de Cotes no arranjo. Para obter bons rendimentos com a especialização, novamente é preciso definir sobre que parte dos dados de entrada se dará a otimização. Este algoritmo, Figura 12, recebe três parâmetros: o limite inferior do intervalo de integração, a ; o limite superior do mesmo intervalo, b ; e o grau do polinômio interpolador que será usado na integração, n . O prévio conhecimento do intervalo de integração economiza um mínimo de computação. Aliás, apenas o esforço de calcular o valor da diferença $a-b$ é economizado, o que em termos computacionais, é irrelevante. Por outro lado, caso o grau do polinômio interpolador seja conhecido, todo o esforço computacional gasto no laço principal do programa pode ser economizado, desde que o laço seja limitado por aquele valor. Assim, foi escolhida como entrada estática o grau do polinômio interpolador n . O programa residual obtido a partir da especialização deste algoritmo em relação a n foi obtido, Figura 13. Os ganhos são relevantes, como pode ser visto a partir da análise da Tabela 5. O intervalo de integração escolhido foi $[1, 2.7818]$, e a função $f(x)$ usada foi $\ln(x)$.

```

static double integ (double a, double b)
{ double h; double x; double y; double s; double integral;
  s = (double) 0;
  if (a > b) { x = a; a = b; b = x; }
  h = (b - a) / 2.0000000000000000;
  x = a; y = (double) 1 / x; s = s + y * 1.0000000000000000;
  x = x + h; y = (double) 1 / x; s = s + y * 4.0000000000000000;
  x = x + h; y = (double) 1 / x; s = s + y * 1.0000000000000000; x = x + h;
  integral = 2.0000000000000000 * (h / 6.0000000000000000) * s;
  return integral; }
double integ_res (double residual_arg, double residual_arg1)
{ double r; r = integ (residual_arg, residual_arg1); return r; }

```

Figura 13: Programa de Integração especializado para $n = 4$.

<i>CMIX/II resultados da especialização</i>			
n	tempo original	tempo residual	speedup
2	7.24s	1.99s	3.6
3	9.21s	2.54s	3.6
4	11.5s	2.97s	3.9
5	13.4s	3.43s	4.0
6	15.63s	3.92s	4.0
7	18.28s	4.42s	4.1
8	19.99s	4.72s	4.2
9	21.93s	5.1s	4.3

Tabela 5: Integração Numérica: desempenho dos programas original e especializado.

6 Autômatos Finitos Determinísticos

Autômatos Finitos Determinísticos (AFD's) são um importante formalismo na Teoria da Computação, dado que são úteis no reconhecimento e na representação de linguagens regulares. Um algoritmo capaz de simular um AFD no reconhecimento de um padrão de caracteres recebe como entrada: uma tabela contendo todas as possíveis transições entre estados presentes naquele autômato; o estado inicial, a partir do qual a palavra de entrada começa a ser consumida; um conjunto de estados finais; e uma palavra de entrada, especificando o padrão que se deseja reconhecer. A saída deste algoritmo é um valor booleano indicando se a palavra de entrada foi reconhecida ou não. Note que uma palavra é reconhecida se a sua computação terminar em um estado pertencente ao conjunto de estados finais do AFD. A Figura 14 mostra um algoritmo desse tipo codificado na linguagem C.

Observe que este programa recebe duas entradas: uma estrutura descrevendo o AFD, e um vetor de caracteres, que contém o padrão a ser reconhecido. Portanto, este algoritmo se presta a simular qualquer tipo de autômato finito determinístico. Um programa mais eficiente, contudo, pode ser obtido a partir deste ao custo da perda da generalidade. Basta que o algoritmo seja adaptado para um AFD específico. A obtenção de tal programa pode ser feita automaticamente, por meio da técnica de *Avaliação Parcial de Programas*. A fim de obter bons resultados com a avaliação parcial, é preciso saber para qual parte da entrada o programa deve ser avaliado. Esta análise é mais produtiva quando

```

typedef struct { char d[NUM_ST][NUM_SIMB], /* tabela de transicao de estados */
                i, /* estado inicial do AFD */
                f[NUM_ST]; /* conjunto de estados finais */
} AFD;
int AFD_sim (AFD *m, char *a)
{ char e = m->i, j=0;
  while (a[j] != '\0') { e = m->d[e][a[j] - 'a']; j++; }
  if (m->f[e]) return 1; else return 0; }

```

Figura 14: Algoritmo para Reconhecimento de Padrões implementado na Linguagem C.

```

(&m, a)
(b0)
b0: e = m->i;goto b1
b1: if (a[j] != '\0') goto b2 else goto b3
b2: e = m->d[e][a[j] - '\a']; j = j + 1; goto b1
b3: if (m->f[e]) goto b4 else goto b5
b4: return 1;
b5: return 0;

```

Figura 15: Algoritmo de Reconhecimento de Padrões escrito em FCL.

```

int AFD_sim (AFD *m, char *a)
{ #pragma cmix residual: AFD_sim::j /* o valor da variavel j deve ser */
  char e = m->i, j=0, aux; /* residual, do contrario ocorrera */
                               /* especializacao infinita. */

  while (a[j] != '\0')
  { if (a[j] == 'a') /* Estas comparacoes constituem um truque */
    e = m->d[e][0]; /* de especializacao, de modo que a variavel*/
  else if (a[j] == 'b') /* e podera continuar estatica apos a BTA*/
    e = m->d[e][1]; /* uma vez que assume somente valores */
  else /* conhecidos durante a especializacao */
    return 0; j++; }
  if (m->f[e]) return 1; else return 0; }

```

Figura 16: AFD adaptado para obter melhores resultados quando submetido a um avaliador parcial.

se conhece de antemão os vários estados que se pode alcançar a partir de um certo valor dos parâmetros de entrada. A proposta nesse caso, é obter um programa específico para um certo AFD, o que sugere que a especialização se dê sobre a estrutura que descreve o autômato.

Para simplificar a análise do algoritmo dado na Figura 14, o mesmo é codificado na linguagem FCL, conforme mostra a Figura 15. Esta nova codificação tem a vantagem de explicitar os vários estados que podem ser alcançados durante a execução do programa.

A análise de tempo de definição definirá a variável j como sendo estática pois, uma vez inicializada com o valor zero ela é apenas incrementada no decorrer do programa, assumindo, dessa forma, somente valores conhecidos. No entanto, em uma análise mais rigorosa do código que aparece na Figura 15, percebe-se que esta variável pode crescer sem limites, já que ela está limitada somente por um valor dinâmico, como se pode verificar na comparação apresentada no ponto **b1** do programa. Este crescimento sem limites levará a um laço de especialização infinito, existindo um novo estado de programa para cada um dos infinitos valores de j . Para contornar este problema, é preciso forçar a variável j a ser definida como uma variável dinâmica. No caso do avaliador parcial CMIX/II, isto é feito via “pragmas de especialização”, como será visto mais à frente.

Uma segunda análise do código em FCL revela ainda um outro problema que pode aparecer no momento da geração dos estados alcançáveis. Para percebê-lo, basta observar a primeira atribuição no bloco **b2**. Note que a segunda coluna da tabela de transição de estados do AFD está indexada pelo valor contido no padrão a , que pela definição inicial do tipo dos valores de entrada, é dinâmico. Assim, em tempo de especialização não é possível determinar o valor do índice na tabela de transições. Esta tabela, então, seria definida pelo especializador como uma estrutura dinâmica, que deveria permanecer no programa residual. Isto é tudo que se quer evitar, uma vez que o objetivo inicial proposto com a especialização foi o de resolver todas as computações que dependessem unicamente do AFD em questão. Para contornar esse problema, é preciso limitar manualmente os valores que aquele índice pode assumir. Na verdade, este é um artifício muito empregado em avaliação parcial, tendo inclusive denominação própria: “*the trick*”. Feitas as modificações no código original, chega-se finalmente ao programa da Figura 16, que se encontra melhor adaptado para ser submetido a um avaliador parcial.

Uma nova análise de tempo de definição sobre o código dado definirá a variável e e a estrutura m como estáticas e a variável j e o padrão a como dinâmicos. Logo, espera-se que as variáveis e e m não existam no código residual que será gerado com a avaliação automática. Isto de fato pode ser observado

```

static int AFD_sim(char *a)
{ char j; j = '\0';
L2: if (a[j] != '\0') { if (a[j] == 'a') j = j + 1; else { if (a[j] == 'b') { j = j + 1;
L1:     if (a[j] != '\0') { if (a[j] == 'a') { j = j + 1;
L3:     if (a[j] != '\0') { if (a[j] == 'a') { j = j + 1; goto L1;
        } else if (a[j] == 'b') j = j + 1; else return 0;
        } else return 0;
        } else if (a[j] == 'b') { j = j + 1; goto L2;
        } else return 0;
    } else return 0;
} else return 0;
} if (a[j] != '\0') { if (a[j] == 'a') { j = j + 1; goto L2;
    } else if (a[j] == 'b') { j = j + 1; goto L3;
    } else return 0;
} else return 0;
} else return 1; }

```

Figura 17: Programa obtido via especialização para palavras com um número par de a's e b's

<i>CMIX/II Resultados da Especialização</i>			
Tamanho do Padrão	Tempo Original	Tempo Residual	speedup
2	1.83s	1.39s	1.3
4	2.82s	2.01s	1.4
8	5.46s	3.97s	1.4
16	10.06s	7.22s	1.4
32	18.98s	13.58s	1.4
64	37.51s	26.61s	1.4

Tabela 6: Resultados comparativos do reconhecedor de padrões original e do programa especializado.

no código mostrado na Figura 17, que foi gerado pela aplicação do avaliador parcial *CMIX/II* sobre o algoritmo exibido na Figura 16. O AFD passado em tempo de especialização reconhece padrões sobre o alfabeto *a*, *b* com um número par de *a*'s e *b*'s.

A aplicação do avaliador parcial transformou todas as consultas à tabela de transições de estados em expressões condicionais no programa residual. Isto é como se o simulador de AFD's tivesse sido compilado para um programa escrito em linguagem C. O programa residual é também ligeiramente mais rápido que o programa original, pois ele verifica o padrão diretamente, sem a necessidade de consultas à tabela de transições de estados do autômato. Conseguiu-se os resultados práticos exibidos na Tabela 6, executando-se os programas residual e original. O tamanho da entrada dinâmica é dado em termos do número de caracteres do padrão que é passado para o AFD. Medidas de tempo de execução foram obtidas com a função `times()` definida na biblioteca `sys_times.h` da linguagem C.

7 Reconhecedor de Expressões Regulares

Reconhedores de padrões são programas que identificam conjuntos de caracteres que atendem a uma certa forma geral. Esta forma geralmente é definida como uma expressão regular, composta pelas operações de união, concatenação e fecho de Kleene. Algoritmos para reconhecer padrões com base em expressões regulares são implementados como máquinas de estados finitos, onde cada estado é alcançado por transições partindo de outros estados. Um programa desta natureza recebe como entrada uma tabela contendo as especificações de transição entre os diversos estados componentes da máquina finita e, obviamente, o padrão que se busca reconhecer. Expressões regulares permitem o reconhecimento de diversas palavras diferentes, podendo mesmo identificar uma classe infinita de conjuntos de caracteres. A expressão '*(a+b)*' por exemplo, é capaz de identificar tanto o padrão '*a*' quanto o padrão '*b*'. Assim, para testar as diversas possibilidades de reconhecimento, o programa precisa ser implementado com uma estrutura que possibilite retrocesso. Isso pode ser feito com funções recursivas, ou então por meio de uma estrutura de dados tipo pilha. O programa mostrado na Figura 18

```

typedef struct { char ch; int n1, n2; } estado;
int match(estado *st, char *a)
{ #pragma cmix residual: match::j
  int n1, n2, item, j=0, N = strlen(a), state = st[0].n1; inicia_dq (); put(scan);
  while (state) { if (state == scan) { j++; put(scan); }
                 else if (st[state].ch == '*') { n1 = st[state].n1; n2 = st[state].n2;
                 push(n1); if (n1 != n2) push(n2); }
                 else switch(a[j]) {
                     case 'a': if (st[state].ch == 'a') put (st[state].n1); break; }
  if (vazia() || (j == N)) return 0;
  state = pop(); } return j; }

```

Figura 18: Simulador de Autômato Finito não Determinístico, implementado em C

foi implementado usando uma estrutura de dados chamada *deque*, que mistura conceitos de fila e de pilha. As funções `pop()` e `push()` que aparecem nesta figura colocam e retiram um elemento na estrutura como seria feito em uma pilha, ao passo que a função `put()` coloca elementos na estrutura como é feito em uma fila.

Como se pode notar, este programa já está preparado para a especialização automática, pois os valores estáticos e dinâmicos estão completamente separados na implementação mostrada na Figura 18. Em particular, o `switch` na última expressão condicional serve para limitar os possíveis valores que o padrão pode assumir. Este algoritmo serve apenas para reconhecer o padrão formado por uma ocorrência do caractere `a`, mas o princípio pode ser facilmente generalizado para qualquer linguagem formal que possa ser reconhecida por AFN: as chamadas *Linguagens Regulares*. Voltando à discussão iniciada na Seção 6, a variável `j` da Figura 18 precisa ser forçada como uma variável dinâmica, a fim de evitar um laço de especialização infinito. A especialização deste programa pode ser levada adiante sem maiores problemas, estando ele implementado da forma apresentada. Com a especialização espera-se que a tabela de estados seja completamente residualizada, e também a estrutura *deque*, uma vez que esta acumula somente valores conhecidos durante a consumação de um padrão qualquer.

A aplicação do avaliador parcial sobre este programa leva de fato à completa residualização da tabela de transições. A estrutura *deque*, por sua vez não foi residualizada. Ao contrário, as funções de inserção de elementos na fila foram separados em diversas subfunções, cada uma responsável pela inserção de um único elemento sobre a *deque*. As funções geradas pela especialização do programa simulador de AFN, tendo como entrada dinâmica o padrão `'a\0'` e o programa residual obtido não são mostrados neste artigo. No caso do programa residual, a explosão de código ocorre devido ao vasto leque de possibilidades geradas pela técnica não determinística na avaliação da expressão regular. Como cada configuração possível da pilha deve ser levada em consideração durante a avaliação parcial, diversos estados diferentes têm de ser previstos, mesmo para um padrão minúsculo como `'a\0'`. Caso o padrão reconhecido pelo AFN discutido neste exemplo fosse mais complexo, certamente outras funções semelhantes àsquelas descritas acima teriam sido geradas.

8 Conclusão

Este artigo mostrou substanciais ganhos de desempenho, decorrentes da geração de códigos especializados a partir da aplicação de um avaliador parcial sobre programas comuns. A avaliação parcial é uma técnica viável para a geração automática de programas eficientes e pode ser vantajosa em diversos casos, destacam-se: a aplicação sobre funções cujos parâmetros variem em diferentes taxas; a otimização de programas separados em diversos módulos; a geração automática de compiladores.

Existem funções que possuem parâmetros de entrada que variam muito, em execuções seguidas, ao passo que outros permanecem fixos, ou variam muito pouco. Este pode ser o caso do programa que resolve equações matriciais, cujo processo de avaliação parcial foi descrito na Seção 3, uma vez que em muitas situações típicas, a matriz principal de sistema (matriz **A**) permanece estática enquanto os vetores independentes (vetor **B**) são alterados a cada execução do código. Nesse caso, a avaliação parcial é vantajosa, pois possivelmente a soma do tempo de geração do programa residual, mais o tempo de diversas execuções desse programa seria inferior ao tempo de se executar o código original diversas

vezes. Ainda nesse caso, o leque de potenciais aplicações é extremamente vasto: (1) reconhecimento de padrões; (2) computação gráfica, com “Ray Tracing”; (3) redes neurais; (4) busca em bancos de dados; (5) computação científica; (6) simulação de circuitos de hardware, etc.

Em segundo lugar, a avaliação parcial tem grande utilidade na programação modular, aumentando a eficiência de programas projetados em módulos. Embora o projeto de um código fonte em diversos módulos seja vantajoso em termos de flexibilidade e facilidade de programação em paralelo, ele exige um custo do programa final assim construído, em termos de eficiência, pois este gasta uma certa quantidade de tempo chamando blocos separados e transportando dados. A avaliação parcial, no entanto, permite a compressão de módulos, por meio de expansão de chamadas intermodulares e da propagação de constantes para os lugares onde serão usadas de fato, além de realizar a precomputação sempre que possível. Por fim, é importante dizer que é possível utilizar avaliadores parciais tanto como compiladores quanto como geradores de compiladores. Imagine, por exemplo, que um avaliador parcial receba um interpretador de programas e um código fonte a ser interpretado. Embora não seja tão trivial perceber isto, o resultado da avaliação parcial seria um programa executável obtido a partir da compilação do código fornecido. Esta situação é bem semelhante àquela demonstrada na Seção 6, no qual uma tabela de transições para um autônomo finito determinístico pôde ser “compilada” para um programa independente da máquina simuladora.

Assim, analisando diversos casos, e partindo dos resultados obtidos, este artigo permite concluir que a avaliação parcial é uma técnica de geração automática de programas que produz bons resultados em termos de otimização de código, além de ter ampla aplicação em áreas teóricas da ciência da computação, como por exemplo, no caso da geração de compiladores.

Referências

- [1] Vladimir O. Di Iorio, Roberto S. Bigonha, Mariza A. S. Bigonha. *Tutorial: Avaliação Parcial de Programas*, Technical Report LLP 05/2000, 34 pages, Laboratório de Linguagens de Programação, UFMG, 2000.
- [2] Equipe do Projeto C-MIX. *C – MIX/II User and Reference Manual*. Manual de referência, DIKU, Denmark, Outubro de 1999
- [3] Frederico F. Campos, Filho. *Cálculo Numérico*. UFMG, Brasil, Fevereiro de 1998
- [4] Neil D. Jones, Carsten K. Gomard e Peter Sestoft *Partial Evaluation and Automatic Program Generation*. Prentice Hall, London, UK, 1993.
- [5] Kerningham, B. and Ritchie, D. *The C Programming Language*. Prentice Hall, 1988.
- [6] Fernando M. Q. Pereira, Roberto S. Bigonha, Vladimir Di Iorio, Mariza A. S. Bigonha, *Avaliação Parcial de Programas usando CMIX/II*, Relatório Técnico do Laboratório de Linguagens de Programação, LLP001/2001, do Departamento de Ciência da Computação, ICEx, UFMG, 2001.