DECIMA CONFERENCIA LATINOAMERICANA DE INFORMATICA

23 - 28 ABRIL '84
VIÑA DEL MAR - CHILE

004.06
C 748d

• UNIVERSIDAD CATOLICA DE VALPARAISO
CENTRO LATINOAMERICANO DE ESTUDIOS EN INFORMATICA. CLEI

X CONFERENCIA LATINOAMERICANA de INFORMATICA

VIÑA DEL MAR ABRIL 1984

documentos de trabajo

CONSTRUÇÃO E OTIMIZAÇÃO DE PROGRAMAS PARA PROCESSAMENTO DE ARQUIVOS SEQÜENCIAIS, UMA APLICAÇÃO DO MÉTODO DOS TRANSFORMADORES DE DATOS

R.C.B. Martins, C.J.P. de Lucena, P.A.S. Veloso
Universidade Católica de Rio de Janeiro - BRASIL

1. INTRODUÇÃO

Programas resolvem problemas. Veloso [1] define um problema como sendo uma estrutura $P = \langle D,O,q \rangle$ onde os elementos de D são os dados para o problema, os elementos de O são os possíveis resultados e p é uma relação binária entre D e O representando as condições do problema.

Um programa resolve um problema P se este programa define uma função f de D em O tal que

$$\forall d \in D \quad q(d;f(d)) \qquad (1)$$

Define-se espaço solução de um problema com o conjunto de todas as soluções, isto é, $\Sigma(P) = [f \in F(P) | f \leq q]$.

Derivar um programa de acordo com o método dos transformadores de dados consiste em, dada uma especificação para os objetos de entrada $d \in D$ e para os objetos de saída $o \in O$, construir um programa tal que (1) seja verificada.

Os métodos de derivação de programa apoiados em análise de dados, como o de Jackson, têm o mesmo procedimento ao tentar de início derivar um mapeamento direto entre a estrutura interna de d e a estrutura interna de o onde $(d;o) \in p$. Exatamente ao tentar este mapeamento direto o método de Jackson encontra dificultades em resolver certas classes de problemas ditas classes de problemas potencialmente com retroação ("backtracking") e classes de problemas potencialmente com conflito de estruturas (conflito de froteira, conflito de estrutura, conflito de ordem). O método dos transformadores de dados propõe uma forma canônica de programação que inclui trivialmente os problemas que fogem ao padrão da metodologia básica de Jackson [2].

2. O MÈTODO DOS TRANSFORMADORES DE DADOS PARA ARQUIVOS SEQUENCIAIS

O método dos transformadores de dados começa ao se expressar as noções abstratas de $d \in D$ e $o \in O$, ao invés de se tentar expressar as representações de dados dessas duas entidades. Esta forma de trabalho, embora usual em outros métodos de programação, não encontra forte aceitação em métodos de análise de dados (vide metodologia de Constantine e Yourdon [4]). A estratégia de derivação de programas pelo método dos transformadores de dados consiste na aplicação de decomposição e redução de problemas utilizando-se dos mecanismos de constructores de tipos de dados de Hoare. Redução e decomposição de problemas são aplicados de forma a se ter uma coleção de problemas solúveis por Jackson que substitua o problema original.

Uma redução do problema $P_0 = \langle D_0, O_0, q_0 \rangle$ ao problema $P_1 = \langle D_1, O_1, q_1 \rangle$, é um par de funções P_{01} = (ins, rec) onde

insere é uma função unária ins: $D_0 \rightarrow D_1$
recupera é uma função unária rec: $O_1 \rightarrow O_0$

tal que para todo $f_1 \in \Sigma(P_1)$ rec. f_1. ins $\in \Sigma(P_0)$. Uma condição suficiente para o par (ins,rec) ser uma redução de P_0 a P_1 é rec.q'. ins$\leq$q onde rec.q'.ins = $[(d;o) \in D_0 \times O_0 \,|\, \exists o_1 \in O_1, \quad o = rec(o_1) \wedge (ins(d);o_1) \in q_1]$.

O primeiro passo do método dos transformadores de dados para arquivos sequenciais consiste em definir D_1 e O_1 como o produto cartesiano de D com O isto é $D_1 = O_1 = D \times O$ e as funções insere e recupera com $\forall d \in D$ ins(d) = $(d;\Lambda)$ e $\forall d_1 \in D_1$ $\forall o \in O$ rec(d_1,o)= d onde Λ é um elemento bem determinado de O (normalmente o elemento Λ é feito igual ao arquivo vazio). Em outras palavras, a redução definida por [ins,rec] faz uso do mecanismo de construtor de dados produto cartesiano (registro) definido por Hoare [3]. Intuitivamente esta redução é proposta para se evitar o problema de conflito de estruturas que ás vezes ocorre quando se aplica o método básico de Jackson, pois agora a entrada tem trivialmente a mesma estrutura da saída. A figura 2.1. exemplifica a solução.

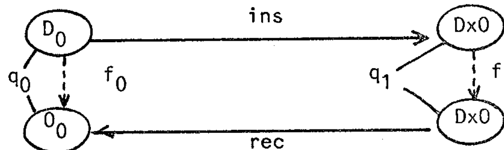

Figura 2.1.

O segundo passo do método é uma nova redução T_{12} = [constrôi, último] do problema $P_1 = \langle D_1, O_1, q_1 \rangle$ ao problema $P_2 = \langle D_2, O_2, q_2 \rangle$ onde

$$D_1 = O_1 = D \times O$$

$$D_2 = O_2 = D_1^*$$

constrôi : $D_1 \rightarrow D_1^*$ e tal que contrôi a sequencia unitária de um determinado argumento

último : $D_1^* \rightarrow D_1$ e tal que retira o último elemento de uma determinada sequência.

Intuitivamente, esta redução é proposta para se evitar o problema de retroação ("backtracking") pois o mecanismo de sequência (também proposto por Hoare) torna dispo-

nível na entrada os elemento necessários ao processamento.

A figura 2.1 fica agora com o aspecto apresentado na figura 2.2.

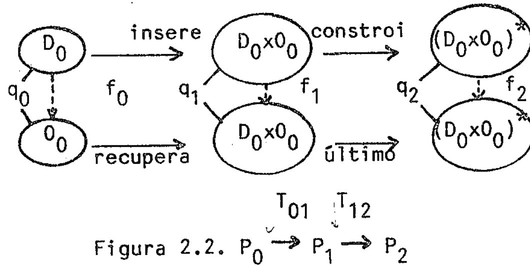

Figura 2.2. $P_0 \xrightarrow{T_{01}} P_1 \xrightarrow{T_{12}} P_2$

Propõe-se agora um esquema padrão de programa que decompõe o problema P_2 em uma série de problemas menores que possam ser volúveis por Jackson. Este esquema é diagramado na figura 2.3 que complementa a figura 2.2.

Uma proposta de programação para o diagrama da figura 2.3. é o programa esquema apresentado na figura 2.4 numa linguagem do gênero PASCAL.

A função altera é definida como
altera (x_3) = adiciona(x_3, transforma (último (x_3)))
onde transforma é uma função de DxO em DxO, a ser definida, que contribui para a solução de P_2 e adiciona adiciona a uma sequência um elemento do mesmo tipo.

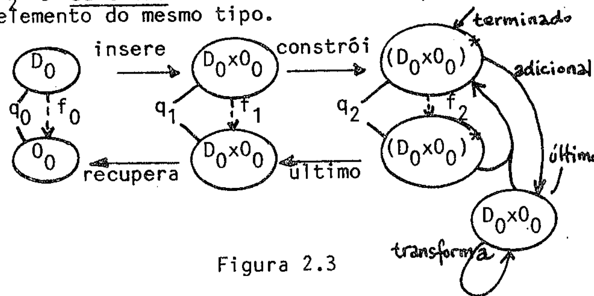

Figura 2.3

```
Program esquema;
type D = seq of objeto1;
     O = seq of objeto2;
   DxO = record i:D;
                r:O
         end;
  (DxO)* = seq of DxO;
var x,d:D;
    y,o:O;
begin
    x ← copia(d);
    f;
    o ← copia(y)
end;
Procedure f;
var x1 y1 :DxO;
begin
    x1.i ← x;
    x1.r ← Λ;
    f1 ;
    y ← y1.r
end;
Procedure f1;
var x2,y2: (DxO)*;
begin
    x2 ← constrói(x1);
    f2;
    y1 ← último(v2)
end;
Procedure f2;
var x3:(DxO)*;
begin
    x3 ← x2;
    while terminado(x3) do
        x3 ← altera (x3);
    y2 ← x3
end;
```

Figura 2.4

Um critério de correção para esquema poderia ser expresso por

i. altera (x_3) = adiciona(x_3,transforma(último(x_3)))

ii. $\forall x_3$ $(DxO)^*$, smllr(transforma(x_3).i,x_3.i)

iii. smllr é uma relação bem fundada em DxO tal que qualquer $d \in D$ está numa cadeia finita como começando em Λ:
smllr(Λ, d_1) smllr(d_1,d_2) ...
smllr(d_n,d)

iv. $\forall x_3$ $(DxO)^*$, terminado $(x_3) \Longrightarrow$
q_2(x_3,constrói(d.i,Λ))

v. $\forall x_3$ $(DxO)^*$,q_2(x_3,constrói(d.i,))
$\Longrightarrow q_0$(último(x_3).r,d)

Intuitivamente, a relação smllr garante que em cada passo a função transforma contribui um pouco para a solução do problema. A relação smllr, que e uma relação bem fundada, caracteriza o elemento vazio, Λ, com um elemento distinguido que é necessariamente alcançado para permitir a terminação do programa.

A condição (v) garante que, quando o programa pára, x_3 é a solução do problema P_2 cuja entrada é obtida da original pela aplicação de insere e constrói. A condição (vi) explicita que a redução garante que a solução de q_2 acarreta a solução de q_0.

3. O PROBLEMA DAS CADEIAS LIMITADAS POR CARACTERES

Jackson em seu livro "Principles of program design" propõe o seguinte problema (pg 130 - Problema 10):

Um programa deve ser projetado para analisar uma cadeia de caracteres, reconhecendo e imprimindo duas subcadeias S1 e S2. S1 termina com o caracter "C" e S2 com o caracter "&" e a cadeia toda com o caracter "%".

Na entrada são disponíveis dois ítens: a cadeia completa e um apontador para uma determinada posição da cadeia. S1 é definida como sendo a cadeia cujo primeiro caracter é o apontado pelo apontador inicialmente e cujo último caracter é o "C"; S2 é a cadeia cujo primeiro caracter é o caracter seguinte ao "C" e cujo último caracter é o "&". Qualquer dos dois caracteres "C" ou "&" pode estar faltando.

É dado que a cadeia total possui o caracter "%" como um dos 100 primeiros caracteres contados a partir do primeiro. Porém não se sabe se as cadeias S1 e S2 estão presentes. Se ambas estiverem presentes e corretas deve ser impresso o seguinte relatório:

```
     CADEIA PERFEITA
S1 = xxxxxC
S2 = yyyyy&
```

Do contrário a cadeia deve ser impressa a partir do ponto inicialmente marcado da seguinte forma:

```
     CADEIA IMPERFEITA
CARACTER-001 = x
CARACTER-002 = y
     .
     .
     .
CARACTER-nnn = %
```

4. A SOLUÇÃO DO PROBLEMA

Como primeiro passo deve-se fazer a caracterização abstrata de D e O. Pode-se fazer isto ao criar-se o programa cadeia como exemplificado na figura 4.1.

```
Programa cadeia;
type D   = file of char;
     O   = relatorio
     DxO = record i:D;
                  r:O
           end
    (DxO)* = file of (DxO);
var x,d:D;
    y,o:O;
begin
    x←copia(d);
    primoreduz(x,y);
    o← copia(y)
end.
```

Figura 4.1

O procedimento primoreduz é o responsável pela redução T_{01} = [ins,rec] e a expressão do procedimento é a da figura 4.2.

```
Procedure primoreduz(x:D;var y:O);
var x1,y1: DxO;
begin
    x1.i←x;
    x1.r←Λ;
    segundoreduz(x1;y1);
    y←y1.r
end
```

Figura 4.2

As atribuições com os seletores i e r fazem o papel do par de funções ins(de insere) e rec(de recupera). O procedimento segundoreduz é o responsavel pelo segundo passo da obtenção da forma canônica do programa e sua codificação extremamente semelhante a do primoreduz está na figura 4.3.

As funções auxiliares constroi e último têm respectivamente que construir uma sequência unitária a partir do registro fornecido e retirar o último registro da sequência apresentada.

Neste ponto da programação tem-se a primeira parte da forma canônica, a obtida por redução, pronta e pode-se passar para a primeira decomposição expressa pelo procedimento primodecomp na figura 4.4.

```
Procedure segundoreduz(x1:DxO;var y1:DxO);

var x2,y2: (DxO)*
begin
    x2← constrói(x1);
    primodecomp(x2;y2);
    y1← último(y2)
end;
```

Figura 4.3.

```
Procedure primodecomp(x2:(DxO)*;var y2:(DxO)*
var x3:(DxO)*
begin
    While comprimento(último(x3).i)≠0 do
          x3←altera(x3);
    y2← x3
end;
```

Figura 4.4

A função altera, figura 4.5, completa a forma canônica do programa onde apenas foi caracterizado o predicado terminado.

```
Procedure altera(x3:(DxO)*):(DxO)*;

var y3:(DxO)*;
    x4:(DxO);
begin
    y3← x3;
    x4←último(y3);
    x4←transforma(x4);
    altera← adiciona(y3,x4)
end;
```

Figura 4.5

A função adiciona adiciona ao final de uma sequência um elemento construindo uma sequência de comprimento maior.

A função transforma completa o desenvolvimento da forma canônica na figura 4.6.

```
Procedure transforma(x4:DxO):DxO;
var x5,x6:D;
    y5,y6:O;
begin
    x5←x4.i;
    y5←y5.r;
    processa(x5,y5,x6,y6);
    transforma.i←x6;
    transforma.r←y6
end;
```

Figura 4.6.

O que a forma canônica permite até o momento é a prova de terminação do programa desde que o procedimento processa garanta que comprimento $(x_6) <$ comprimento (x_5). O problema de construir o relatório fica bem mais simplificado pois a forma canônica permite que o relatório seja construído caracter a caracter.

O procedimento processa passa a ter a função de construir o relatório y_6 a partir da entrada x_6. A construção deste relatório depende do estado do problema: estado "indefinido" quando não tiver sido lido o caracter "%".

- estado "certo" quando tiver sido lido os caracteres "C" e "&"
- estado "errado" quando tiver sido lido o caracter "%" sem ter sido lido os caracteres "C" e "&".

A leitura e retirada de um caracter de x_6 determina o estado do problema e a construção do relátorio.

```
Procedure processa (x5:D;y5:0; var x6:D;
                                   var y6:0);
type "estado" = ("Λ","C","c","e");
     "relatório: case estado of
                 "Λ": record C: file of char;
                             &: file of char;
                             %: file of char;
                      end;
                 "C": record C: file of char;
                             &: file of char;
                             %: file of char;
                      end;
                 "c"  record C: file of char;
                             &: file of char;
                      end;
                 "e"  record %: file of char
                      end
                 otherwise
                 end;
var tipo: "estado";
    letra:char;x6:"relatório";
begin
    letra:=primeiro(x5);
    x6:=final(x5);
[cálculo do estado]
          case letra of
          "C": if tipo   =Λ[não definido]
               then tipo = "C"
               else tipo = Λ ;
          "&": if tipo   = "C"
               then tipo = "c" [correto]
               else tipo = Λ ;
          "%": if tipo   = "c"
               then tipo = "c"
               else tipo = "e"; [erro]
          otherwise
               estado    = Λ
          end;
[cálculo do relatório]
          case tipo of
          "Λ": begin
                x6.C:= adiciona(x5.c, letra);
                x6.%:= adiciona(x6.%, letra)
               end;
          "C": begin
                x6.C:= x5.C
                x6.&:= adiciona(x6.&, letra)
                x6.%:= adiciona(x6.%, letra)
              end;
          "c": begin
                x6.C:= x5.C
                x6.&:= x5.&
               end;
          "e": begin
                x6.%:= x5.%
               end;
          otherwise
          end
end;
```

Figura 4.7

Observe-se que só em processa pode-se definir os tipos relatório e estado que até o instante tinha um significado intuitivo. Observe que a sequência de relatórios é completamente heterogênea, mas, quando o problema termina, o faz com o relatório certo, isto é, com a cadeia ruim ou com as cadeias boas.

Processa também garante que o comprimento de x_6 é sempre estritamente menor que x_5, isto é, a prova da terminação pode ser feita e o comprimento zero é alcancado no estado "c" ou "e" garantindo assim a correção total do problema.

5. TRANSFORMAÇÃO DA FORMA CANÔNICA EM UM PROGRAMA OPERACIONAL

5.1. Introdução

Nesta parte do trabalho tenta-se mostrar caminhos para a transformação dos programas obtidos pelo método dos transformadores de dados em programas operacionais.

Segundo Jackson [2] o termo otimização é uma palavra mal escolhida para expressar esta transformação. Otimizar um programa significa torná-lo o menor e o mais rápido possível. Com isto, geralmente consegue-se torná-lo de difícil entendimento, de difícil manutenção e, finalmente, mais vulnerável a erros. Aliás, a receita de Jackson para "otimizar" programas é:

1ª: Não faça
2ª: Não faça ainda

A idéia contida no método dos transformadores de dados concorda, essencialmente com as regras básicas de Jackson. Não faça "otimização" da cópia canônica do programa. Mantenha-a, pois todo o trabalho de verificação da correção parcial e da terminação do algoritmo depende dela. A partir desta copia é que, por transformações, caso a caso, obtémse novas cópias do programa. Cada transformação deve ser cuidadosamente justificada e vai depender de uma série de fatores como: software de suporte, conversão de entrada e saída, dados típicos, etc Staa [5].

A primeira consideração sobre transformação, com vistas á procura de uma versão operacional tem a ver com a noção de forma conônica de programas. Chegar a esta forma no processo de programação acarreta na criação de "overheads" de memória e tempo que podem e devem ser eliminados. A partir da forma canônica, a análise da solução a ser adotada talvez deva seguir a proposta de Knuth [6] reforcada em Staa [5] propoese o seguinte para otimização de programas gerados pelo método dos transformadores de dados:

PASSO 1: Toma-se a cópia da forma canônica do programa provada correta, ou melhor, correta por construção, documenta-se esta cópia e executam-se testes que evidenciem o desempenho do programa.

PASSO 2: A partir da cópia canônica do programa eliminam-se os elementos redundantes, necessários apenas ao processo de teste.

PASSO 3: A partir da análise de elementos como redução de necessidades de memória,

necessidades de tempo de C.P.U., software de suporte, etc, aplicam-se as transformações necessárias ao programa.

Com exceção dos passos 1 e 2 que podem ter uma formulação geral o passo 3 deve ser estudado caso a caso.

5.2 Otimazação da Forma Canônica

Seja a forma canônica de um programa para processamento de arquivos gerado pelo método dos transformadores de dados, apresentado na Figura 5.1

```
Program esquema:
type D =
     0 =
     (Dx0) = record i:D;
                    r:0
             end;
     (Dx0)* = seq of (Dx0)*;
var  d,x:D;
     o,y:0;
     x4,x1,y1:Dx0;
     y3,x2,y2,x3:(Dx0)*
procedure altera:(Dx0)*;
     begin
          y3←x3;
          x4←último(x3);
          x4←transforma(x4);
          altera←adiciona(y3,x4)
     end [altera];
procedure f2;
     begin
          x3←x2;
          while comprimento(último(x3).i) ≠ 0
               do x3←altera(x3);
          y2←x3
     end [f2];
procedure f1;
     begin
          x2←constroi(x1);
          p2;
          y1←último(y2)
     end [f1];
procedure f;
     begin
          x1.i←x;
          x1.r←Λ;
          p1;
          y←y1.r
     end [f];
begin
     x←copia(d);
     p;
     o←copia(y)
end  [esquema].
```

Figura 5.1

Pode-se notar de imediato que as duas reduções definidas por [insere;recupera], [constrôi,último] ocorrem obrigatoriamente e com isto os procedimentos f e f_1 poderiam ser reduzidos a um único procedimento que chamaríamos de reduz e o programa esquema passaria a ter a forma da figura 5.2.

```
Prógrâm esquema;
type D =
     0 =
     (Dx0) = record i:D;
                    r:0
             end
     (Dx0)* = seq of (Dx0);
var d,x:D;
     o,y:0;
     x4,x1,y1:Dx0;
     y3,x2,y2,x3:(Dx0)*;
procedure altera (Dx0)*;
     begin
          y3←x3;
          x4←último(x3);
          x4←transforma(x4);
          altera←adiciona(y3,x4)
     end [altera];
procedure f2;
     begin
          x3←x2
          while comprimento(último(x3).i) ≠
          0 do x3←altera(x3);
          y2←x3
     end [f2];
procedure reduz;
     begin
          x1.i←x;
          x1.r←Λ;
          y2←constrôi(x1);
          p2;
          y1←último(y2)
          y←y1.r;
     end [reduz];
begin
     x←copia(d)
       reduz
     o←copia(y)
end [esquema].
```

Figura 5.2

Como num programa de processamento de arquivos, já testado, a sequência de computação de torna desnecessária, os procedimentos f_2 e altera podem ser fundidos e o tipo (Dx0)* eliminado passando a ter-se apenas disponível a última componente processada na computação e o programa esquema passa a ser o da Figura 5.3.

```
Program esquema;
type D =
     0 =
     (Dx0) = record i:D;
                    r:0
             end
var  d,x:D;
     o,y:0;
     x3,x2,y2:(Dx0);
procedure altera;
     begin
          x3←x2;
          while comprimento(x3.i) ≠ 0 do
               x3←trasforma(x3);
          y2←x3
          end [altera];
procedure reduz;
     begin
          x2.i←x;
          x2.r←Λ;
          altera;
          y←y2.r
          end [reduz];
begin
     x←copia(d);
     reduz;
     o←copia(y);
end [esquema].
```

Fifura 5.3

Finalmente, pode-se eliminar a chamada dos procedimentos altera e reduz e ter-se um único corpo de programa e o esquema passa a ser o da Figura 5.4.

```
Program esquema;
type D =            ;
     0 =            ;
     (Dx0) = record i:D;
                    r:0
             end;
var  d,x:D;
     o,y:0;
     x2:Dx0;
begin
     x←copia(d);
     x2.i←x;
     x2.r←Λ;
     while comprimento(x2.i) ≠ 0 do
           x2←transforma(x2);
     y← x2.r;
     o← cópia(y)
end [esquema].
```

Figura 5.4

Evidentemente a nova versão de esquema é bem mais eficiente tanto em tempo de computação quanto em utilização de memória, mas em contrapartida, toda a visibilidade quanto à sequência de elementos computados, e a prova de terminação deixaram de ficar auto documentados. O procedimento transforma que é específico de cada problema terá de ser estudado caso a caso.

5.3 Considerações sobre a Otimização dos Procedimentos Transforma e Processa para o Problema de Análise de Cadeia.

A consideração inicial é apenas compatibilizar variáveis e eliminar a chamada do procedimento transforma e aí o procedimento processa. Com isto o programa esquema passa a ter a configuração da figura 5.5.

```
Program esquema;
type D =
     0 =
     (Dx0) = record i:D;
                    r:0
             end;
var  d,x:D
     o,y:0
     x2:Dx0;
begin
     x← copia(d);
     x2.i←x;
     x2.r←Δ ;
     while comprimento(x2.i) ≠ 0 do
          processa(x2.i, x2.r);
     y←x2.r
     o←cópia(y)
end [esquema].
```

Figura 5.5

O procedimento processa, a menos das trocas de variáveis para compatibilizar os nomes pode ser aproveitado praticamente na sua codificação oritinal. O programa tem a forma final apresentada na figura 5.6, a menos das funções copia, primeiro, final comprimento, e está bem perto da implementação do PASCAL 6000.

```
Program esquema:
type estado = ("Λ","C","c","e");
     relatório = case estado of
                 "Λ": record C: file of char;
                             &: file of char;
                             %: file of char
                      end;
                 "C": record C: file of char;
                             &: file of char;
                             %: file of char
                      end;
                 "c": record C: file of char;
                             &: file of char
                      end;
                 "e": record %: file of char
                      end
                 otherwise
                 end;
     D = file of char;
     0 = relatório;
     Dx0 = record i:D;
                  r:0
           end;
var  d,x:D:
     o.y:0;
     x2:Dx0;
Procedure processa(var x2.i:D, var x2.r:0);
var  tipo:char;
     letra: char;
begin
     letra:=primeiro(x2.i);
     x2.i:=final(x2.i);
     case letra of
                 "C": if tipo = "Λ"
                      then tipo = "C"
                      else tipo = "Λ";
                 "&": if tipo = "C"
                      then tipo ="c"
                      else tipo = "e";
                 "%": if tipo = "c"
                      then tipo = "c"
                      else tipo = "e"
                 otherwise
                           tipo = "Λ"
                 end;
case tipo of
          "Λ": begin
                  x2.r.C: = adiciona
                  (x2.r.C,letra);
                  x2.r.%: = adiciona
                  (x2.r.%, letra)
               end;
          "C": begin
                  x2.r.C: = x2.r.C;
                  x2.r.&: = adiciona
                  (x2.r.&,letra);
                  x2.r.%: = adiciona
                  (x2.r.%,letra)
               end;
          "c": begin
                  x2.r.C: = x2.r.C;
                  x2.r.&: = x2.r.&
               end;
          "e": begin
                  x2.r.%: = x2.r.%
               end;
          otherwise
          end;
end;
     x←copia(d);
     x2.i←x;
```

```
      x2.r ← Λ;
      while comprimento(x2.i) ≠ 0 do
            processa(x2.i;x2.r);
      y ← x2.r
      o ← copia(y)
end   [esquema].
```

Figura 5.6

5.4 Outras Considerações

Para cada problema, ou melhor para cada programa solução de um problema, tem-se uma maneira de se otimizar o código. Não se apresenta aqui a forma de documentar as diversas fases do proceso de otimização por que isto foge ao escopo deste trabalho.

6. CONCLUSÕES

O método dos transformadores de dados procura reduzir o problema da programação de aplicações do tipo processamento de arquivos sequenciais, através de uma mudança na formulação do problema, a uma série de transferências simples dos dados de entrada para a saída. Nada mais do que as características da saída desejada determinam as operações que serão aplicadas aos dados de entrada ao longo do processo de transferência.

A colocação do problema de programação nesses termos simplifica substancialmente o processo de solução de problemas do gênero processamento de arquivos sequencias, além de introduzir vantagens adicionais tais como: fácil determinação da terminacao de programas (condicao de parada), padronização dos programas produzidos através do método, facilidade de manutenção de programas e clareza da documentação.

A redução do problema a uma série de transferências simples, requer uma primeira transformação básica na sua formulação. Requeremos que o programa que está sendo especificado tenha acesso permanente aos dados de entrada associados a todos os estágios das saídas produzidas. Para isto, o método transforma os dados de entrada do problema no produto cartesiano dos dados de entrada com os dados de saída. Mais especificamente. o programa que será a solução do problema encarará como entrada o objeto usualmente especificado como entrada, conjugado ao objeto usualmente especificado como saída(inicialmente um objeto vazio) e como saída o objeto usualmente especificado como entrada (finalmente um objeto vazio) conjugado ao objeto produzido como saída (após a execução completa do programa). A figura 6.1 ilustra o que acabamos de escrever.

O método torna explícito um raciocínio usual em programação: o programa termina quando se esgota a entrada e a saída está completa

O programa P será decomposto pelo método em n programas menores, cada um com o objetivo de construir un elemento de saída. A rigor, o que fazemos neste estágio é prever a necessidade de se projetar um conjunto de programas capaz de manipular a sequência de situações que ocorrem desde a entrada transformada até a sua modificação passo a passo em uma saída transformada. A formulação do problema que prevê a construção da saída elemento a elemento através de um número de programas iguais a este número de elementos é a transformação final que permite reduzir o problema inicial a uma série simples de transferências guiadas pelas características das saídas desejadas. A figura 6.2 ilustra a nova formulação do problema.

O método dos transformadores de dados reduziu o problema á especificação de um programa P_i que produz um registro de saída por vez. No exemplo dado o programa P_i é o procedimento processa.

O método dos transformadores de dados também se inspira na idéia de que, para um produto software que terá um longo ciclo de vida, a existência de um modelo preciso e legível, como o produzido pelo método, é tão importante quanto a existência de uma implementação eficiente para o mesmo. Isto é particularmente verdade quando este modelo ou especificação é um programa executável. O procedimento descrito na seção 5, permite, através de refinamentos (transformações) a passagem do modelo produzido pelo método para uma implementação eficiente do mesmo. O método deverá futuramente, ser auxiliado por um software especial de apoio ao desenvolvimento de programas, que até o presente momento já permite a conversão da forma canônica em um programa PASCAL e a sua execução.

APÊNDICE

Glosário de Funções

copia - copia o argumento produzindo uma nova instância do tipo

primeiro - exibe o primeiro elemento de uma sequência, i.e., first($\langle l_1, l_2, \ldots, l_n \rangle$) = primeiro ($\langle l_1, l_2, \ldots, l_n \rangle$) = l_1 e a sucessão original não é modificada

retira - exibe e remove o primeiro elemento de uma sucessão

último - exibe o último elemento de uma sucessão, i.e., last ($\langle l_1, l_2, \ldots, l_n \rangle$) = último ($\langle l_1, l_2, \ldots, l_n \rangle$) = l_n

constrói - constrói uma sucessão unitária, i.e., make(l_1) = cria(l_1) = $\langle l_1 \rangle$

final - constrói uma sucessão removendo o elemento inicial da sucessão, i.e.,tail ($\langle l_1, l_2, \ldots, l_n \rangle$) = final ($\langle l_1, l_2, \ldots, l_n \rangle$) = $\langle l_2, \ldots, l_n \rangle$

comprimento - calcula o comprimento de uma sequência

REFERÊNCIAS

1. Veloso, P.A.S., Veloso, S.R.M., Problem Decomposition and Reduction: Applicability, Soundness, Completeness: Trappl, R., Klir, J., Pichler, F. (eds); Progress in Cybernetics and Systems Research Vol. VIII (Proc. of 5th EMCSR, Vienna, 1980): Hemisphere Publ. Co. 1980.
2. Jackson, M.A. Principles of Program Design London: Academic Press, 1975.
3.Hoare, C.A.R., Notes on Data Structuring In

Dahl, O., J., Dijkstra, E.W., Hoare, C.A.R., Structured Programming. Academic Press, 1972.
4. Yourdon, E., Constantine, L.L. Structure Design: Fundamentals of a Discipline of Computer Program and System Design. Yourdon Press, 1978.
5. Von Staa, A., Engenharia de Programas, Livros Técnicos e Cientificos Editora, Rio de Janeiro, 1983.
6. Knuth, D.E., Structured Programming with go to statement; Computing Survey (614) ACM, New York, pg 261-302, Dec. 1974.
7. Veloso, P.A.S.; Martins, R.C.B.; On Reducibilities Among General Problems - Seventh European Meeting on Cybernetics and Systems Research - Viena, Austria, 1984.

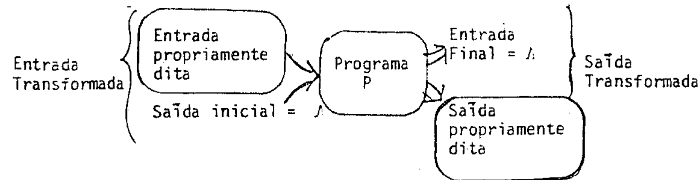

Figura 6.1

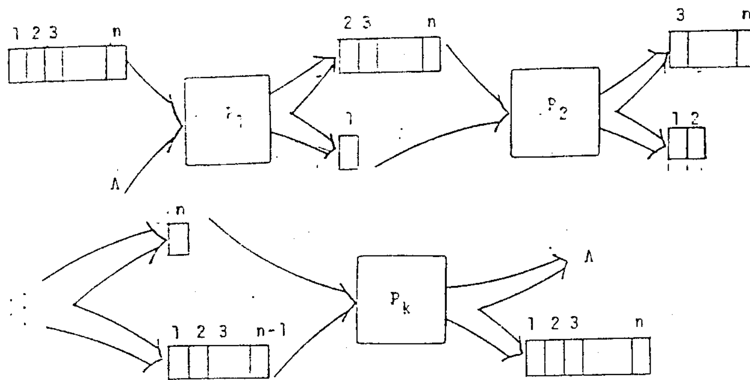

Figura 6.2