VOLUME 3
NÚMERO 1
ANO 1983/1984

REVISTA BRASILEIRA DE COMPUTAÇÃO

Publicado com o apoio do CNPq e FINEP.

RBC, Rio de Janeiro, V. 3, nº 1, 1983/1984, pp. 5 a 18

CONSTRUÇÃO DE PROGRAMAS POR TRANSFORMADORES DE DADOS: INTRODUÇÃO E ESTUDO DE CASO.

C.J.P. Lucena
R.C.B. Martins
P.A.S. Veloso

PUC/RJ
Departamento de Informática
Rua Marquês de São Vicente, 255
22453 – Rio de Janeiro, RJ

SUMÁRIO

Este trabalho introduz um novo método de programação chamado o método dos transformadores de dados. Em particular, apresenta-se uma especialização do método orientada para problemas do tipo processamento de arquivos sequenciais. O método é exemplificado através de um exemplo que não é solúvel pelo método de Jackson. Ilustra-se como o método reduz o problema original a um conjunto de sub-problemas que podem ser resolvidos através de uma aplicação direta do método de Jackson. Produz-se uma solução que é correta por construção.

ABSTRACT

This paper introduces a new method for program construction, namely the method of data transformers. In particular, its specialization oriented towards problems concerning processing of sequential files is presented. The method is presented by means of an example that is not solvable by Jackson's method. The method reduces the original problem to a set of subproblems that can then be solved by a direct application of Jackson's method, thereby production a solution that is correct by construction.

Pontifícia Universidade Católica do Rio de Janeiro - Departamento de Informática, Rua Marquês de São Vicente, 225 - Gávea - RJ - 22453
Trabalho parcialmente financiado pela FINEP e CNPq.

1. INTRODUÇÃO

Como em muitas áreas da engenharia também na área de engenharia de software, grande parte dos resultados da pesquisa básica (neste caso pesquisa em teoria da programação) vem levando um tempo muito longo para serem transferidos para a indústria. Em realidade, a maioria do trabalho em derivação formal de programas tem tido pouco ou nenhum impacto nas aplicações rotineiras de processamento de dados que se desenvolvem na indústria de computação. Por outro lado, como a programação de problemas do gênero processamento de arquivos sequenciais não tem sido estudada do ponto de vista formal, profissionais experimentados da área de programação não dispõem de ferramentas para expressar suas idéias sobre metodologias de programação de maneira rigorosa.

Mesmo as propostas bem sucedidas de Jackson [3], Warnier [5] e Yourdon e Constantine [6] só puderam ser tornadas precisas através de exemplificação exaustiva. Frequentemente, aspectos sutis dessas metodologias não puderam ser expressos com o nível de precisão que é encontrado, por exemplo, na maior parte da literatura sobre construção de programas. O método dos transformadores de dados não só lida com a classe de problemas que pode ser resolvida pelo método básico de Jackson [3] como também pode resolver de uma maneira uniforme, problemas somente resolvíveis por Jackson através de grandes adaptações do seu método básico.

Cowan e Lucena [1] ao introduzirem um novo fator (níveis abstratos de especificação para dados e programas e subsequente implementação em termos de níveis mais concretos de abstração) no método de Jackson, resolveram o problema de ordenação para ilustrar como o exercício do raciocínio abstrato sobre um problema pode conduzir a soluções novas ou mesmo a soluções julgadas inviáveis devido a limitações de um determinado método.

Tratou-se de investigar a questão de até que ponto o enfoque original proposto por Cowan e Lucena [1] poderia ser generalizado e formalizado como um método.

O método dos transformadores de dados envolve a aplicação de transformações aos dados do problema ao nível de sua formulação inicial. Ele utiliza as noções formais de redução e decomposição de problemas [4]. As transformações de dados são expressas em termos de construtores de tipos de dados propostos por Hoare [2]. O método reduz o problema original a um conjunto de sub-problemas que po

dem ser resolvidos através da aplicação direto do método de Jackson. Produz-se assim uma solução que é correta por construção.

O presente trabalho formula o método dos transformadores de dados ao mesmo tempo que o aplica ao problema de "ordenação" (não resolvível pelo método básico de Jackson).

2. APLICAÇÃO DO MÉTODO DE TRANSFORMADORES DE DADOS AO PROBLEMA DE "ORDENAÇÃO"

Selecionamos o problema de "ordenação" para exemplificar a aplicação do método por uma série de razões. Em primeiro lugar, o problema é muito conhecido e portanto o leitor pode concentrar sua atenção no método de solução do problema, compará-lo com outras soluções possíveis, sem perder tempo com complexidades intrínsecas do problema. Em segundo lugar, como a "ordenação" exemplifica uma situação de "backtracking" (ou, pelo menos, algum "backtracking") ele ilustra um caso no qual o método básico de Jackson não pode ser diretamente aplicado.

Também, tiraremos proveito da concisão da formulação do problema de "ordenação" para ilustrar a teoria do método dos transformadores de dados através do seu desenvolvimento. Seria mais difícil fazer o mesmo com um problema cuja definição fosse mais longa ou complexa.

Seja A um conjunto totalmente ordenado, $d=\langle a_1,a_2,\ldots,a_n\rangle \in D$ uma sequência finita de elementos de A e $o=\langle b_1,b_2,\ldots,b_n\rangle \in O$ uma sequência finita de elementos de A. Fazer uma "ordenação" significa resolver um problema $P = \langle D,O,SORT\rangle$ tal que SORT(o,d) é a relação binária em DxO definida por

$$(i)\ \{a_1,\ldots,a_n\} = \{b_1,\ldots,b_n\}$$
$$(ii)\ (\forall i, j)\ 1\leq i<j\leq n \Rightarrow b_i < b_j$$

Com o propósito de simplificação vamos supor que $a_i \neq a_j$ para todo $i \neq j$ e $d \neq \Lambda$, onde Λ denota uma sequência vazia.

Graficamente, o que apresentamos até agora, pode ser expresso na figura 1.

Na figura 1, a relação SORT do problema P é uma relação binária entre D e O e o programa p é uma função total entre D e O que implementa SORT e tal que

$$(\forall d:D)\ SORT\ (p(d),d)$$

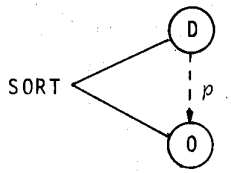

Figura 1

A partir deste ponto estamos prontos para aplicar o primeiro passo do método que consiste em reduzir o problema original P = <D,O,SORT> ao problema P_1 = <DxO,DxO,$SORT_1$>. Observe-se que associamos, a nível de formulação do problema, os dados de entrada aos dados de saída, tomando DxO, o produto cartesiano de D e O. A figura 2 ilustra o conceito de redução (representada pelo símbolo Γ) do problema P ao problema P_1.

Numa linguagem do gênero Pascal, um procedimento p que resolveria o problema P original, poderia ser expresso da seguinte maneira:

```
Procedure p;
   var x1,y1:DxO;
      begin
           x1.i ← x;
           x1.r ← Λ;
           p1;
           y ← y1.r
      end {p};
```

Note que os seletores i e r simulam as funções *ins* e *ret* que constroem um par ordenado (*ins* de *insere*) e obtêm o segundo elemento de um par ordenado (*ret* de *retira*), respectivamente, Para que o procedimento p resolva o problema original, após esta primeira transformação nos dados do problema, é necessário que x seja uma *cópia* de d (entradas originais) e o (saída original) seja uma *cópia* de y.

```
x ← copia (d);
o ← copia (y)
```

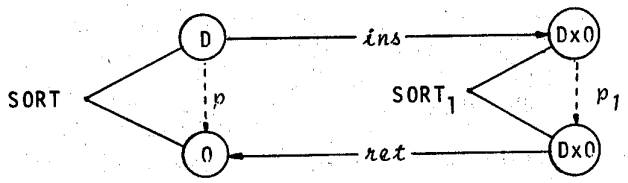

Figura 2: $P \downarrow P_1$

Aplicamos agora o segundo passo do método que consiste em uma segunda redução do problema para tratar com uma nova forma transformada dos dados. Trabalharemos agora com (DxO)* que é o espaço das sequências de pares entrada-saída. A figura 3 representa graficamente esta nova redução.

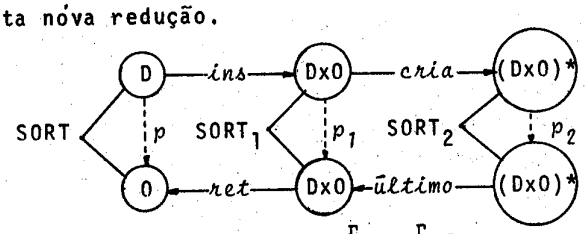

Figura 3: $P \downarrow P_1 \downarrow P_2$

O procedimento p_1 pode agora ser escrito da seguinte forma:

```
Procedure p1;
   var x2, y2 : (DxO)*
      begin
           x2 ← cria(x1);
           p2;
           y1 ← último(y2)
      end {p1};
```

As funções *cria* e *último* teriam de ser codificadas segundo as suas definições usuais para arquivos, isto é, *cria* produz uma sequência de comprimento unitário e *último* extrai o último elemento de uma sequência. Note-se que até este ponto, tudo que foi feito destinou-se a rever a formulação do problema de forma a colocá-lo em uma forma canônica. A revisão da formulação foi sendo feita a partir de transformações nos dados que o problema deve manipular. Mais adiante apresentaremos uma motivação informal para a reorganização proposta para o problema.

O passo seguinte do método prevê uma primeira decomposição de P_2 (representada por $P_2\uparrow$). A figura 4 representa graficamente esta

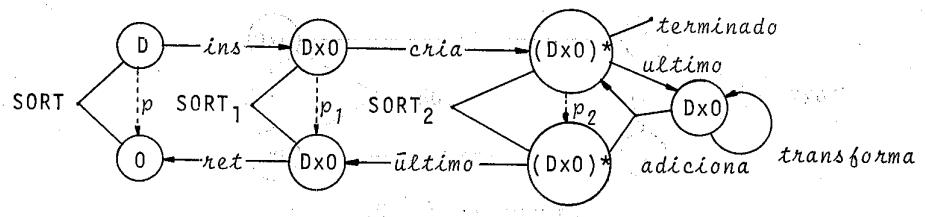

Figura 4: $P \overset{\Gamma}{\downarrow} P_1 \overset{\Gamma}{\downarrow} P_2 \uparrow (último, transforma, adiciona)^*$

Podemos agora expressar o programa p_2, que utiliza a função *atualização* apresentada a seguir.

```
Procedure p2;
   var x3: (DxO)*;
      begin
          x3 ← x2;
          while ¬ terminado(x3) do
                x3 ← atualização(x3);
          y2 ← x3;
      end {p2};
Procedure atualização(x3:(DxO)*):(DxO)*;
    var x4: DxO;
        y3: (DxO)*;
       begin
            y3 ← x3;
            x4 ← último(x3);
            x4 ← transforma(x4);
            atualização ← adiciona(y3,x4)
       end {atualização};
```

Para o próximo passo, procederemos a uma nova decomposição, separando os elementos do par ordenado (a sequência de entrada da sequência de saída) além de retirarmos um elemento da sequência de entrada, processando-o e colocando na saída. Esta idéia pode ser expressa graficamente através do diagrama da figura 5.

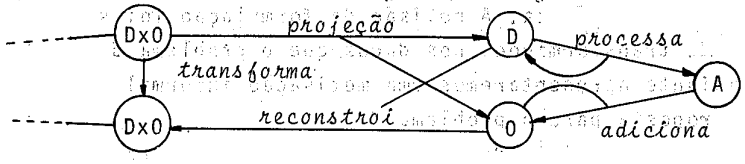

Figura 5

Pode-se visualizar este passo de decomposição, como sendo um desdobramento da Figura 4 que ilustra a decomposição do programa transforma. A função projeção obtêm o primeiro e segundo componentes do par ordenado (é simulada pelos seletores *i* e *r* no *transforma* a ser apresentado a seguir). A função *reconstrói* monta um par ordenado a partir de dois elementos dados.

É necessário, agora, definir processa de tal forma que toda vez que ele seja executado ele reduza a entrada e amplie a saída, contribuindo para a solução do problema.

Como o problema "ordenação" é muito bem conhecido, é simples identificar a operação central de processa, sem a necessidade de decomposições adicionais. A operação central de processa neste caso será a seleção do elemento *mínimo* da entrada e sua adição ao fim da sequência de saída. A operação determina uma sequência de buscas, cada uma envolvendo uma única passagem sobre a entrada. Podemos neste ponto propor um código para transforma e processa.

```
Procedure transforma(x4:DxO):DxO;
    var x5,x6:D;
        y5,y6:O;
        minimo: objeto pertencente a A;
      begin
          x5 ← x4.i;
          y5 ← x4.r;
          processa;
          y6 ← adiciona(y5,minimo);
          transforma ← reconstroi(x6,y6)
      end {transforma};

Procedure processa;
      begin
          minimo ← primeiro(x5);
          x5 ← final(x5);
          x6 ← Λ;
          while   x5 ≠ Λ do
            if minimo < primeiro(x5)
               then begin
                         x6 ← adiciona(x6,primeiro(x5));
                         x5 ← final(x5)
                    end
```

```
            else begin
                    x6 ← adiciona(x6,minimo);
                    minimo ← primeiro(x5),
                    x5 ← final(x5)
                 end
    end {processa};
```

As funções *primeiro* e *final* têm seus significados usuais quando aplicadas as sequências, isto é, "head" e "tail", respectivamente.

Devemos agora especificar o predicado terminado de tal forma a satisfazer as condições de correção do programa. Para tal notamos que processa reduz a cada passagem o comprimento da primeira componente do par ordenado que está sendo transformado. Isto naturalmente sugere que este processo termina quando o comprimento da primeira componente for zero. Podemos definir como:

$\forall x_3 \in (DxO)^*$, $terminado(x_3) \Leftrightarrow comprimento(ultimo(x_3).i)=0$

Da maneira pela qual processo é construído $comprimento(transforma(x_3).i) < comprimento(x_3.i)$ o que garante a terminação do programa. A prova da correção pode ser feita por indução sobre a maneira pela qual a sequência de saída é construída (em cada passo introduzimos o menor elemento possível).

No apêndice apresentamos uma versão completa da solução expressa numa linguagem do gênero Pascal.

3. 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 tipo processamento de arquivos sequenciais, além de introduzir vantagens adicionais tais como: fácil determinação da terminação de programas (condição 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. Requer-se 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 ilustra o que acabamos de escrever.

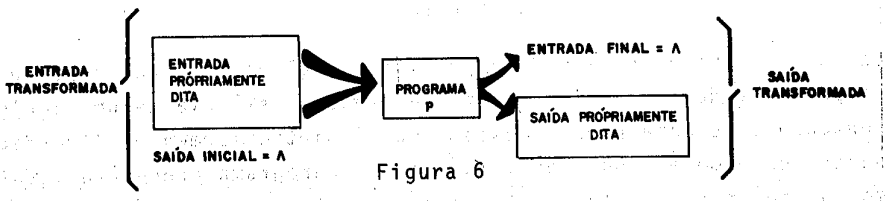

Figura 6

O metodo 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 um 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 7 ilustra a nova formulação do problema.

O método dos transformadores de dados reduziu o problema à 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 vi

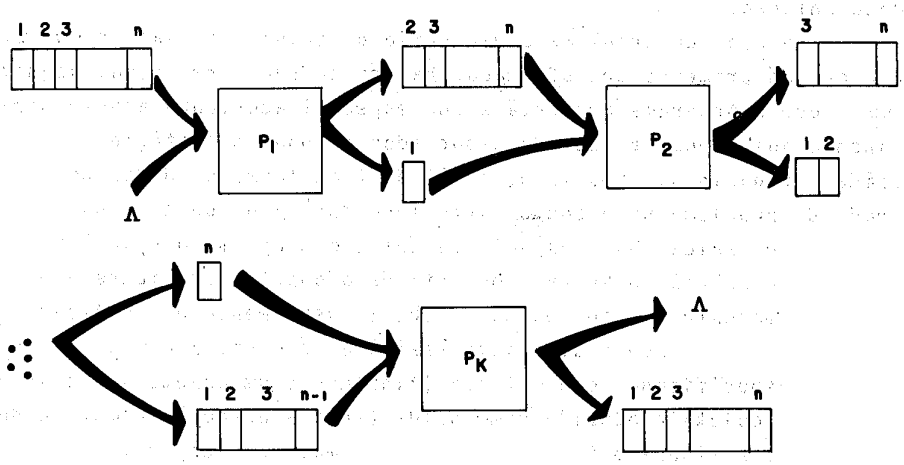

Figura 7

da, a existência de um modelo preciso e legível tal qual 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. Estamos no momento trabalhando num procedimento que, através de refinamentos (transformações), nos permita passar do modelo produzido pelo método para uma implementação eficiente do mesmo. O método deverá futuramente ser apoiado por um software especial de apoio ao desenvolvimento de programas.

APÊNDICE

```
Program sort;
   type D = seq de objetos pertencentes a A;
        O = seq de objetos pertencentes a A;
        (DxO) = record i:D;
                       r:O
                end
        (DxO)*= seq de (DxO);
   var x,d,x5,x6:D;
       y,o,y5,y6:O;
       x1,y1:DxO;
       x2,y2:(DxO)*;
       mínimo : objeto pertencente a A;
```

```
Procedure processa;
        begin
             minimo ← primeiro(x5);
             x5 ← final(x5);
             x6 ← Λ;
                 while x5 ¬= Λ do
                      if minimo < primeiro(x5)
                      then
                          begin
                               x6 ← adiciona(x6,primeiro(x5));
                               x5 ← final(x5)
                          end
                      else
                          begin
                               x6 ← adiciona(x6,minimo);
                               minimo ← primeiro(x5);
                               x5 ← final(x5)
                          end
        end {processa};
Procedure transforma(x4:Dx0):Dx0;
        begin
             x5 ← x4.i;
             y5 ← x4.r;
             processa;
             y6 ← adiciona(y5,minimo);
             transforma ← reconstroi(x6,y6)
        end {transforma};
Procedure atualização(x3:(Dx0)*):(Dx0)*;
  var x4:Dx0;
      y3:(Dx0)*;
        begin
             y3 ← x3;
             x4 ← ultimo(x3);
             x4 ← transforma(x4);
             atualização ← adiciona(y3,x4)
        end {atualização};
```

```
Procedure p2;
  var x3:(Dx0)*
      begin
           x3 ← x2
           while comprimento(ultimo(x3).i) ≠ 0 do
                x3 ← atualização(x3);
           y2 ← x3
      end {p2};
Procedure p1;
      begin
           x2 ← cria(x1);
           p2;
           y1 ← ultimo(y2)
      end {p1};
Procedure p;
      begin
           x1.i ← x;
           x1.r ← Λ;
           p1;
           y ← y1.r
      end {p};
      begin
           x ← cópia(d);
           p;
           o ← copia(y)
      end {sort}.
```

REFERÊNCIAS

1. Cowan, D.D., Graham, J.W., Welch, J.W., Lucena, C.J. A Data-directed Approach to Program Construction, Software-Practice and Experience. Vol. 10, Waterloo, 1980.
2. Hoare, C.A.R., Notes on Data Structuring. in Dahl, O.,J.,Dijkstra, E.W., Hoare, C.A.R., Structured Programming. Academic Press: 1972.
3. Jackson, M.A. Principles of Program Design. London: Academic Press, 1975.
4. 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.

5. Warnier, J.D. Logical Construction of Programs. New York: Van Nostrand Reinhold, 1974.
6. Yourdon, E., Constantine, L.L. Structure Design: Fundamentals of a Discipline of Computer Program and System Design. Yourdon Press, 1978.