SOBRE ELIMINAÇÃO DE RECURSÃO EM PROGRAMAS

MARIA ELISABETE BRUNO VIVIAN

DISSERTAÇÃO APRESENTADA AO INSTITUTO DE MATEMÁTICA E ESTATÍSTICA DA UNIVERSIDADE DE SÃO PAULO, PARA OBTENÇÃO DO GRAU DE MESTRE EM MATEMÁTICA APLICADA.

ORIENTADOR: PROF. DR. VALDEMAR WAINGORT SETZER

SÃO PAULO, SETEMBRO DE 1979.

A minha mãe

e

ao Alceu

AGRADECIMENTOS

Ao Professor Dr. Valdemar W. Setzer,

que como orientador, mas principalmente como amigo foi o grande incentivador para a conclusão deste trabalho. Sua paciência, carinho e dedicação constantes excederam em muito o que seria de se esperar de uma orientação;

aos meus amigos do MAP,

pelo estímulo, pelas eventuais discussões a respeito do nosso tema e pelo parcial desencargo de minhas tarefas usuais;

ao Conselho Nacional de Desenvolvimento Científico e Tecnológico (CNPq) e à IBM pelo auxílio que me concederam;

à Luzia do Carmo Namiki e Regina Helena da Silva pelo excelente serviço de datilografia.

São Paulo, setembro de 1979

M.E.B.V.

Í N D I C E

Pag.

PREFÁCIO 1

CAPÍTULO I - INTRODUÇÃO 3

1. Linguagem 3
2. Definições básicas e nomenclatura. 6
3. Classificação de esquemas recursivos 10

CAPÍTULO II - HISTÓRICO 13

CAPÍTULO III - CATÁLOGO DE REGRAS 37

CAPÍTULO IV - APLICAÇÃO DE MÉTODOS DE PROVA 67

1. Definições 67
2. Método da indução recursiva 68
3. Método de indução sobre um conjunto bem fundado 74
4. Método da indução computacional... 77
5. Método da asserção intermitente... 86
6. Aritmetização do controle 94

BIBLIOGRAFIA 105

PREFÁCIO

> I have always felt that the transformation from recursion to iteration is one of the most fundamental concepts of computer science.
>
> D.E.Knuth

Em tempos recentes a programação de computadores sofreu uma transformação muito grande quando se mostrou que poderia haver uma técnica para o desenvolvimento de programas. Até a bem pouco tempo essa técnica era aplicada de maneira sistemática mas não formal. Mais recentemente apareceram trabalhos que mostraram a possibilidade do processo de programação seguir um roteiro semelhante ao das deduções matemáticas (ou da lógica), utilizando-se processos formais. Isso é conseguido através da aplicação de regras de transformação. Se tais regras forem previamente demonstradas como sendo corretas, pode-se desenvolver programas corretos desde que se parta de formulações iniciais também corretas. Nesse sentido, temos uma nova técnica, oposta a já tradicional técnica de prova de "corretude" de programas, iniciada por Floyd /15/ e que conta com vários desenvolvimentos teóricos e práticos.

Nosso trabalho versará sobre um dos tópicos de transformações formais de programa, qual seja, o de eliminação da recursão. Esse tópico é importante pois praticamente sempre é mais eficiente processar-se um programa iterativo do que um recursivo. No entanto é em geral muito mais simples desenvolver um programa inicial recursivo do que um iterativo,principalmente em problemas com formulações indutivas.

Devido às limitações deste trabalho, deixamos de tecer comentários a respeito da eficiência das diferentes versões de cada procedimento. Essa análise pode ser feita teoricamente, mas na prática depende de fatores por vezes de difícil acesso ao programador, como por exemplo, características particulares dos computadores e da implementação de procedimentos recursivos. De qualquer maneira a eliminação de recursão é útil para a utilização de rotinas em linguagens que não permitem a declaraçao de procedimentos recursivos. A formulação recursiva permite, em geral, uma abordagem em nível de abstração mais alto do que a iterativa, possibilitando a obtenção de uma visão conceitual do problema.

O trabalho consta de quatro capítulos cujos conteúdos relatamos a seguir. No Capítulo I introduzimos os conceitos e notações básicos. No Capítulo II traçamos um histórico do desenvolvimento desta área da computação. Este capítulo foi resultado da pesquisa bibliográfia que realizamos. No Capítulo III, elaboramos um catálogo de regras de eliminação obtidas através de generalizações de regras que surgiram na bibliografia. No Capítulo IV apresentamos a prova de algumas regras do catálogo para exemplificar métodos de provas que estabelecem a validade destas regras.

Os capítulos são denotados por números romanos. As secções dentro de cada capítulo e as definições, lemas e teoremas dentro de cada secção são numerados sequencialmente a partir de 1. Ao nos referirmos a um ítem, como por exemplo ítem 2.5 do capítulo I nos referimos por 2.5 no capítulo I e por I.2.5 nos demais capítulos.

CAPÍTULO I

INTRODUÇÃO

1. LINGUAGEM

Usaremos a linguagem ALGOL-68 /20/, para descrever os procedimentos que ocorrem neste trabalho. A ela juntaremos alguns recursos adicionais como o comando de atribuição múltipla, o tipo stack e funções definidas sobre o tipo stack. Em pontos de alguns procedimentos inserimos, em forma de comentário, um número entre chaves para nos referirmos àqueles pontos do procedimento.

1.1 Definição do comando de atribuição múltipla:

O comando de atribuição múltipla terá a seguinte sintaxe:

$$(<var_1>,<var_2>,\ldots,<var_n>):=(<expr_1>,<expr_2>,\ldots,<expr_n>)$$

Este comando é equivalente a:

$$t_1:=<expr_1>;$$
$$t_2:=<expr_2>;$$
$$\vdots$$
$$t_n:=<expr_n>;$$

$$<var_1> := t_1;$$
$$<var_2> := t_2;$$
$$\vdots$$
$$<var_n> := t_n$$

onde para $1 \leq i \leq n$, t_i não ocorre em $<expr_j>$ e é diferente de $<var_j>$ para $1 \leq j \leq n$.

1.2 Definição do tipo stack

Em alguns esquemas de procedimento aparecerão declarações de variáveis do tipo ("mode") stack. Quando fizermos uma transformação num programa, usando uma regra que envolve um desses procedimentos, deveremos acrescentar às declarações das variáveis globais do programa a criação deste novo tipo juntamente com os procedimentos definidos abaixo, que operam sobre o tipo stack.

Para criarmos o tipo stack fazemos:

```
mode stack = struct(ref stack next, t info)
```

O tipo t que aparece na definição de stack deve ser em cada programa, definido de forma compatível com os valores que desejamos empilhar.

Estamos supondo que não haja no programa a ser transformado nenhum identificador com os nomes: stack, push, pop , top, remove e empty.

1.2.1 - Procedimento push

Este procedimento empilha o valor de uma variável x na pilha.

```
proc push = (ref stack s, t x) void:
  begin ref stack s1 = heap stack;
    info of s1:=x; next of s1:=s;
    s:=s1
  end
```

1.2.2 - Procedimento pop

Este procedimento copia o valor do topo da pilha na variável x, desempilhando este valor.

```
proc pop = (ref stack s, ref t x) void:
  begin
    x:=info of s;
    s:=next of s
  end
```

1.2.3 - Procedimento top

Este procedimento tipo função devolve o valor do topo da pilha.

```
proc top = (stack s) t:
  info of s
```

1.2.4 - Procedimento remove

Este procedimento desempilha o valor do topo da pilha.

```
proc remove = (ref stack s) void:
  s:=next of s
```

1.2.5 - Procedimento empty

Este procedimento devolve o valor false se há algum elemento na pilha e true caso contrário.

```
proc empty(stack s) bool:
  s:=:nil
```

2. DEFINIÇÕES BÁSICAS E NOMENCLATURA

2.1 - Uma função é um procedimento que devolve um valor e não tem efeitos colaterais.

2.2 - Um procedimento do tipo função é um procedimento que devolve um valor.

2.3 - Dada uma linguagem de programação L um esquema de procedimento em L é um procedimento em que ocorrem tipos de parâmetros e de variáveis locais, funções, operadores, predicados, ou alguma sequência de comandos não especificamente em L mas sim, em forma de meta-símbolos que indicam família de símbolos de L. Dessa maneira um esquema de procedimento representa uma família de procedimentos em L. Um programa da família é obtido fornecendo-se uma interpretação (cf.2.4) para cada meta-símbolo do esquema na linguagem L. Muitas vezes, no texto que se segue, empregamos o termo esquema ou as expressões esquema de função e esquema de procedimento do tipo função.

Os seguintes meta-símbolos, com ou sem índice, serão usados para expressar ítens sintáticos nos esquemas de um procedimento:

t	- tipos ("modes") primitivos ou compostos
p,q	- constantes de predicados totais
a,b,c,d,e	- constantes de funções totais
R	- sequência de comandos ("serial-clause") que não altera o valor dos parâmetros do procedimento e que tem eventualmente variáveis locais.

x,y - parâmetros ou variáveis locais; podem ser entendidos como uma enupla de valores.

i,j,k,m,n - identificadores de contadores inteiros

z - variável local

s - identificador de pilha

L - rótulo

2.4 - Uma interpretação I de um esquema de procedimento F representada por F_I, consiste de:

a) um domínio D

b) para cada meta-símbolo de constante de predicado p um predicado total

$$p_I : D \to \{true, false\}$$

c) para cada meta-símbolo de constante de função n-área h ∈ {a,b,c,d,e,...} uma função total

$$h_I : D^n \to D$$

d) para cada meta-símbolo de tipo t um tipo válido na linguagem de programação L.

e) para cada sequência de comando R uma sequência de comandos válida na linguagem de programação L (que não altera o valor do parâmetro).

2.5 - Um esquema de procedimento que define um procedimento f é dito recursivo se no corpo do procedimento ocorre pelo menos uma chamada de f. Nesse trabalho todos os procedimentos recursivos serão da forma:

```
proc f = (t1 x) t2:
  begin
    α1
(1) if p(x) then α2 f(a(x)) α3
             else α4 fi
    α5
  end
```

onde α_i, para $1 \leq i \leq 5$, é uma sequência de símbolos do esquema e a ocorrência de qualquer seleção em α_2 é da forma if...fi; o mesmo acontecendo com α_3. Devido a necessidade da parada da recursão e a possibilidade de transformações de esquemas levando à forma (1), esta não limita a generalidade dos esquemas de interesse. O tipo $\underline{t}_1$ (eventualmente uma composição de tipos) corresponderá sempre a chamada por valor.

Note-se que estamos excluindo recursão indireta, isto é, aquela em que existe uma cadeia de procedimentos $f=f_1, f_2, \ldots, f_{n-1}, f_n=f$ onde f_i chama f_{i+1}, para $1 \leq i < n$, pois não ocorrem neste trabalho. Esta restrição não é muito limitante, já que a maioria dos esquemas recursivos indiretos pode ser transformada através da técnica de expansão (cf.2.12) em esquemas recursivos diretos.

2.6 - Um esquema de procedimento será chamado iterativo se ele não é recursivo e não contém como variável local nenhuma variável do tipo stack.

2.7 - Um esquema de procedimento é denominado iterativo com pilha se ele não é recursivo e se contém como variável local alguma variável do tipo stack.

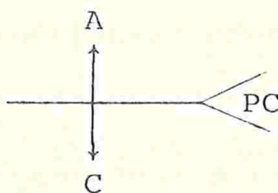
Observe-se que um esquema pode ser iterativo e no entanto ter uma particular interpretação que leva a um procedimento recursivo. Por exemplo o esquema iterativo:

proc f = ($\underline{t}_1$ x) $\underline{t}_2$:

while p(x) do R od

com uma determinada interpretação de R contendo uma chamada de f.

2.8 - Uma regra de eliminação de recursão, ou simplesmente regra é um tripla <A,C,PC> onde A é um esquema de procedimento recursivo, C é um esquema de procedimento iterativo ou iterativo com pilha e PC é um conjunto de pré-condições sobre as funções e predicados que ocorrem em A e C. Uma regra será representada na seguinte forma:



O esquema recursivo A é denominado de antecedente, enquanto que o esquema iterativo B é denominado consequente da regra.

Nas pré-condições PC omitiremos os quantificadores para que a notação não fique por demais carregada.

2.9 - Um instantâneo é a descrição das informações associadas a cada ponto de um programa, a saber: valor corrente de cada variável ativa, valor corrente de todos os valores empilhados e do topo de cada variável ativa do tipo stack, a sequência de todos os registros a serem lidos num arquivo sequencial de entrada e a sequência de todos os registros que foram "gravados" num arquivo sequencial de saída e, finalmente, a sequência de todos os registros e do apontador de um arquivo de acesso aleatório.

2.10 - Um procedimento f é equivalente a um procedimento g se chamadas de f e g com os mesmos parâmetros, ocorrendo no mesmo instantâneo S de um programa P, retornam produzindo em seu retorno um mesmo valor E e um mesmo instantâneo S' de P ou ambas não retornam. Portanto podemos substituir em P a declaração e as chamadas de f pela declaração e chamadas de g.

2.11 - Uma regra de eliminação <A,C,PC> é consistente se para qualquer interpretação I que satisfaça PC, A_I é equivalente a C_I. Se conseguimos provar que uma regra de eliminação é consistente dizemos que foi provada a consistência da regra.

2.12 - Expansão ("unfolding", /10/) é a substituição da chamada de um procedimento pelo seu corpo, trocando-se formalmente os parâmetros formais pelos parâmetros atuais.

2.13 - Contração ("folding", /10/) é a troca de uma sequência de comandos pela chamada de um procedimento cujo corpo, substituindo-se os parâmetros formais pelos parâmetros atuais dessa

chamada resulta numa sequência idêntica à sequência original.

Podemos esquematicamente representar a contração e a expansão da seguinte maneira /5/:

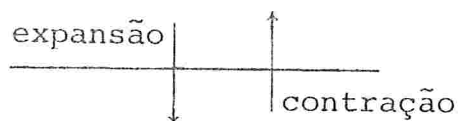
<u>proc</u> f = ($\underline{t}_1$ x) $\underline{t}_2$:

<u>begin</u>

R

<u>end</u>

...f(x_o)...



<u>proc</u> f = ($\underline{t}_1$ x) $\underline{t}_2$:

<u>begin</u>

R

<u>end</u>

...$R^x_{x_o}$...

onde $R^x_{x_o}$ derrota a sequência de comandos em que toda ocorrência de x foi trocada formalmente por x_o.

3. <u>CLASSIFICAÇÃO DE ESQUEMAS RECURSIVOS</u>

3.1 - A chamada de f no esquema (1) de 2.5 é dita <u>iterativa</u>/23/ se ela é executada imediatamente antes do ponto de retorno da função, isto é, se o esquema (1) é equivalente a:

```
proc f = (t1 x) t2:
  begin
    α1
    if p(x) then α2 f(a(x)); go to L α3
            else α4
    α5
L:end
```

3.2 - A chamada de f no esquema (1) de 2.5 é dita <u>linear</u> /30/se em α_2 e α_3 não existe nenhuma outra chamada de f.

3.3 - A chamada de f no esquema (1) de 2.5 é denominada, <u>encai-xada</u> ("nested") /32/ se a sequência de símbolos f(a(x)) de (1), ocorre como parâmetro de uma chamada de f em α_2.

3.4 - A chamada de f no esquema (1) de 2.5 é denominada <u>em cas cata</u> /34/ se existe alguma chamada de f em α_2 ou α_3 que não seja encaixada.

3.5 - Um esquema recursivo de f é <u>iterativo</u>, <u>linear</u>, <u>encaixado</u> ou <u>em cascata</u> se toda chamada de f for respectivamente iterati va, linear, encaixada ou em cascata.

CAPÍTULO II

HISTÓRICO

Em 1962, J.McCarthy em /23/ introduziu o conceito de recursão iterativa (cf.I.3.1), afirmando que a eliminação da recursão de uma função só é imediata se a recursão é iterativa.

No ano seguinte, McCarthy publicou um artigo /24/ no qual foi proposto um formalismo baseado em recursão e expressões condicionais do tipo

$$\underline{\text{if}}\ p_1\ \underline{\text{then}}\ e_1\ \underline{\text{else}}\ \underline{\text{if}}\ p_2\ \underline{\text{then}}\ e_2\ \ldots\ \underline{\text{else}}\ \underline{\text{if}}\ p_n\ \underline{\text{then}}\ e_n$$

para aplicação na então emergente teoria da computação. Uma de suas contribuições mais importantes ao nosso assunto foi a introdução de uma nova técnica de prova conhecida como <u>indução recursiva</u> ("recursive induction") que pode ser usada para estabelecer a equivalência de funções definidas recursivamente. Essa técnica pode ser assim descrita: suponha que desejemos provar que duas funções $g(x)$ e $h(x)$, definidas recursivamente, são equivalentes; o método de indução recursiva consiste em determinar uma equação funcional:

$$(1) \qquad f(x) = E(x,f)$$

da qual podemos provar:

a) que define uma função f

b) que é satisfeita por h e g.

Esse método foi usado por D.C.Cooper em 1966, /11/ na prova da consistência (cf.I.2.11) de regras de eliminação de recursão (cf.I.2.8). Observe-se, portanto, que Cooper foi o primeiro a introduzir regras de eliminação, da maneira como entendemos neste trabalho. Para obtê-las, o autor generalizou tres funções que calculam o fatorial que foram examinadas por McCarthy em /22/. Para que, no caso geral, os esquemas obtidos permanecessem equivalentes, foi necessário formular algumas pré-condições (cf.I.2.8) sobre os esquemas. No estabelecimento dessas pré-condições, Cooper baseou-se em duas idéias: a idéia da função comutativa à esquerda e a idéia da função inversa. A primeira das regras, transcrita abaixo, é fundamentada na primeira idéia:

```
     proc f = (t1 x) t2:
(2)    if x≠k then a(x,f(b(x)))
              else d fi

         a(u,k) = e(u,k)                      (3 )
         a(u,e(v,w)) = e(v,a(u,w))            (3')

     proc f = (t1 y) t2:
       begin t1 x:=y,t2 z:=d;
(4)      while x≠k do (x,z):=(b(x),e(x,z)) od;
         z
       end
```

A idéia foi extraída de uma análise das expressões computadas nas chamadas de f em (2) e (4) para um dado x_o. No que se segue, vamos empregar algumas vezes a forma abreviada $g^i x$ cuja definição recursiva é:

$$g^i x = \begin{cases} x & \text{se } i=0 \\ g(g^{i-1}(x)) & \text{se } i>0 \end{cases}$$

Se o esquema (2) termina para x_o isto é, se existe $n \in \mathbb{N}$ tal que $b^n x_o = k \land \forall_i [0 \le i < n \land b^i x_o \ne k]$ então o valor computado por f em (2) é:

$$(5) \qquad a(x_o, a(bx_o, \ldots, a(b^{n-2}x_o, a(b^{n-1}x_o, d))\ldots))$$

Em (4) para o mesmo x_o, temos:

$$(6) \qquad e(b^{n-1}x_o, e(b^{n-2}x_o, \ldots, e(bx_o, e(x_o, d))\ldots))$$

Podemos verificar por indução que as expressões (5) e (6) computam o mesmo valor, quando se assume (3) e (3') (para efeito da prova chamemos a função f definida em (4) de $\bar{f}$):

i) Para n=0 temos $f(x_o) = d = \bar{f}(x_o)$

ii) Para n=1 temos $f(x_o) = a(x_o, d) = e(x_o, d) = \bar{f}(x_o)$ pela pré-condição (3).

iii) Para n>1 temos:

$$f(x_o) = a(x_o, a(bx_o, \ldots, a(b^{n-2}x_o, a(b^{n-1}x_o, d))\ldots))$$

$$= a(x_o, a(bx_o, \ldots, a(b^{n-2}x_o, e(b^{n-1}x_o, d))\ldots))$$

por (3)

$$= e(b^{n-1}x_o, a(x_o, a(bx_o, \ldots, a(b^{n-2}x_o, d)\ldots)))$$

por n aplicações da pré-condições (3')

$$= e(b^{n-1}x_o, e(b^{n-2}x_o, \ldots, e(bx_o, e(x_o, d))\ldots))$$

por hipótese de indução

$$= f(x_o)$$

Da idéia da função inversa surgiu a regra abaixo, cujo antecedente (cf.I.2.8) é o mesmo da regra anterior:

```
proc f = (t1 x) t2:
  if x≠k then a(x,f(b(x)))
          else d fi
```

$$u=b_1(b(u))=b(b_1(u)) \quad (7)$$

```
proc f = (t1 x) t2:
  begin t1 y:=k,t2 z:=d;
(8)  while x≠y do (y,z):=(b1(y),a(b1(y),z)) od;
     z
  end
```

Para o referido x_o, isto é, para x_o tal que $b^n x_o=k$ é verdadeiro e $b^i(x_o)=k$ é falso para $0\leq i\leq n$ o valor calculado no esquema (8) é:

$$a(b_1^n k, a(b_1^{n-1}k, \ldots, a(b_1^2 k, a(b_1 k, d))\ldots)) \quad (9)$$

uma vez que se $b^n x_o=k$ então $b_1^n b^n x_o=x_o=b_1^n k$, por (7). É fácil ver que as expressões (5) e (9) computam o mesmo valor observando-se que $b_1^i k=b^{n-i}x_o$ para $0\leq i\leq n$.

Desenvolvemos as regras III.8 e III.6 baseando-nos nessas duas regras, respectivamente.

No capítulo IV, aplicamos o método de McCarthy segundo o uso feito por Cooper, para provar a consistência da regra III.8.

O método de McCarthy foi retomado em 1971 por J.C. Morris /25/, que o generalizou introduzindo um novo princípio de indução recursiva. Para poder empregar esse princípio em funções parciais, Morris acrescenta ao domínio da função um objeto indefinido ω, de tal forma que toda função passa a ser total sobre o novo domínio. Assim, $f(x)=\omega$ se e somente se $x=\omega$

ou o valor de f(x) é indefinido. De forma análoga, todas as operações aritméticas e lógicas, os operadores de relação e o comando <u>if</u> são estendidos. Com isso, pode-se estabelecer o conceito de $f \subset g$, isto é, f é <u>estendida</u> por g, da seguinte maneira: $f(x) \neq \omega \Rightarrow f(x)=g(x)$.

O método consiste em fazer uma prova por indução usando uma sequência infinita de funções $\{f_i\}$. Definimos cada f_i como a f original, trocando as ocorrências das chamadas recursivas de f por uma chamada da função f_{i-1}. Para completar a sequência, precisamos dizer que f_o devolve o valor ω para qualquer chamada. Morris usando esse princípio prova a consistência da regra:

```
         proc f = (t1 x) t2:
(10)       if p(x) then a(f(b(x)))
                    else d fi
```

$$\updownarrow$$

```
         proc f = (t1 y) t2:
           begin t1 x:=y, t2 z:=d;
(11)          while p(x) do (x,z):=(b(x),a(z)) od ;
              z
           end
```

Quando comparamos os esquemas (2) e (10), observamos que Cooper, sem nenhuma restrição adicional, poderia ter substituído o predicado x=k de (2) por um predicado genérico p(x). Desse modo o esquema (10) seria um caso particular de (2), onde a função a seria unária, e, portanto, não dependeria diretamente de x. Com isso, a eliminação da recursão torna-se muito mais simples. Essa regra é englobada pela regra III.4.

Em 1970, M.S.Paterson e C.E.Hewitt /28/ fizeram um

estudo para comparar duas classes de esquemas: a classe de esquemas iterativos (cf.I.2.6) e a classe de esquemas de função (cf.I.2.1), recursivos. O principal resultado do trabalho é a prova de que existe um esquema recursivo que não é equivalente a nenhum esquema iterativo. O esquema exibido é o que se segue:

```
          proc f = (t1 x) t2:
(12)        if p(x) then a(f(b(x)),f(c(x)))
                 else d fi
```

A prova é feita usando-se uma certa família de interpretações I_n (cf.I.2.4) cujo domínio é o conjunto de todas as cadeias compostas pelos símbolos de funções a,b,c,d, em que a, b,c são interpretadas como concatenações (.) assim definidas : a(u,v)=a.u.v, b(v)=b.v, c(v)=c.v e onde $p_{I_n}(x)$ é verdadeiro se o comprimento da cadeia x é n e é falso caso contrário. Os autores mostraram que pelo menos n+1 variáveis locais são necessárias para a computação num esquema iterativo de f(λ) onde λ é a cadeia vazia. Como n não é limitado não pode haver tal esquema iterativo que compute f. Os antecedentes das regras III. 22 e III.23 são generalizações desse esquema, sendo que os consequentes (cf.I.2.8) são esquemas iterativos com pilha (cf.I. 2.7).

O exemplo fornecido acima poderia sugerir que para qualquer esquema recursivo, a introdução de uma pilha somente com os valores dos parâmetros permitiria produzir um esquema iterativo com pilha equivalente. Mas Paterson e Hewitt mostraram um outro esquema que exige também que se empilhe uma indicação do endereço de volta e não simplesmente os valores dos parâmetros. Esse esquema é:

```
proc f = (t1 x) bool:
  if p(x) then true
       else if f(b(x)) then f(c(x))
                      else false fi
```

Os autores ainda apresentam, sem prova, uma regra de eliminação cujo antecedente é (2) modificado. Essa modificação consiste em permitir que d dependa do parâmetro x, chegando-se a:

```
        proc f = (t1 x) t2:
(13)      if p(x) then a(x,f(b(x)))
                  else d(x) fi
```

Apesar da pequena modificação, as regras de Cooper não mais podem ser aplicadas pois elas exigem uma constante na parte else, isto é, um valor conhecido a priori para qualquer x_o. No caso de (13) esse valor não é conhecido para x_o, mas pode ser calculado como $d(b^n x_o)$ onde n é um inteiro tal que $p(b^n x_o)$ é falso e $p(b^i x_o)$ é verdadeiro, $0 \leq i \leq n$. Isso levou ao esquema de Paterson e Hewitt que efetua inicialmente esse cálculo e, portanto é inerentemente ineficiente, apresentando no entanto, a vantagem de não requerer nenhuma pré-condição:

```
        proc f = (t1 y) t2:
          begin t1 x:=y, x1, x2, t2 z;
            while p(x) do x:=b(x)od;
            (z,x):=(d(x),y);
(14)        L:(x1,x2):=(b(x),y);
            if p(x) then while p(x1) do
                                    (x1,x2):=(b(x1),b(x2))
                                 od;
                         (z,x):=(a(z,x2),b(x));
                         go to L fi ;
            z
          end
```

A idéia do consequente do esquema III.7 do catálogo é baseada na idéia de (14).

Um trabalho feito pouco tempo antes do último citado, foi o trabalho de H.R.Strong Jr. /30/ que se propos a estabelecer os fundamentos para o estudo sistemático da transformação de esquemas de funções,recursivos em esquemas iterativos. Aparentemente é nesse artigo que se introduz alguns termos de classificação de tipos de recursão, como por exemplo recursão linear (cf.I.3.2) e outros. Strong, nesse trabalho, não está interessado, precisamente, em regras de eliminação mas está interessado em fornecer algoritmos que traduzam um esquema de função recursivo de um determinado tipo num esquema iterativo equivalente. Apesar disso, aparece nesse artigo, a regra de eliminação:

```
          proc f = (t1 x) t2:
(15)        if p(x) then a(f(b(x)))
                    else d(x) fi
```

```
          proc f = (t1 y) t2:
            begin t1 x:=y,t2 z;
(16)          while p(x) do x:=b(x) od;
              (x,z):=(y,d(x));
              while p(x) do (x,z):=(b(x),a(z)) od ;
              z
            end
```

Este esquema generaliza o esquema de Morris, permitindo que d dependa de x. Por isso aparece esse primeiro *while* adicional em (16). A idéia do consequente dessa regra é muito

semelhante à idéia do consequente de III.4.

Posteriormente, juntamente com S.A.Walker, Strong publicou um outro trabalho /32/ na mesma linha do anterior. Contudo, os autores não trabalham diretamente sobre os esquemas recursivos mas sobre grafos dirigidos associados aos esquemas. Nesse artigo, também, foi introduzido o conceito de recursão encaixada (cf.I.3.3) . Walker e Strong, no apêndice do trabalho, enunciam uma série de regras de eliminação. Essas regras visam exemplificar as caracterizações por eles apresentadas. Muitas delas nunca tinham ocorrido antes na literatura e, dessa maneira serviram de fonte para várias regras do capítulo III. São as seguintes as regras baseadas nas regras de Walker e Strong: III.4, III.12, III.17, III.19 e III.23. As regras III.13 e III.18 são generalizações de regras por eles apresentadas.Vale a pena destacar que em nenhum dos trabalhos anteriores aparece o uso explícito de pilha para se construir esquemas iterativos com pilha equivalente. Em alguns dos esquemas essa pilha só é usada para se armazenar o valor dos parâmetros, como é o caso do esquema III.19. Em outros ela também serve para se empilhar uma indicação do endereço do retorno (III.13, III.18 e III.23).

Em 1973, aparece no trabalho de E.W.Dijkstra /14/ , sobre a construção de uma base axiomática simples para a semântica de uma linguagem de programação, a seguinte regra de eliminação:

<u>proc</u> f = <u>void</u>:

(17) <u>if</u> p <u>then</u> R;f <u>fi</u>

<u>proc</u> f = <u>void</u>:

(18) <u>while</u> p <u>do</u> R <u>od</u>

Aparentemente é a primeira vez que se faz uma regra envolvendo um procedimento que não é do tipo função. Partindo

de (17) e utilizando os axiomas desenvolvidos para o comando *if* e para as chamadas de função, Dijkstra definiu o comando *while* e enunciou os axiomas que permanecem invariantes sob o mesmo. Essa regra é englobada pela regra III.1.

Em 1973, J.Darlington e R.M.Burstall publicaram a primeira versão do trabalho /10/ descrevendo um sistema que refina programas e dando uma idéia de sua implementação.

Esse sistema opera quatro tipos de transformações:

i) remoção de recursão;

ii) eliminação de computações redundantes por dois processos: combinar malhas e procurar subexpressões comuns, para serem calculadas uma vez só;

iii) troca de uma chamada de procedimento por seu respectivo corpo (expansão).

iv) reaproveitamento de posições de memórias não mais necessárias.

O sistema trabalha sob o comando do programador. É ele quem determina qual das quatro operações descritas acima deve ser feita para conseguir uma versão mais eficiente do programa. A vantagem do processo é que podemos partir de um programa recursivo claramente formulado, cuja corretude é fácil de provar; o sistema se incumbe de aplicar as transformações escolhidas, de forma a garantir a equivalência das versões obtidas; procura-se chegar a uma solução final mais eficiente.

A contribuição dos autores para o nosso assunto, foi a elaboração de uma tabela de regras que era utilizada na transformação (i). Tais regras estão englobadas em várias regras do catálogo. Da tabela destacamos duas regras originais. A primeira delas, abaixo, tem como antecedente o esquema (2) modificado: introduz-se uma função unária c aplicada ao primeiro argumento (x) de a. Essa alteração, também poderia ter sido introduzida nas regras de Cooper e de Paterson e Hewitt com facilidade.

```
        proc f = (t1 x) t2:
(19)      if p(x) then a(c(x),f(b(x)))
                  else d fi
```

$$a(u,a(v,w)) = a(a(u,v),w) \qquad (20)$$

```
        proc f = (t1 y) t2:
          begin   t1 x:=y,t2 z;
            if p(x) then (z,x):=(c(x),b(x))
                         while p(x) do
(21)                                (z,x):=(a(z,c(x)),b(x))
                                 od;
                         z:=a(z,d)
                    else z:=d fi;
             z
          end
```

Observe-se que só se pode provar a consistência da regra se a função a tiver a propriedade associativa. Tomando-se um x_o tal que n é o menor inteiro para o qual $p(b^n(x_o))$ é falso, teremos respectivamente em (19) e (21) os seguintes valores computados:

$$a(c(x_o),a(c(bx_o),\ldots,a(c(b^{n-2}x_o),a(c(b^{n-1}x_o),d))\ldots))$$

e

$$a(a(\ldots(a(a(c(x_o),c(bx_o)),c(b^2x_o)),\ldots,c(b^{n-1}x_o)),d)$$

o que mostra a necessidade da condição (20). A regra III.5 baseia-se nesta regra de Burstall e Darlington.

A outra regra que citamos, tem como antecedente o es

quema (12). As condições de aplicabilidade da regra são altamente restritivas, sendo que elas foram, provavelmente, obtidas a partir de análises da função de Fibonacci:

```
proc fib=(int n) int:
  if n>1 then fib(n-1)+fib(n-2)
         else 1   fi
```

A regra é:

```
       proc f = (t1 x) t2:
         if p(x) then a(f(b(x)),f(c(x)))
(22)
              else d
```

$$\begin{array}{ll} a(u,v) = a(v,u) & (23\ ) \\ c(u) = b(b(u)) & (23') \\ p(u) \wedge p(b(u)) \Rightarrow p(b(b(u))) & (23") \end{array}$$

```
        proc f = (t1 y) t2:
          begin t1 x:=y, t2 z1:=d,z2:=d;
(24)        while p(x) do
                   (x,z1,z2):=(b(x),a(z2,z1),z1)
                  od;
            z1
          end
```

Essa não é exatamente a regra apresentada pelos autores, uma vez que a pré-condição (23") foi acrescentada em /34/ por Wössner e nos parece de fato necessária, já que, z_2 é inicializado com d em (24). A generalização dessa regra é III.20.

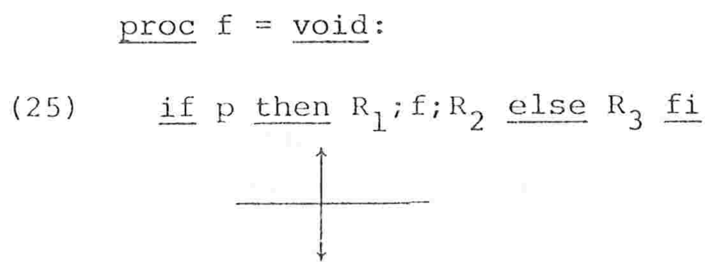
Posteriormente os autores publicaram outros trabalhos, /12/ e /13/, onde eles relatam o funcionamento e algumas aplicações de um outro sistema bastante diferente do anterior. Esse

novo sistema operaria não mais com programas, mas com funções recursivas no sentido matemático com algumas extensões de notação. As transformações seriam feitas através de algumas regras de inferência das quais destacamos expansão (cf.I.2.12) e contração (cf.I.2.13).

Em 1974, aparecem publicados nos anais do Seminários sobre Métodos de Programação, realizados em Munique, alguns trabalhos sobre eliminação de recursão. Algumas novas contribuições surgem nesses trabalhos. Kröger /19/, por exemplo, introduziu, sem prova, uma outra regra que envolve procedimento que não é do tipo função e que pode ser considerada como transformação da regra de Strong, cujo par de esquemas é (15)-(16):

<u>proc</u> f = <u>void</u>:

(25) <u>if</u> p <u>then</u> R_1;f;R_2 <u>else</u> R_3 <u>fi</u>



<u>proc</u> f = <u>void</u>:

  <u>begin</u>

    <u>while</u> p <u>do</u> R_1 <u>od</u>;

(26) R_3;

    <u>dojustasoften</u> R_2 <u>od</u>

  <u>end</u>

Observa-se a introdução do comando <u>dojustasoften</u> que tem nessa regra o seguinte significado: R_2 deve ser executado tantas vezes quantas for executado R_1, isto é,

<u>proc</u> f = <u>void</u>:

  <u>begin</u> <u>int</u> i:=0;

    <u>while</u> p <u>do</u> i:=i+1;R_1 <u>od</u>;

    R_3;

    <u>while</u> i>0 <u>do</u> i:=i-1;R_2 <u>od</u>

  <u>end</u>

(Obviamente, i não ocorre em R_1, R_2 e R_3.).

A regra III.3 é uma generalização dessa regra para o caso de permitirmos parâmetros em f.

Wössner /33/, usando o método da indução recursiva provou a consistência de uma versão corrigida e otimizada da regra (22)-(24) de Burstall e Darlington. Essa otimização consiste em suprimir a pré-condição (23) trocando em (24) a atribuição $z_1:=a(z_2,z_1)$ por $z_1:=a(z_1,z_2)$.

Em 1976, R.S.Bird /6/ faz um estudo do esquema recursivo abaixo:

(27)
```
proc f = (t1 x) void:
    if p(x) then R1;f(a(x));f(b(x));R2 fi
```

Bird estava interessado em encontrar um esquema iterativo com pilha eficiente que fosse equivalente a esse esquema. Para isto, ele fez uma abordagem diferente na maneira de atacar a questão, decompondo o problema em duas partes: supôs que R_2 fosse vazio e depois que R_1 fosse vazio.

Obteve assim duas regras de eliminação para particularizações do esquema original. Na obtenção da primeira regra, cujo antecedente se reduziu a:

(28)
```
proc f = (t1 x) void:
    if p(x) then R1;f(a(x)),f(b(x)) fi
```

Bird utilizou a regra III.1 para a segunda chamada recursiva e, para tratar a primeira chamada, empregou uma pilha como na regra III.2, resultando regra III.25. Para obtenção da segunda, cujo antecedente é:

(29)
```
proc f = (t1 x) void:
    if p(x) then f(a(x));f(b(x)); R2 fi
```

o autor define uma outra versão do procedimento f, usando um procedimento auxiliar g(x). Esse procedimento é:

```
proc f' = (t1 x) void:
  begin
    stack s;
    proc g = (t1 x') void:
(30)    if p(x) then push (s,x);g(b(x));g(a(x)) fi;
    s:=nil;
    g(x);
    while ¬ empty(s) do pop(s,x);R2 od
  end
```

Usando o método da indução recursiva de Morris, Bird provou que os procedimentos f e f' são equivalentes. Então, foi aplicada a regra anterior ao procedimento g, obtendo-se um esquema iterativo com pilha. O esquema final obtido é o consequente da regra III.26.

Finalmente, para obter o esquema mais geral equivalente a (27), Bird procede de uma maneira semelhante ao último caso. Este esquema é o III.24, e sua prova e dedução são apresentados no ítem IV.4.

Um grupo que executou um trabalho com objetivos semelhantes ao de Burstall e Darlington foi o grupo da Universidade Técnica de Munique, liderados por F.L.Bauer /2/. Eles desenvolveram um sistema denominado CIP ("computer-aided, intuition-guided, programming") /4/, que supervisiona o processo de desenvolvimento de um programa. A linguagem empregada pelo sistema é de aplicação bastante ampla envolvendo desde formulações matemáticas com conjuntos e predicados com quantificadores até comandos orientados para a máquina. Essa linguagem está descrita em /3/. Existe no sistema uma tabela das mais variadas transformações que incluem regras de eliminação. O programador pode consultar essa tabela e ordenar ao sistema que aplique uma determinada transformação a uma versão do programa que está sendo desenvolvido. O sistema não possui uma estratégia de aplicação de transformações.

Ele, simplesmente, através de ordem do programador, se encarrega de fazer todas as operações puramente mecânicas, como no caso do sistema inicial de Burstall e Darlington. Uma outra função do sistema é a documentação de todas as versões do programa. Dessa forma, é possível voltar a uma das versões anteriores quando a estratégia estabelecida pelo programador levou a uma versão não desejada.

Muitos trabalhos desse grupo foram publicados nos anos recentes. Alguns tratam da síntese de programa recursivos através de CIP, como por exemplo /7/, /16/, /27/. Outros, que descrevemos abaixo, apresentam regras aplicáveis a esquemas recursivos muito mais elaborados dos que vinham sendo estudados até aqui.

Em 1976, H.Partsch e P.Pepper escreveram um artigo , /26/ com o propósito de apresentar regras de eliminação que fossem aplicáveis a procedimentos do tipo:

```
        proc hanoi = (int i, char ta,tb,tc) void:
          if i>0 then hanoi(i-1,ta,tc,tb);
(31)                  print(("mova disco",i,"de",ta,"para",tb));
                      hanoi (i-1,tc,tb,ta) fi
```

Esse procedimento soluciona o conhecido problema da Torre de Hanoi que é assim enunciado: existem três torres: ta, tb e tc, na cidade de Hanoi. Na torre ta estão empilhados n discos de tamanhos diferentes em ordem decrescente de tamanho, isto é, o maior está mais embaixo. O objetivo do problema é mudar os discos da torre ta para a torre tb, usando, se necessário, a torre tc como auxiliar, obedecendo as seguintes regras: i) somente um disco pode ser movido de cada vez; ii) um disco só pode ser colocado no chão ou sobre um outro disco maior que ele.

Inicialmente, Partsch e Pepper analisaram o seguinte esquema recursivo, que é uma generalização de (31):

```
       proc f = (int x):
(32)
         if x>0 then f(x-1);R1;f(x-1) fi
```

Eles afirmam e provam que o número de vezes que R_1 é executado é 2^n-1 na chamada f(n). Além disso eles conseguem estabelecer e provar a relação que existe entre a ordem i da execução de R_1 e o valor corrente que x deve assumir. Com isto eles obtêm a primeira regra do artigo que é a regra III.27.

A partir daí eles generalizam a regra obtida e chegam finalmente a um caso bem geral que está transcrito na regra III. 28. No capítulo IV, apresentamos uma prova dessa regra, que generaliza a prova de Partsch e Pepper para a regra III.27.

Aplicando tais regras ao procedimento (31) e fazendo uma série de pequenas transformações no procedimento iterativo obtido, os autores, chegam à eficiente solução enunciada abaixo

```
proc hanoi=(int n,char t1,t2,t3)void:
  begin char ta:=t1,tb:=t2,tc:=t3,int c;
    if odd n then (tb,tc):=(tc,tb) fi;
    for c to 2↑n-1
      do int i,p;
(41)        (i,p):=(1,c);
            while p mod 2=0 do (i,p):=(i+1,p÷2) od;
            if odd(i-1) then print(("move disco",i,"de",tc,"para",ta))
                        else print(("move disco",i,"de",ta,"para",tc))
                                                              fi;
           (ta,tb,tc):=(tb,tc,ta) od;
    if odd n then (ta,tc):=(tc,ta) fi
  end
```

No último ítem do artigo Partsch e Pepper ainda apresentam outras generalizações de (32). A que achamos mais interessante é a regra III.29.

Outro artigo é o /5/ de Bauer, Partsch, Pepper e Wössner, que retomam várias regras surgidas em outros trabalhos , transformando os esquemas originais em procedimentos que permitem efeitos colaterais, operando sobre variáveis locais e globais. Uma única regra nova foi, lá introduzida, dizendo respeito, mais uma vez, a uma variação do esquema (2) que é a que empregaremos nas regras do capítulo III. O consequente dessa regra, dessa vez, utiliza-se de uma pilha para armazenar o valor do parâmetro x. A regra, da maneira como aparece nesse trabalho, é a III.9.

M.Broy em /8/, apresentou um trabalho muito interessante que consiste da síntese de um programa que computa a função de Ackermann cuja definição recursiva é dada a seguir. O programa iterativo com pilha obtido é muito mais eficiente do que outros que apareceram anteriormente na literatura como é o caso do apresentado por Rice em /29/.

```
        proc Acker=(int m,n) int:
          if m=0 then n+1
(42)             else if n=0 then Acker (m-1,1)
                             else Acker (m-1,Acker(m,n-1)) fi
```

Para remover as duas primeiras recursões do esquema foi usada a regra III.1, visto que tais chamadas são iterativas.

Depois disso feito, restou uma única chamada recursiva linear. Para eliminá-la, Broy utilizou uma pilha, enunciando um princípio que nos inspirou na elaboração da regra III.2.

Em 1977, R.Steinbrüggen, /31/, desenvolveu e provou uma regra aplicável sob certas condições a uma forma modificada do esquema (12):

```
        proc f = (t_1 x) t_2:
(43)      if p(x) then a(e(x),[f(b(x)),f(c(x))])
             else d(x) fi
```

onde [,] denota um par de valores. É o único esquema que conhecemos contendo uma constante de função binária em que cada argumento é um par. Note-se que nos esquemas vistos anteriormente os tipos dos parâmetros e resultados podem ser considerados como compostos mas esse fato não leva à forma de (43). Isto permitiu ao autor apresentar uma regra de transformação que não elimina a recursão mas cujo consequente só possui uma chamada recursiva, isto é, sua recursividade é linear. Essa transformação visou os esquemas que satisfazem as pré-condições:

$$p(x) \land \neg p(b(x)) \Rightarrow f(c(x)) = d_1(b(x))$$

$$p(x) \land p(b(x)) \Rightarrow f(c(x)) = a_1(e_1(b(x)), [f(b(b(x))), f(c(b(x)))])$$

Steinbrüggen pode então aplicar a regra III.5 assumindo a associatividade da constante de função a. Isto levou a uma regra que simplificamos resultando em III.21. Esta regra é mais abrangente do que a de Burstall e Darlington-Wössner que tratava desse tipo de recursão, vista em (22)-(24). Ela aplica-se a problemas como o seguinte exemplo /31/ que calcula $\binom{n}{r}$.

```
        proc comb = (int n,r) int:
(44)      if n>r and r>0 then comb(n-1,r)+comb(n-1,r-1)
                   else 1 fi
```

Como método de prova esse trabalho utiliza além das técnicas de Burstall e Darlington /10/, a técnica de definir asserções que ele prova serem invariantes com as chamadas e que são baseadas no método de Floyd/15/ usadas na prova de programas.

Finalmente, citamos quatro trabalhos que nos parecem relevantes para nosso assunto. Todos eles foram publicados em 1978.

O primeiro deles é um trabalho de Z.Manna e R.Waldinger /22/ que tem por finalidade estudar o uso do método da asserção intermitente na prova da corretude total de programas.Es

te método foi introduzido por Burstall em 1974 /9/. Para fazermos uso desse método devemos inserir condições em forma de comentários em alguns pontos do programa de tal forma que em alguma vez que o controle passar por um dos referidos pontos as condições a eles associadas serão verdadeiras. Isto é, o controle poderá passar muitas vezes pelo ponto sem que as condições especificadas sejam satisfeitas. Se colocarmos um destes comentários, ou asserção intermitente, no fim executável do programa descrevendo as especificações de saída, estaremos provando simultaneamente que o programa termina e produz a saída desejada.

A contribuição importante de Manna para o nosso tema específico é empregar essa técnica para provar a consistência da regra III.22. No capítulo IV, faremos uso desse método para provar as regras III.4 e III.9.

Outro trabalho referente ao assunto é o de M.A. Auslander e H.R. Strong /1/ que apresentam de uma maneira informal uma técnica geral para se eliminar a recursão de um procedimento. A técnica é exemplificada por uma aplicação a um programa que lista todos os ciclos primos de um grafo dirigido.

Essa técnica consiste, basicamente dos seguintes passos:

i) remover a recursão de predicados e transformar os comandos do tipo ...do R od que tiverem em R uma chamada recursiva em comandos do tipo L:if...then R; go to L fi.

ii) criar pilhas e empilhar todos os valores de variáveis locais e parâmetros e uma indicação do ponto de retorno antes de cada chamada e desempilhá-los depois da chamada. Esse passo deve ser detalhadamente estudado (podendo ser eventualmente eliminado, como no caso de recursão iterativa) para que só sejam empilhadas as informações que não puderem ser obtidas de outra maneira como por exemplo uma variável cujo valor pode ser deduzido de outra.

iii) transformar os parâmetros e variáveis locais ao procedimento em variáveis globais tendo cuidado em re-denominar os identificadores. Converter as declarações de procedi

mentos em rótulos, converter cada chamada de procedimento em *go to*, converter cada retorno em *go to* fazendo uma consulta à indicação de retorno da pilha.

Para exemplificar a técnica de Auslander e Strong, faremos uma aplicação da mesma ao programa abaixo, que imprime as 10 primeiras linhas do triângulo de Pascal com auxílio do procedimento (44) que calcula $\binom{n}{r}$:

```
begin
  proc comb(int n,r)real:
    if n>r and r>0 then comb(n-1,r)+comb(n-1,r-1)
                   else 1 fi;
  for i from 0 to 10
   do  newline;
       for j from 0 to 1 do print(comb(i,j))od od
end
```

Obtendo,

```
begin  int n,r,k,real y,z,cb,stack s:=nil,t:=nil;
  for i from 0 to 10
   do newline;
      for j from 0 to i
       do COMB: if n>r and r>0
                then    push(t,1);n:=n-1;
                        go to COMB;
                     L1:push(s,y);push(t,2);
                        (n,r):=(n-1,r-1);
                        go to COMB;
                     L2:cb:=y+z
                else cb:=1;
               if ¬empty(t)
                then pop(t,k);
                     if k=1 then(n,y):=(n+1,cb);go to L1
                            else(n,r,z):=(n+1,r+1,cb);
                                pop(s,y);go to L2 fi;
          print(cb) od od
end
```

A seguir citamos o trabalho de H.Wössner /34/, que é a primeira incursão didática no nosso tema.

Achamos que as contribuições mais importantes que aparecem no artigo referem-se ao estudo de um caso especial de recursão encaixada, à técnica de tabulação de valores correntes e à técnica de desembaraçamento de controle. Sobre recursão encaixada, Wössner estudou o esquema abaixo:

<u>proc</u> f = (<u>t</u>$_1$ x) <u>t</u>$_2$:

(45) <u>if</u> p(x) <u>then</u> f(f(b(x)))

<u>else</u> c(x) <u>fi</u>

Ele apresentou, inicialmente, uma regra geral que não impõe pré-condições aos operadores b e c. Essa regra funciona com o auxílio de um contador que opera da seguinte maneira: toda vez que ocorre o "retorno" de uma chamada, isto é, que p(x) é falso para o valor corrente de x decrementa-se de 1 o contador; caso contrário atualiza-se o valor de x e incrementa-se 1 ao contador. O procedimento termina quando o contador atinge 0. Essa regra é a III.14.

Posteriormente, Wössner deduziu duas outras regras , nas quais há como pré-condição a comutatividade de b e c, isto é, c(b(u)) = b(c(u)). A primeira delas aplica-se no caso em que não há chamada recursiva depois de que se tenha atingido a parte <u>else</u> pelo menos uma vez, isto é, supondo que n é tal que

$$p(b^n\ x_o) \wedge \forall i\ [0 \leq i < n \wedge \neg p(b^i\ x_o)]$$

então $f(x_o) = c^{n+1}\ b^n\ x_o$. Isto é o mesmo que dizer que (45) é equivalente ao esquema recursivo linear (15) (cf.I.3.5) onde as funções d e a são substituídas por c. A regra é a III.15.

A segunda delas, regra III.16, só se aplica aos procedimentos que têm a propriedade p(x)⟹p(c(b(x))), além, é claro , da pré-condição da comutatividade de c e b. Wössner esboçou também a prova dessas regras, empregando o método da indução recursiva de McCarthy.

A técnica de tabulação de valores correntes só se aplica às funções recursivas para os quais existe uma ordem total

$\geq$ sobre o tipo do parâmetro t_1 tal que na computação de $f(x_o)$ todas as chamadas recursivas que ocorrem usam somente valores de argumentos que precedem x_o segundo $\geq$. Precisamos também conhecer para que valores do argumento a função f termina. Dessa maneira podemos tabular sucessivamente todos os valores assumidos por f, a começar dos valores para os quais f termina, até atingir o valor $f(x_o)$. No corpo da função f devemos substituir as eventuais chamadas recursivas f(b(x)) por uma consulta à tabela na posição onde se encontra tal valor. Na verdade, uma variante simplificada dessa técnica foi introduzida por H.G.Rice em 1965 /29/, quando este autor apresentou um programa esquemático em FORTRAN IV que calculava funções recursivas f para as quais cada chamada só dependia do valor da chamada anterior.

A técnica do desembaraçamento de controle tem por objetivo rearranjar o procedimento recursivo de tal forma que o menor número possível de parâmetros e variáveis locais precisem ser empilhados e ainda que se tente evitar o empilhamento de uma indicação do endereço de retorno. Para se conseguir isso, as chamadas recursivas devem ser isoladas com o auxílio da declaração de variáveis locais, construindo-se uma sequência de atribuições de cada valor intermediário calculado para aquelas variáveis. Diz-se que um parâmetro ou variável local está desembaraçado se o seu valor não é usado antes e depois de uma chamada recursiva. Os valores das variáveis e parâmetros desembaraçados não precisam ser empilhados. Por exemplo o esquema de procedimento abaixo que é equivalente a (45) tem as variáveis locais e o parâmetro desembaraçados.

```
proc f = (t1 y) t1:
  begin  t1 x:=y,z1,z2;
    if p(x) then x :=b(x);
                 z1:=f(x);
                 z2:=f(z1);
                 z2
          else c(x) fi
  end
```

Finalmente, citamos o trabalho de G.Huet e B.Lang/17/, cujo resultado principal é descrever e provar um algorítmo de reconhecimento de qual regra pode ser aplicada numa dada função recursiva. Como ilustração, no apêndice do artigo são incluídos 6 regras de eliminação. Duas dessas regras são originais: III.10 e III.11.

CAPÍTULO III

CATÁLOGO DE REGRAS DE ELIMINAÇÃO

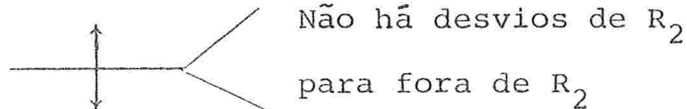
As regras de eliminação de recursão (cf. I.2.8) que se seguem podem ser agrupadas em quatro blocos de acordo com a classificação das chamadas recursivas definidas em I.3 a saber:

a) recursão iterativa, regra 1
b) recursão linear, regras 2 a 13
c) recursão encaixada, regras 14 a 19
d) recursão em cascata, regras 20 a 29

Ao pé de cada regra haverá, quando for o caso, referências a ocorrências de versões da regra na bibliografia. Todos os meta-símbolos de identificadores de variáveis locais e parâmetros declarados nos consequentes das regras e que não ocorrem no antecedente devem ser interpretados (cf. I.2.4) de maneira que não ocorram nas sequências de comandos R interpretadas.

1.

```
proc f = (t1 x) t2:
  begin {1}
    R1;
    if {2} p(x) then R2; f(a(x)); go to L
                else R3 fi;
    R4
  L:{3} end
```




```
proc f = (t1 y) t2:
  begin t1 x:=y;{4}
   L1: R1;
       if {5} p(x) then R2; x:=a(x); go to L1
                   else R3 fi;
       R4;
  L:{6} end
```

2.

```
proc f = (t1 x) t2:
  begin D u;
    R1;
    if p(x) then R2; f(a(x)); R3
             else R4  fi
  end
```

```
proc f = (t1y) t2:
  begin  t1 x:=y, stack s, D u;
    s:=nil;
    L: R1;
       if p(x) then R2; push(s,x); push(s,u);
                     x:=a(x); go to L fi;

       R4;
       while ¬  empty (s) do pop(s,u); pop(s,x);
                                 R3 od
  end
```

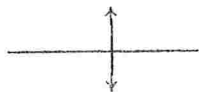
Obs:

1) D u representa a sequência de declarações das variáveis locais do bloco mais externo de f, indicadas por u .

2) R_3 pode alterar o valor de x.

3.

```
proc f = (t1 x) t2:
  if p(x) then R1; f(a(x)); R2
          else R3  fi
```



```
proc f = (t1 y) t2:
  begin t1 x:=y, stack s;
    s:=nil;
    while p(x) do R1; push(s,x); x:=a(x) od;
    R3;
    while ¬ empty (s) do pop(s,x); R2  od;
  end
```

Ref.: /2/, /5/, /19/, /26/

4.

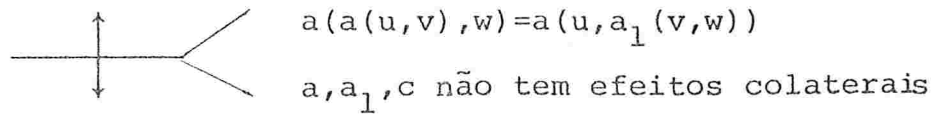
```
proc f = (t1x) t2:
  begin {1}
    if p(x) then R1; a(f(b(x)))
            else R2; c(x) fi;
    {2}
  end
```

```
proc f = (t1y) t2:
  begin t1 x:=y, int k, t2 z;
    {3} k:=0;
        while {4} p(x) do R1;
                          (k,x):=(k+1,b(x)) od;
        R2;
        z:= c(x);
        while {5} k>0 do (k,z):=(k-1,a(z)); z od;
        z {6}
  end
```

Ref.: /2/, /5/, /30/, /32/, /34/

5.

```
proc f = (t1x) t2:
  if p(x) then R1; a(f(b(x)), c(x))
          else R2; d(x) fi
```

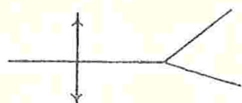



```
proc f = (t1y) t2:
  begin   t1 x:=y, t2 z;
    if ¬  p(x) then R2; z:=d(x)
               else R1;(x,z):=(b(x), c(x));
                    while p(x)
                      do  R1;  (x,z):=(b(x),a1(c(x),z))
                      od;
                      R2; z:=a(d(x), z) fi;
    z
  end
```

Ref.: /5/, /10/, /17/, /31/, /34/

7.

```
proc f = (t1x) t2:
  if     p(x) then R1; a(f(b(x)), c(x))
              else R2; d(x) fi
```

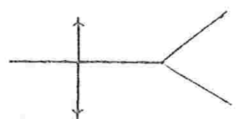


b não tem efeito colateral

```
proc f = (t1x) t2;
  begin t1x:=y, t2z, int j,k;
    k:=0;
    while p(x) do R1;
                    (k,x):=(k+1,b(x)) od;
    R2;
    z:=d(x);
    while k>0 do (k,j,x):=(k-1,k,y);
                  while j>0 do (x,j):=(b(x),j-1)od;
                  z:=a(z,c(x)) od;
    z
  end
```

Ref.: /28/, /32/, /34/

6.
```
proc f = (t1 x) t2:
  if p(x) then R1; a(f(b(x)), c(x))
          else R2; d(x) fi
```



$u=b_1(b(u)=b(b_1(u))$

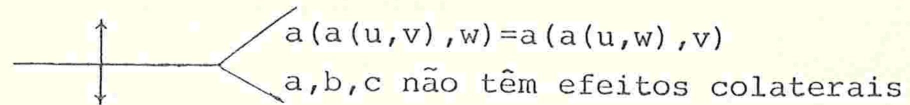
b_1 não tem efeito colateral

```
proc f = (t1 y) t2:
  begin t1 x:=y, t2 z, int k;
    k:=0;
    while p(x) do R1; (x,k):=(b(x),k+1) od;
    R2;
    z:=d(x);
    while k>0 do x:=b1(x);
                 (z,k):=(a(z,c(x)),k-1) od;
    z
  end
```

Ref.: /2/, /5/, /10/, /11/, /17/, /32/, /34/

8.

```
proc f = (t1 x) t2:
  if p(x) then R1; a(f(b(x)), c(x))
         else R2; d(x) fi
```

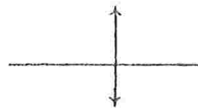



```
proc f = (t1 y) t2:
  begin t1 x:=y, t2 z;
    while p(x) do R1; x:=b(x) od;
    R2;
    (z,x):=(d(x),y);
    while p(x) do (x,z):=(b(x),a(z,c(x))) od;
    z
  end
```

Ref.: /5/, /11/, /17/, /34/

9.

```
proc f = (t1 x) t2:
  begin {1}
    if p(x) then R1; a(f(b(x)), c(x))
            else R2; d(x) fi
    {2}
  end
```

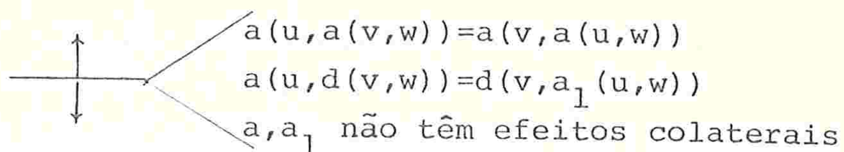


```
proc f = (t1 y) t2:
  begin t1 x, t2 z, stack s;
    s:=nil;
     x:=y;
  {3}push (s,x);
     while{4}p(x) do R1; x:=b(x); push (s,x) od;
     R2;
     z:=d(x);
     pop (s,x);
     while{5} ¬ empty (s) do pop (s,x);
                              z:=a(z, c(x));
                              z od;
     z {6}
  end
```

Ref.: /5/, /34/

10.

```
proc f = (t1 x, t2 y) t3:
  if p(x) then a(x,f(b(x),y))
          else d(x,y) fi
```

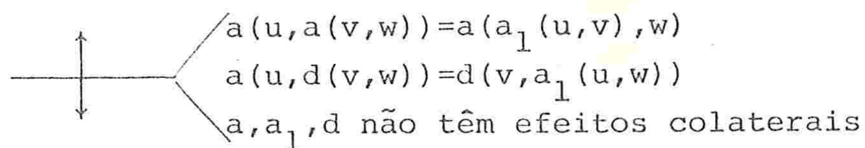



```
proc f = (t1 x1, t2 y2) t3;
  begin t1 x:=x1, t2 y:=y1;
    while p(x) do (x,y):=(b(x), a1(x,y)) od;
    d(x,y)
  end
```

Ref.: /17/

11.

```
proc f = (t1 x, t2 y) t3;
  if p(x) then a(y,f(b(x),c(x)))
          else d(x,y)  fi
```

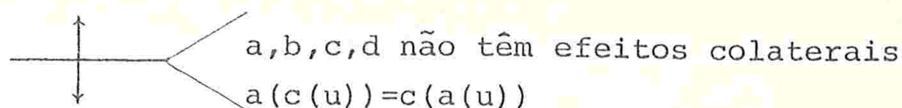



```
proc f = (t1 x1, t2 y1) t3:
  begin t1 x:=x1, t2 y:=y1;
    while p(x) do (x,y):=(b(x), a1(y,c(x))) od;
    d(x,y)
  end
```

Ref.: /17/

12.

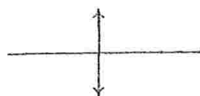
```
proc f = (t1x) t2:
  if p(x) then R1; a(f(b(x)))
         else if q(x) then R2; c(f(d(x)))
                      else R3; e(x) fi fi
```




```
proc f = (t1y) t2
  begin t1 x:=y, t2 z;
   L1: if p(x) then R1; x:=b(x); go to L1
              else if q(x) then R2; x:=d(x);
                           go to L1 fi fi;
       R3;
       (x,z):=(y,e(x));
   L2: if p(x) then (z,x):=(a(z),b(x);
                  go to L2
              else if q(x) then (z,x):=(c(z),d(x));
                             go to L2 fi fi;
       z
  end
```

Ref.: /32/

13.
```
proc f = (t1 x) t2:
  if p1(x) then R1; a1(f(b1(x)), c1(x))
   elif p2(x) then R2; a2(f(b2(x)), c2(x))
    elif .
         :
       elif pn(x) then Rn; an(f(bn(x)), cn(x))
                 else Rn+1; d(x)
```

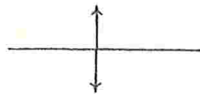


```
proc f = (t1 y) t2:
  begin t1 x:=y, t2 z, stack s;
    s:=nil;
    L1:if p1(x)   then   R1;push(s,x);push(s,1);x:=b1(x);
                         go to L1
        elif p2(x)then R2;push(s,x);push(s,2);x:=b2(x);
                         go to L1
         elif .
              :
           elif pn(x) then Rn;push(s,x);push(s,n);x:=bn(x);
                            go to L1;

      Rn+1;
      z:=d(x);
      while ¬  empty (s)
        do  pop (s,k); pop(s,x);
            case k
             in z:=a1(z,c1(x)),
                z2:=a2(z,c2(x)),
                :
                zn:=an(z,cn(x)) esac  od;
      z
  end
```

14.

```
proc f = (t1 x) t1:
  if p(x) then R1; f(f(b(x)))
          else R2; c(x) fi
```

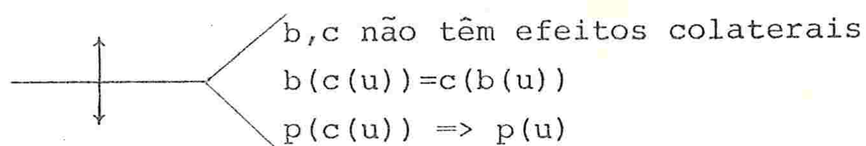


```
proc f = (t1 y) t1:
  begin t1 x:=y, int k:=1;
    if ¬ p(x) then R2; c(x)
              else while k>0
               do if p(x) then R1; (x,k):=(b(x),k+1)
                          else R2; (x,k):=(c(x),k-1) fi
               od;
               x   fi
end
```

Ref.: /34/

15.

```
proc f = (t1 x) t1:
  if p(x) then R1; f(f(b(x)))
          else c(x) fi
```

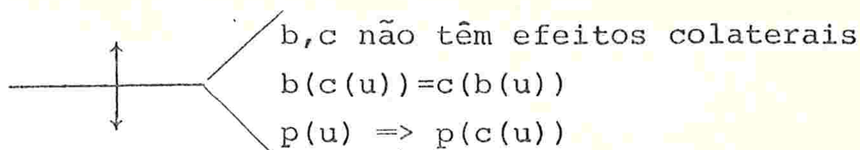



```
proc f = (t1 y) t1:
  begin t1 x:=y, z;
    while p(x) do R1; (x,z):=(b(x),c(b(z))) od;
    c(x)
  end
```

Ref.: /34/

16.

```
proc f = (t1x) t1:
  if p(x) then f(f(b(x)))
          else c(x) fi
```

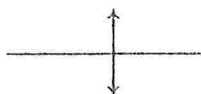



```
proc f = (t1y) t1:
  begin t1x:=y;
    while p(x) do x:=c(b(x)) od;
    c(x)
  end
```

Ref.: /34/

17.

```
proc f = (t1 x) t1:
  if p(x) then R1; f(a(f(b(x))))
          else R2; c(x)  fi
```

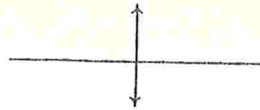


```
proc f = (t1 y) t1:
  begin t1 x:=y, z, int k=1;
    L1:while p(x) do R1; (x,k):=(b(x),k+1) od;
       R2;
       (z,k):=(c(x),k-1);
       if k≠0 then x:=a(z);
                   go to L1  fi;
       z
  end
```

Ref.: /32/

18.

```
proc f = (t1 x) t1:
  if p(x) then R1; f(a1(f(a2(...f(an(f(b(x))))...)))
        else R2; c(x) fi
```



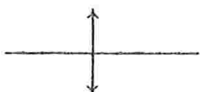
```
proc f = (t1 y) t1:
  begin t1 x:=y, z, stack s;
    s:=nil;
    L: while p(x) do  R1; x:=b(x);
                  for k to n do push (s,k) od od;
       R2;
       z:=c(x);
       if ¬ empty (s)
        then case top (s)
             in  x:=a1(z),
                 x:=a2(z),
                 :
                 x:=an(z) esac;
           remove(s);
           go to L fi;

       z

  end
```

19.

```
proc f = (t1x) t2:
  if p(x) then R1; f(a(f(b(x)), c(x)))
           else R2; d(x) fi
```

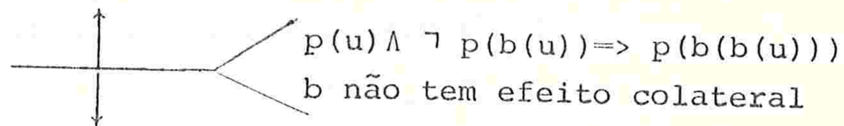


```
proc f = (t1y) t2:
  begin t1 x:=y, t2 z, stack s;
    s:=nil;
    L: while p(x) do R1;push(s,x); x:=b(x) od;
       R2;
       z:=d(x);
       if ¬ empty (s) then pop(x,s);
                           x:=a(z,c(x));
                           go to L  fi;
       z
  end
```

Ref.: /32/

20.

```
proc f = (t1 x) t2:
  if p(x) then a(f(b(x)), f(b(b(x))))
         else d  fi
```

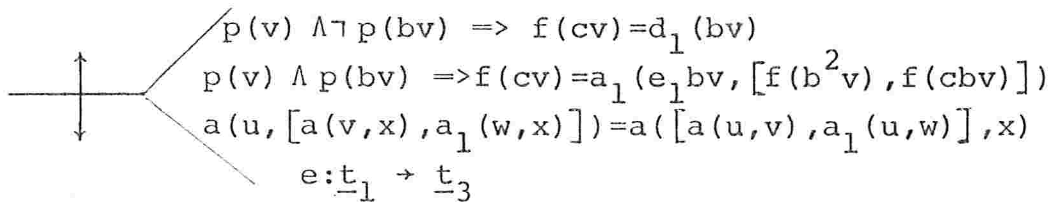



```
proc f = (t1 y) t2:
  begin t1 x:=y, t2 z1:=d, z2:=d;
    while p(x) do
          (x,z1,z2):=(b(x),a(z1,z2),z1)
          od;
    z1
  end
```

Ref.: /10/, /17/, /33/, /34/

21.

```
proc f = (t₁x) t₂:
  if p(x) then a(e(x),[f(b(x)),f(c(x))])
      else d(x) fi
```



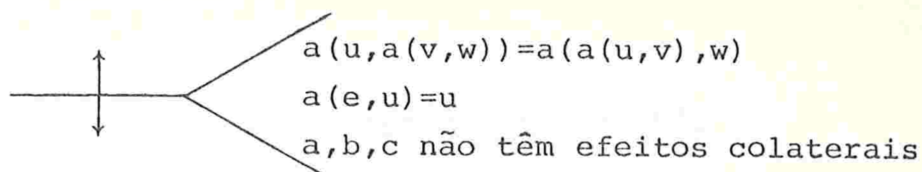

```
proc f = (t₁y) t₂:
  begin t₁x:=y, t₃z;
    if ¬ p(x) then d(x)
          else (x,z):=(b(x),e(x));
               while p(x)
                do z:=[a(z,e(x)),a₁(z,e₁(x))];
                   x:=b(x) od;
               a(z,[d(x),d₁(x)]) fi
  end
```

Ref.: /31/

(Obs.: os argumentos da constante de função a são pares de valores denotados por [,]).

22.

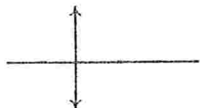
```
proc f = (t1x) t2:
   if p(x) then R1; a(f(b(x)), f(c(x)))
           else R2; d(x) fi
```




```
proc f = (t1y) t2:
  begin t1 x:=y, t2 z, stack s;
    s:=nil;
    push (s,x);
    z:=e;
    while ¬ empty (s)
       do pop(s,x);
          if p(x) then R1; push(s,c(x));
                       push(s,b(x));
                  else R2; z:=a(z,d(x)) fi od;
    z
  end
```

Ref.: /22/

23.
```
proc f = (t1 x) t2:
  if p(x) then R1; a(f(b(x)), f(c(x)))
          else R2; d(x) fi
```



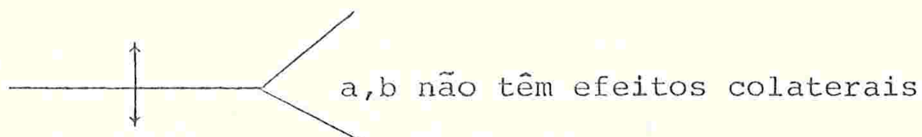
```
proc f = (t1 y) t2:
  begin t1 x:=y, t2 z, stack s;
    s:=nil;
    L1: while p(x) do R1; push (s,x);
                      push (s,1);
                      x:=b(x);
                   od;
        R2;
        z:=d(x);
    L2: if ¬ empty (s)
          then if top(s) = 1 then remove(s);
                                  pop(s,x);
                                  push(s,z);
                                  push(s,2);
                                  x:=c(x);
                                  go to L1
                             else remove (s);
                                  pop(s,x);
                                  z:=a(x,z);
                                  go to L2 fi fi;

        z

  end
```

Ref. /32/

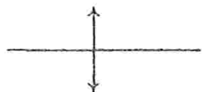
24.
```
proc f = (t1 x) void:
  if p(x) then R1; f(a(x)); f(b(x)); R2 fi
```




```
proc f = (t1 y) void:
  begin t1 x:=y, stack s1, s2, int k:=0;
    s1:=nil; s2:=nil;
    L : while p(x) do R1; push(s1,x); push(s2,b(x));
                      x:=a(x) od;
        if k>0 then k:=0; pop(s1,x); R2 fi;
        if ¬ empty(s2) then pop(s2,x); k:=1;
                                  go to L fi
end
```

25.

```
proc f = (t1 x) void:
  if p(x) then R1; f(a(x)); f(b(x)) fi
```

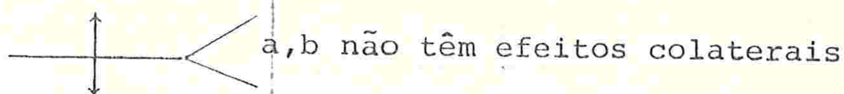


```
proc f = (t1 y) void:
  begin t1 x:=y, stack s;
    s:=nil;
    L :while p(x) do R1; push(s,x); x:=a(x) od;
       if ¬ empty (s) then pop(s,x); x:=b(x);
                                  go to L fi
  end
```

Ref.: /6/

26.

```
proc f = (t1x) void:
  if p(x) then f(a(x)); f(b(x)); R2 fi
```



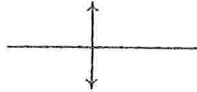

```
proc f = (t1y) void:
  begin t1x:=y, stack s1, s2;
    s1:=nil; s2:=nil;
    push (s2,x);
    while ¬ empty (s2)

     do pop (s2x);
        while p(x) do push(s1,x);push(s2,a(x));
                      x:=b(x) od od;
    while ¬ empty (s1) do pop (s1,x); R2 od
  end
```

Ref. /6/

27.

```
proc f = (int x)void:
  if x>0 then f(x-1); R1; f(x-1) fi
```

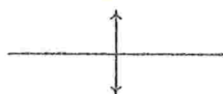


```
proc f = (int y)void:
  begin int x:=y,j;
    proc g = (int j) int:
      begin int k:=1, n:=j ;
        while n mod 2=0
         do  (k,n):=(k+1,n÷2) od;
        k
      end;
    for j to 2↑y-1 do x:=g(j); R1 od;
  end
```

Ref.: /5/, /26/, /34/

28.

```
proc f = (int x) void:
  if i>0 then R_o;
            f(x-1); R_1; f(x-1);...;R_{k-1}; f(x-1);
            R_k
        else R_{k+1} fi
```

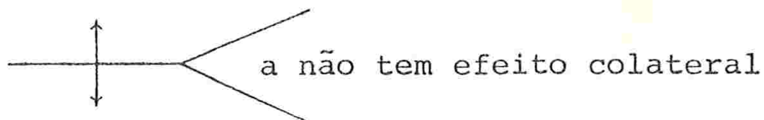


```
proc f (int y) void;
  begin int i, j, m, x;
    proc g=(int i, ref int j, m)void:
      begin int n:=i;
        m:=1;
        while n mod k=0
         do (m,n):=(m+1,n ÷k) od;
        j:=n mod k
      end;
    for x from y by -1 to 1 do R_o od;
    x:=0; R_{k+1};
    for i to k↑j-1
     do g(i, j, m);
        for x from 1 by 1 to m-1 do R_k od
        x:=m;
        case j in R_1, R_2,...,R_{k-1} esac;
        for x from m-1 by -1 to 1 do R_o od;
        x_o:=0; R_{k+1} od;
    for x from 1 by 1 to y do R_k od;
  end
```

Ref.: /26/

29.

```
proc f = (t1x) void:
  if p(x) then f(a(x)); R1; f(a(x)) fi
```




```
proc f = (t1y) void:
  begin t1x:=y, int i,j,n:=0;
    proc g = (int i) int:
      begin int k:=1, n1:=i;
        while n1 mod 2=0
         do (k,n1):=(k+1, n1÷2) od;
        k
      end;
    while p(x) do n:=n+1; x:=a(x) do;
    for i to 2↑n-1
     do    (j,x):=(g(i),y);
           for m to n-j do x:=a(x) od;
           R1  od
  end
```

Ref.: /26/

CAPÍTULO IV

APLICAÇÕES DE MÉTODOS DE PROVA

1. DEFINIÇÕES

Nas secções que se seguem empregaremos as seguintes definições:

1.1 - Seja R uma sequência de comandos. A expressão

$$R^{x_1,x_2,\ldots,x_m}[v_1,v_2,\ldots,v_m]$$

denota a execução de R em que as variáveis $x_1,x_2,\ldots,x_m$ assumem os valores $v_1,v_2,\ldots,v_m$ respectivamente.

1.2 - Um instantâneo (cf.I.2.9) S_1 é <u>idêntico</u> a um instantâneo S_2, denotado por $S_1 \equiv S_2$, se as informações descritas em S_1 são as mesmas que as informações descritas em S_2.

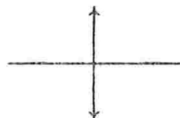
1.3 - Sejam dois instantâneos S_1 e S_2. Dezemos que S_1 é <u>idêntico</u> a S_2 <u>a menos</u> de $x_1,x_2,\ldots,x_m$, denotado por $S_1 \equiv S_2/\{x_1,x_2,\ldots,x_m\}$ se as informações descritas por S_1 coincidem com as informações descritas por S_2, a menos das variáveis $x_1,x_2,\ldots,$

x_m que podem não ter valores iguais ou não estarem definidas em S_1.

1.4 - Dados dois esquemas que definem procedimentos f e f_1, dizemos que as chamadas $f(x_o,x_1,\ldots,x_m)$ e $f_1(y_o,y_1,\ldots,yn)$ são <u>equivalentes</u> se quando elas forem efetuadas no mesmo instantâneo, ou ambas não retornam, ou retornam computando o mesmo valor e produzindo instantâneos idênticos.

1.5 - A seguinte transformação (T_1) define o comando <u>while</u>:

L:<u>if</u> p <u>then</u> R_1; <u>go</u> <u>to</u> L <u>else</u> R_2



<u>while</u> p <u>do</u> R_1 <u>od</u>; R_2

1.6 - Dizemos que um procedimento f está <u>indefinido</u> para um certo valor x_o se o valor x_o é indefinido ou se a chamada $f(x_o)$ não retorna. Se a chamada $f(x_o)$ retorna então dizemos que ela é <u>definida</u>.

2. MÉTODO DA INDUÇÃO RECURSIVA

Para exemplificar o método da indução recursiva de McCarthy /24/, faremos uso do mesmo para provar uma simplificação da regra III.8. Esta simplificação consiste em não permitir a ocorrência de sequências de comandos representadas pelos meta-símbolos R_1 e R_2 e não permitir efeitos colaterais nas interpretações dos símbolos de constantes de predicados e funções. Portanto, os esquemas de procedimento da regra serão transformados em esquemas de função. Como neste caso não haverá efeito colateral, na prova da equivalência dos esquemas não é necessário considerarmos os instantâneos, mas basta considerarmos o valor computado pela função.

A prova que apresentamos é baseada na prova da re

gra (2)-(4) do Capítulo II que ocorreu no trabalho de Cooper /11/.

Para aplicarmos o método de McCarthy precisamos:

a) Reescrever os esquemas de função em forma de equações funcionais. A regra será reescrita da seguinte maneira:

(1) f(x) = <u>if</u> p(x) <u>then</u> a(f(b(x)),c(x)) <u>else</u> d(x) <u>fi</u>

a(a(u,v),w) = a(a(u,w),v) (2)

(3) $f_1(x) = g_2(x,g_1(x))$

(4) $g_1(x)$ = <u>if</u> p(x) <u>then</u> $g_1(b(x))$ <u>else</u> d(x) <u>fi</u>

(5) $g_2(x,z)$ = <u>if</u> p(x) <u>then</u> $g_2(b(x),a(z,c(x)))$ <u>else</u> z <u>fi</u>

Podemos mostrar que o conjunto de equações (3), (4) e (5) que compõem o consequente da regra é equivalente ao consequente simplificado da regra III.8 pelas seguintes transformações:

A equação (4) pode ser reescrita como:

<u>proc</u> g_1=($\underline{t}_1$ x) $\underline{t}_2$:

<u>if</u> p(x) <u>then</u> $g_1(b(x))$ <u>else</u> d(x) <u>fi</u>

que é equivalente pela regra III.1 a:

<u>proc</u> g_1=($\underline{t}_1$ y) $\underline{t}_2$:

<u>begin</u> $\underline{t}_1$ x:=y;

L:<u>if</u> p(x) <u>then</u> x:=b(x); <u>go</u> <u>to</u> L <u>else</u> d(x) <u>fi</u>

<u>end</u>

Neste último esquema podemos empregar a transformação T_1 (cf.1.5), obtendo:

```
        proc g1=(t1 y) t2:
          begin t1 x:=y;
(6)         while p(x) do x:=b(x) od;
            d(x)
          end
```

Aplicando-se a mesma sequência de transformações à equação (5) obtém-se:

```
        proc g2=(t1 y, t2 z1) t2:
          begin t1 x:=y, t2 z:=z1;
(7)         while p(x) do (x,z):=(b(x),a(z,c(x))) od;
            z
          end
```

Reescrevendo a equação (3) em forma de esquema de função tem-se:

```
        proc f1=(t1 x) t2:
          g2 (x,g1(x))
```

que é equivalente a:

```
        proc f1=(t1 y) t2:
          begin t1 x:=y, t2 z;
            (x,z):=(y,g1(x));
            g2(x,z)
          end
```

que, por expansão, usando-se (6) e (7), nas chamadas de g_1 e g_2 respectivamente é equivalente ao consequente da regra III.8 que transcrevemos abaixo:

```
proc f1=(t1 y) t2:
  begin t1 x:=y, t2 z;
    while p(x) do x:=b(x) od;
    (x,z):=(y,d(x));
    while p(x) do (x,z):=(b(x),a(z,c(x))) od;
    z
  end
```

b) enunciar as seguintes propriedades sobre expressões condicionais (cf.II) /24/:

b1) **if** p(x) **then** **if** p(x) **then** R_1 **else** R_2 **fi**
else **if** p(x) **then** R_3 **else** R_4 **fi** **fi**
=**if** p(x) **then** R_1 **else** R_4 **fi**

b2) f(**if** p(x) **then** E_1 **else** E_2 **fi**)=**if** p(x) **then** f(E_1)
else f(E_2) **fi**

onde E_1 e E_2 são expressões.

Primeiramente vamos provar que

$$g_2(b(x),\ a(z,c(x))) = a(g_2(b(x),\ z),\ c(x)) \tag{8}$$

Para isso, vamos definir duas novas funções g_3 e g_4 da seguinte maneira:

$$g_3(x,y,z) = g_2(x,a(z,y)) \tag{9}$$

(10) $g_4(x,y,z) = a(g_2(x,z),y)$

Lema 2.1

As equações de funções (9) e (10) são equivalentes.

Prova

Para provar que g_3 e g_4 são equivalentes, vamos definir uma equação funcional recursiva $g_5(x,y,z)$:

$$g_5(x,y,z) = \underline{\text{if}}\ p(x)\ \underline{\text{then}}\ g_5(b(x),y,a(z,c(x)))\ \underline{\text{else}}\ a(z,y)\ \underline{\text{fi}}$$

e vamos provar que ela é satisfeita por g_3 e g_4.

$g_3(x,y,z) = \underline{\text{if}}\ p(x)\ \underline{\text{then}}\ g_2(b(x),a(a(z,y),c(x)))\ \underline{\text{else}}\ a(z,y)\ \underline{\text{fi}}$

por expansão usando (5)

$= \underline{\text{if}}\ p(x)\ \underline{\text{then}}\ g_2(b(x),a(a(z,c(x)),y))\ \underline{\text{else}}\ a(z,y)\ \underline{\text{fi}}$

pela pré-condição (2)

$= \text{if}\ p(x)\ \text{then}\ g_3(b(x),y,a(z,c(x)))\ \underline{\text{else}}\ a(z,y)\ \underline{\text{fi}}$

por contração usando (9)

$g_4(x,y,z) = a(\underline{\text{if}}\ p(x)\ \underline{\text{then}}\ g_2(b(x),a(z,c(x)))\ \underline{\text{else}}\ z\ \underline{\text{fi}},y)$

por expansão usando (5)

$= \underline{\text{if}}\ p(x)\ \underline{\text{then}}\ a(g_2(b(x),a(z,c(x))),y)\ \underline{\text{else}}\ a(z,y)\ \underline{\text{fi}}$

pela propriedade b2

$= \underline{\text{if}}\ p(x)\ \underline{\text{then}}\ g_4(b(x),y,a(z,c(x)))\ \underline{\text{else}}\ a(z,y)\ \underline{\text{fi}}$

por contração usando (10)

O que demonstra o lema □

Teorema 2.1

As equações de funções (1) e (3) são equivalentes.

Prova

Vamos mostrar que as funções f e f_1 de (1) e (3), respectivamente, satisfazem a equação funcional recursiva abaixo:

$$f_2(x) = \underline{\text{if}}\ p(x)\ \underline{\text{then}}\ a(f_2(b(x)),c(x))\ \underline{\text{else}}\ d(x)\ \underline{\text{fi}}$$

A equação da função f obviamente satisfaz a equação funcional acima. Analisemos a equação da função f_1.

$f_1(x) = g_2(x, g_1(x))$

por definição (3)

$= \underline{\text{if}}\ p(x)\ \underline{\text{then}}\ g_2(b(x),a(g_1(x),c(x)))\ \underline{\text{else}}\ g_1(x)\ \underline{\text{fi}}$

por expansão usando (5)

$= \underline{\text{if}}\ p(x)\ \underline{\text{then}}\ a(g_2(b(x),g_1(x)),c(x))\ \underline{\text{else}}\ g_1(x)\ \underline{\text{fi}}$

pelo lema 2.1 considerando x,y,z respectivamente como b(x), c(x), $g_1(x)$

$= \underline{\text{if}}\ p(x)\ \underline{\text{then}}\ a(g_2(b(x),\ \underline{\text{if}}\ p(x)\ \underline{\text{then}}\ g_1(b(x))$
$\underline{\text{else}}\ d(x)\ \underline{\text{fi}}),c(x))$
$\underline{\text{else}}\ \underline{\text{if}}\ p(x)\ \underline{\text{then}}\ g_1(b(x))\ \underline{\text{else}}\ d(x)\ \underline{\text{fi}}\ \underline{\text{fi}}$

por expansão usando (4)

$= \underline{\text{if}}\ p(x)\ \underline{\text{then}}\ \underline{\text{if}}\ p(x)\ \underline{\text{then}}\ a(g_2(b(x),g_1(b(x))),c(x))$
$\underline{\text{else}}\ a(g_2(b(x),c(x)))\ \underline{\text{fi}}$
$\underline{\text{else}}\ \underline{\text{if}}\ p(x)\ \underline{\text{then}}\ g_1(b(x))\ \underline{\text{else}}\ d(x)\ \underline{\text{fi}}\ \underline{\text{fi}}$

por b2

$= \underline{if}\ p(x)\ \underline{then}\ a(g_2(b(x),\ g_1(b(x))),\ c(x))\ \underline{else}\ d(x)\ \underline{fi}$

por b1

$= \underline{if}\ p(x)\ \underline{then}\ a(f_1(b(x)),\ c(x))\ \underline{else}\ d(x)\ \underline{fi}$

por contração usando (3)

O que demonstra o teorema 2.1 . □

3. MÉTODO DE INDUÇÃO SOBRE UM CONJUNTO BEM FUNDADO

Um conjunto bem fundado /15/ é um conjunto no qual podemos definir uma ordem, de tal modo que não existe nenhuma sequência decrescente infinita segundo essa ordem ou, em outras palavras, que toda sequência decrescente tem mínimo. Essa ordem é também chamada de ordem bem fundada. Podemos empregar conjuntos bem fundados como base para a prova de indução matemática usando o seguinte princípio de indução bem fundada:

> Seja B um conjunto munido de uma ordem bem fundada>. Para provar que P(b) é verdadeiro para todo elemento b de B, devemos provar que P(b) é verdadeiro se b é o mínimo de uma sequência decrescente e provar que P(b) é verdadeiro assumindo que P(b') é verdadeiro onde b' é todo elemento de B tal que b>b'.

Esse método foi introduzido por Floyd em /15/ para provar terminação de programas.

Exemplificaremos esse princípio na prova da consistência da regra III.1. Nos lemas 3.1, 3.2 e no teorema 3.1 que se seguem usaremos a notação f_1 para nos referirmos ao procedimento declarado no consequente da regra.

Lema 3.1

Se para um certo x_o, $f(x_o)$ está definida, os instantâneos em {1} e {3} são S_1 e S_3 respectivamente e na chamada

$f_1(x_o)$, o instantâneo S_4 em {4} é tal que $S_1 \equiv S_4/\{y\}$ então $f_1(x_o)$ está definida, computa o mesmo valor e $S_3 \equiv S_6/\{x,y\}$ onde S_6 é o instantâneo em {6}.

Prova

Vamos provar por indução em x_o sobre a ordem induzida pela computação do procedimento f. Nessa ordem se $x_o > x'_o$ então ocorre uma chamada $f(x'_o)$ na computação de $f(x_o)$, onde $f(x_o)$ está definida. Em particular se $f(x_o)$ está definida então $x_o > a(x_o)$. Esta ordem é bem fundada uma vez que uma sequência infinita nesta ordem corresponderia a uma computação não definida no procedimento f, o que não pode ocorrer pois a ordem só está estabelecida para chamadas definidas de f.

Base da indução: x_o é o menor elemento de uma sequência.

Podemos afirmar neste caso que não há nenhuma chamada recursiva de f na chamada $f(x_o)$ pois se houvesse, x_o não seria o menor elemento da sequência. Daí, podemos garantir que se o controle atingir {2} então $p(x_o)$ será falso (visto que não há desvios de R_2 para fora de R_2). Portanto para esta chamada será executada uma sequência de comandos idêntica a uma chamada com parâmetro x_o do seguinte procedimento:

<u>proc</u> f_2 = ($\underline{t}_1$ x) $\underline{t}_2$:

(1) <u>begin</u> {7}

R_1; p(x); R_3; R_4;

L:{8} <u>end</u>

No consequente da regra a parte <u>then</u>, também nunca será atingida pois $p(x_o)$ é falso.

Portanto para $x=x_o$, $f(x_o)$, $f_1(x_o)$ e $f_2(x_o)$ executam a mesma sequência de comandos se os instantâneos S_1, S_4, S_7 em {1}, {4} e {7} respectivamente são tais que $S_1 \equiv S_7$ e $S_1 \equiv S_4/\{y\}$ uma vez que y não ocorre no corpo de f_1. Isto ga

rante que os instantâneos S_3, S_6 e S_8 de {3}, {6} e {8} são tais que $S_3 \equiv S_6/\{x,y\}$ e $S_3 \equiv S_8$.

Passo da indução: x_o não é o mínimo de nenhuma sequência.

Só precisamos analisar a primeira vez que $p(x_o)$ é verdadeiro pois até que isto aconteça as computações de f e f_1 são idênticas pois $S_1 \equiv S_3/\{y\}$ e y não ocorre no corpo de f_1. Chamemos de α a sequência de comandos executada pelo antecedente (ou pelo consequente) de III.1 até a primeira vez que $p(x_o)$ seja verdadeiro. Quando o controle atingir a parte then executará ${}^{x}R_2[x_o]$; $f(a(x_o))$; go to L;L: terminando a execução de $f(x_o)$. Portanto a sequência executada é:

(12) {1} ${}^{x}\alpha[x_o]$; {2} $p(x_o)$; ${}^{x}R_2[x_o]$; $f(a(x_o))$; go to L;L:{3}

Numa chamada $f_1(x_o)$ que satisfaça as hipóteses deste lema é executado:

$$ {}^{x}\alpha[x_o];\ p(x_o);\ {}^{x}R_2[x_o];\ x:=a(x_o);\ \text{go to } L_1 $$

Mas voltar a L_1 com x valendo $a(x_o)$ é o mesmo que executar f_1 com valor $a(x_o)$, ou seja executarmos a seguinte sequência:

(13) {4} ${}^{x}\alpha[x_o]$; {5} $p(x_o)$; ${}^{x}R_2[x_o]$; $x:=a(x_o)$; $f_1(a(x_o))$ {6}

Para podermos aplicar a hipótese de indução precisamos verificar se os instantâneos S_7 e S_8 associados aos pontos {1} e {4} das chamadas $f(a(x_o))$ e $f_1(a(x_o))$ respectivamente, satisfazem as condições do lema. Observe-se que a menos de y que não está definido em (12) as demais informações são idênticas até imediatamente depois de ${}^{x}R_2[x_o]$. Portanto $S_7 \equiv S_8/\{y\}$. Como $x_o > a(x_o)$ e $f(a(x_o))$ está definida temos que $f_1(a(x_o))$ também está definida, computa o mesmo valor e os instantâneos S_9 e S_{10} válidos em {3} e {6} das cha

madas $f(a(x_o))$ e $f_1(a(x_o))$ respectivamente são tais que $S_9 \equiv S_{10}/\{x,y\}$. Os valores computados nas chamadas $f(x_o)$ e $f'(x_o)$ são iguais pois são os valores computados respectivamente nas chamadas $f(a(x_o))$ e $f_1(a(x_o))$. O instantâneo S_3 válido em {3} de (12) é tal que $S_3 \equiv S_9/\{x\}$ pois em S_3 x vale x_o e em S_9 x vale $a(x_o)$; e os instantâneos S_6 de {6} em (13) é tal que $S_6 \equiv S_{10}/\{x,y\}$. Portanto $S_3 \equiv S_6/\{x,y\}$ o que demonstra o lema.□

Lema 3.2

Se para um certo x_o, $f_1(x_o)$ está definida, S_4 é o instantâneo em {4} nesta chamada e S_1 é o instantâneo em {1} numa chamada de $f(x_o)$ onde $S_1 \equiv S_4/\{y\}$ então $f(x_o)$ está definida.

Prova

Podemos provar por indução sobre a ordem> induzida pela computação do esquema iterativo. Nesta ordem $x_o > a(x_o) > \ldots > a^n(x_o)$ onde $\neg\, p^n(x_o) \wedge \forall i[0 \leq i < n \wedge < p(a^i(x_o))]$. Esta ordem é bem fundada pois só consideramos as sequências decrescentes que satisfaçam a condição acima.

Esta prova pode ser feita de modo análogo à prova do lema 3.1.□

Teorema 3.1

A regra III.1 é consistente.

Prova

Segue dos lemas 3.1 e 3.2.□

4. MÉTODO DA INDUÇÃO COMPUTACIONAL

Para exemplificar o princípio da indução de Morris /25/, também chamado de indução computacional, descrito no capítulo II, vamos empregá-lo para provar a consistência da regra III.24. Essa prova ocorreu no trabalho de Bird /6/.

Para provar que o antecedente é equivalente ao consequente da regra vamos considerar, inicialmente, o seguinte esquema de procedimento:

```
          proc g=(t1 y, t2 k1) void:
            begin t1 x:=y, int k:=k1;
              if p(x) then R1; push(s,x);
(14)                      g(a(x),0);g(b(x), k+1)
                     else while k>0 do k:=k-1;
                                      pop(s,x); R2 od fi
            end
```

onde a variável s é uma variável global ao procedimento g , sendo do tipo stack.

Lema 4.1

A chamada $g(x_o,k_o+1)$ é equivalente à sequência $g(x_o,k_o)$; pop(s,x); R_2.

Prova

Para provar o lema, vamos declarar uma família de esquemas de procedimento $\{g_i\}$, $i \geq 0$ do seguinte modo:

a) Se i=0 então g_o está indefinido para qualquer valor dos parâmetros;

b) Se i>0 então g_i é declarado como:

```
proc g_i = (t_1 y, t_2 k_1) void:
  begin t_1 x:=y ; int k:=k_1;
    if p(x) then R_1; push (s,x);
                      g_i-1(a(x),0);g_i-1(b(x),k+1)
         else while k>0 do k:=-k-1;
                      pop(s,x); R_2 od fi
  end
```

Base da indução: i=0

As chamadas $g_o(x_o,k_o+1)$ e $g_o(x_o,k_o)$ estão ambas indefinidas, portanto são equivalentes.

Passo da indução: i>0

A chamada $g_i(x_o,k_o+1)$ é equivalente por expansão a:

```
begin t_1 x:=x_o, int k:=k_o+1;
  if p(x)  then R_1; push(s,x);
                  g_i-1(a(x_o),0); g_i-1(b(x_o),k_o+2)
         else while k>0 do k:=-k-1; pop(s,x);
                        R_2 od fi
  end
```

que por hipótese de indução é equivalente a:

```
begin t_1 x:=x_o, int k:=k_o+1;
  if p(x) then R_1; push(s,x);
               g_i-1(a(x),0); g_i-1(b(x_o);k_o+1);pop(s,x);R_2
       else while k>0 do k:=-k-1; pop(s,x);
                     R_2 od fi
end
```

Neste último esquema se inicializarmos k com k_o (ao invés de k_o+1) devemos acrescentar após o comando while k>0 do ... od a sequência pop(s,x); R_2, obtendo:

```
begin t1 x:=xo, int k:=ko;
  if  p(x) then R1; push(s,x);
            g_{i-1}(a(x),0); g_{i-1}(b(x),k+1); pop(s,x); R2
       else while k>0 do k:=k-1; pop(s,x); R2 od;
            pop(s,x); R2 fi
end
```

que é equivalente a

```
begin t1 x:=xo, int k:=ko;
  if p(x) then R1; push(s,x);
            g_{i-1}(a(x),0); g_{i-1}(b(x),k+1)
       else while k>0 do k:=k-1; pop(s,x); R2 od fi ;
  pop(s,x); R2
end
```

que é equivalente por contração à sequência abaixo, supondo sem perda de generalidade, que exista uma variável x de tipo t_1 global ao procedimento.

$$g_i(x_o,k_o);\ pop(s,x);\ R_2.\square$$

Teorema 4.1

A chamada $f(x_o)$, onde f é o procedimento declarado no antecedente da regra III.24 é equivalente a $g(x_o,0)$.

Prova

Seja a família $\{f_i\}$ declarada de maneira análoga à família $\{g_i\}$ do lema 4.1.

Base da indução: $i=0$

$f_o(x_o)$ e $g_o(x_o,0)$ estão ambas indefinidas para qualquer valor de x_o.

Passo da indução: $i>0$

A chamada $f_i(x_o)$ é equivalente por expansão a:

```
begin t1 x:=xo;
  if p(x) then R1; f_{i-1}(a(xo)); f_{i-1}(b(xo)); R2 fi
end
```

Como f_{i-1} não tem nenhum efeito sobre a pilha s podemos introduzir os comandos push(s,x) e pop(s,x), depois de R_1 e antes de R_2 respectivamente sem alterarmos a equivalência, chegando a:

```
begin t1 x:=xo;
  if p(x) then R1; push(s,x); f_{i-1}(a(xo)); f_{i-1}(b(xo));
                pop(s,x); R2 fi
end
```

Usando a hipótese de indução temos:

```
begin t1 x:=xo;
  if p(x) then R1; push(s,x); g_{i-1}(a(xo),0); g_{i-1}(b(xo),0);
               pop(s,x); R2 fi
end
```

Pelo lema 4.1, obtemos:

```
begin t1 x:=xo
  if p(x) then R1; push(s,x); gi-1(a(x),0); gi-1(b(xo),1) fi
end
```

Introduzindo-se uma variável local de tipo inteiro k, inicializada com 0, podemos estabelecer o resultado:

```
begin t1 x:=xo, int k:=0;
  if p(x) then R1; push(s,x); gi-1(a(x),0); gi-1(b(x), k+1)
        else while k>0 do k:=k-1;
                          pop(s,x); R2 od fi
end
```

pois quando k=0 o comando while introduzido não é executado nenhuma vez. Esta última forma por contração é equivalente à chamada:

$$g_i(x_o,0).\ \Box$$

Lema 4.2

O procedimento g declarado em (14) é equivalente a:

```
     proc g=(t1 y, int k1) void:
       begin t1 x:=y, int k:=k1, stack s1:=nil;
         L:while p(x) do R1; push(s,x); push(s1, b(x));
                        x:=a(x); go to L od;
(15)       while k>0 do k:=k-1; pop(s,x); R2 od;
           if ¬ empty(s1) then pop(s1,x); k:=1;
                              go to L fi
       end
```

Prova

Inicialmente, podemos aplicar na segunda chamada recursiva do procedimento g em (14) uma versão ligeiramente modificada da regra III.1, onde o antecedente é:

```
proc f=(t1 y) t2:
  begin t1 x:=y;
    R1;
    if p(x) then R2; f(a(x)); go to L
            else R3 fi;
    R4
  end
```

obtendo:

```
proc g=(t1 y, int k1) void:
  begin t1 x:=y, int k:=k1;
    L: if p(x) then R1; push(s,x); g(a(x),0);
                    x:=b(x); k:=k+1; go to L
              else while k>0 do k:=k-1; pop(s,x);
                                   R2 od fi
  end
```

Para eliminarmos, definitivamente, a chamada recursiva que resta podemos empregar uma pilha de acordo com seu uso feito na regra III.2, observando que não há necessidade de empilharmos os valores dos parâmetros (y, k_1) uma vez que estes valores estão armazenados nas variáveis locais (x e k):

```
proc g=(t1 y, int k1) void:
  begin t1 x:=y, int k:=k1, stack s1:=nil;
    L: if p(x) then R1; push(s,x); push(s1,x);
                    push(s1,0); x:=a(x); go to L fi;
       while p(x) do k:=k-1; pop(s,x); R2 od;
       while ¬ empty (s1) do pop(s1,k); pop(s1,x);
                             x:=b(x); k:=k+1;
                             go to L od
  end
```

Neste último esquema podemos efetuar as seguintes transformações que preservam a equivalência:

a) aplicar a transformação T_1 de 1.5 no comando if...fi;

b) eliminar os comandos push(s_1,0), pop(s_1,k) e trocar o comando k:=k+1 por k:=1;

c) trocar o comando while ¬ empty(s_1) do...go to L od; por um comando if ¬ empty(s_1) then...go to L fi, uma vez que o predicado ¬ empty(s) do primeiro só é executado uma única vez, cada vez que se atinge o comando while, devido ao desvio incondicional que ocorre no fim executável da parte do do while.

Obtemos, então, o esquema:

```
proc g=(t1 y, int k1) void:
  begin t1 x:=y, int k:=k1, stack s1:=nil;
    L: while p(x) do R1; push(s,x); push(s1,x);
                     x:=a(x); go to L od;
       while k>0 do k:=k-1; pop(s,x); R2 od;
       if ¬ empty (s1) then pop(s1,x); x:=b(x);
                            k:=1; go to L fi
  end
```

Finalmente, podemos ainda efetuar a transformação a baixo que leva a uma forma ligeiramente mais eficiente que essa última, a saber:

d) ao invés de empilharmos o valor da variável x em s_1 através de push(s_1,x), empilhar diretamente o valor de b(x), através de push(s_1,b(x)), uma vez que o primeiro comando que se segue a pop(s_1,x) é o comando de atribuição x:=b(x). Portanto trocamos push(s_1,x) por push(s_1,b(x)) e eliminamos a atribuição x:=b(x).

Aplicando essa transformação obtemos o esquema (15).

□

Teorema 4.2

A regra III.24 é consistente.

Prova

Inicialmente, afirmamos que o antecedente da regra III.24 é equivalente, pelo teorema 4.1 ao esquema abaixo:

```
proc f=(t1 y) void:
  begin stack s:=nil;
    g(y,0)
  end
```

que por expansão usando (14) é equivalente a:

```
     proc f=(t1 y) void:
       begin t1 x:=y, int k:=0, stack s:=nil, s1:=nil;
         L: while p(x) do R1; push(s,x); push(s1,b(x));
(16)                   x:=a(x); go to L od;
            while k>0 do k:=k-1; pop(s,x); R2 od;
            if ¬ empty (s1) then pop(s1,x); k:=1;
                              go to L fi
       end
```

Observamos, finalmente, que como no corpo do proce

dimento f de (16) os únicos valores atribuídos a k são 0 e 1, o comando while k>0 do k:=k-1...od pode ser trocado pelo comando if k>0 then k:=0...fi, obtendo-se o consequente de III. 24. □

5. MÉTODO DA ASSERÇÃO INTERMITENTE

Segundo a terminologia de Manna e Waldinger /22/usaremos a frase: "alguma vez Q em {i}" para denotar que Q é uma asserção intermitente (cf.II) no ponto {i}, isto é, alguma vez o controle passará por {i} com a asserção Q satisfeita.

Para provarmos que uma regra é consistente pelo método da asserção intermitente devemos provar dois teoremas. O primeiro deles tem como uma das hipóteses a condição de $f(x_o)$ estar definida (que a computação do procedimento resursivo f numa chamada com argumento x_o termina). Esta condição é necessária para garantir que o procedimento iterativo também termine e que retorne o mesmo valor. Além disso este teorema também, estabelece que os efeitos colaterais serão os mesmos. O segundo teorema deve garantir que o procedimento recursivo termina quando o procedimento iterativo assim o faz. Com esses dois teoremas configura-se a equivalência entre os dois esquemas. Para provar os teoremas, enuncia-se e prova-se alguns lemas que descrevem asserções intermitentes referentes a pontos internos do corpo do procedimento iterativo. As provas dos lemas são feitas por indução sobre os elementos de um conjunto bem fundado. Para exemplificar o método, empregamos o mesmo nos teoremas 5.1 e 5.2 que estabelecem a consistência da regra III.4 e nos teoremas 5.3 e 5.4 que provam a consistência da regra III.9. No teorema 5.3 e no lema 5.3 usamos a seguinte notação: se x_o é um valor e s' é uma pilha então x_o.s' representa uma pilha cujo elemento do topo é x_o. Nos lemas e teoremas desta secção denotaremos por f e f_1, os procedimentos declarados respectivamente no antecedente e consequente das regras. Além disso representaremos por $f(x_o)$ o valor computado por f nesta chamada.

Teorema 5.1

Se alguma vez, $x=x_o$ e S_3 é o instantâneo em {3} , $f(x_o)$ está definida e se S_1 e S_2 são respectivamente os instantâneos em {1} e {2} quando $x=x_o$, onde $S_1 \equiv S_3/\{k,y,z\}$ então alguma vez $z=f(x_o)$ e S_6 é o instantâneo em {6} com $S_2 \equiv S_6/\{k,x,y,z\}$.

Lema 5.1

Se alguma vez para algum $i \geq 0$, $x=b^i(x_o)$, $i=k$ e S_4 é o instantâneo em {4} , $f(b^i(x_o))$ está definida e S_1 e S_2 são respectivamente os instantâneos em {1} e {2} quando $x=b^i(x_o)$ onde $S_1 \equiv S_4/\{k,y,z\}$ então alguma vez $z=f(b^i(x_o))$, $k=i$ e S_5 é o instantâneo em {5} onde $S_2 \equiv S_5/\{k,x,y,z\}$.

Prova

Provemos por indução em x_o sobre a ordem > induzida pela computação do antecedente da regra. Vamos assumir indutivamente que o lema vale para $x=b^j(x_o)$ para qualquer j tal que $b^i(x_o) > b^j(x_o)$ e mostraremos que ele é válido para $x = b^i(x_o)$.

Base da indução:

Isto se dá na primeira vez que para um dado $i, p(b^i(x_o))$ é falso, e sendo seguido o ramo <u>else</u>, c(x) é atingido.

No procedimento recursivo é executado:

$$p(b^i(x_o));\ {}^{x}R_2[b^i(x_o)];\ c(b^i(x_o))$$

retornando o valor $c(b^i(x_o))$.

Se alguma vez em {4} $x=b^i(x_o)$ e $k=i$, como $p(b^i(x_o))$ é falso é executado:

$$p(b^i(x_o));\ {}^{x}R_2[b^i(x_o)];\ z:=c(b^i(x_o))$$

e então atinge-se {5} com $z=c(b^i(x_o)) = f(b^i(x_o))$.

Observamos, também, que como foi executada a mesma sequência de comandos, a menos da atribuição à variável z, então se $S_1 \equiv S_4/\{k,y,z\}$ temos $S_2 \equiv S_5/\{k,y,z\}$ (neste caso x assume o mesmo valor $b^i(x_o)$ nos dois instantâneos) onde S_5 é o instantâneo em {5}.

Passo da indução: quando $p(b^i(x_o))$ é verdadeiro.

No procedimento recursivo é executada a seguinte sequência:

$$\{1\}\quad p(b^i(x_o));\ {}^{x}R_1[b^i(x_o)];\ \{7\}\ a(f(b(b^i(x_o))))\ \{2\}$$

retornando o valor $a(f(b^{i+1}(x_o)))$. ($f(b^{i+1}(x_o))$ está definida visto que $b^i(x_o) > b^{i+1}(x_o)$ e por hipótese $f(b^i(x_o))$ está definida). Se alguma vez em {4}, $p(b^i(x_o))$ é verdadeiro é executado:

$$\{4\}\quad p(b^i(x_o));\ {}^{x}R_1[b^i(x_o)];\ \{8\}\ k:=k+1;\ \{9\}\ x:=b(b^i(x_o));$$

voltando a {4} com $x=b^{i+1}(x_o)$ e $k=i+1$, com um instantâneo S_4'.

Como os instantâneos S_1 e S_4 de {1} e {4} respectivamente são tais que $S_1 \equiv S_4/\{k,y,z\}$, temos $S_7 \equiv S_8/\{k,y,z\}$, onde S_7 e S_8 são os instantâneos em {7} e {8} respectivamente, uma vez que k,y e z não ocorrem em p e R_1. O que difere o instantâneo S_9 de {9} do instantâneo S_8 é o valor de k, logo $S_7 \equiv S_9/\{k,y,z\}$. Em seguida nos dois esquemas é executado $b(b^i(x_o))$ e este é atribuído a x, no primeiro caso através da chamada recursiva e no segundo pelo comando de atribuição. Portanto, sendo S_1' o instantâneo válido em {1} da execução da chamada $f(b^{i+1}(x_o))$, temos $S_1' \equiv S_4'/\{k,y,z\}$; chamemos de S_2' o instantâneo em {2} da execução desta última chamada.

Por hipótese de indução alguma vez $z=f(b^{i+1}(x_o))$ e $k=i+1$ em {5} e S_5' é o instantâneo válido onde $S_2' \equiv S_5'/\{k,x,y,z\}$. Como $i \geq 0$ temos $i+1=k \geq 1$. Portanto a condição $k>0$ é satisfeita e

são executados os comandos $k:=k-1$ e $z:=a(z)$ ou seja temos $z = a(f(b^{i+1}(x_o))) = f(b^i(x_o))$ e $k=i$ em {5}, com um novo instantâneo S_5. No esquema recursivo depois da volta da chamada recursiva também é executado $a(f(b^{i+1}(x_o)))$. Portanto $S_2 \equiv S_5/\{k, x,y,z\}$. □

Prova do Teorema 5.1

Suponhamos que alguma vez $x=x_o$ e S_3 é o instantâneo em {3}, $f(x_o)$ está definido e S_1 e S_2 são respectivamente os instantâneos em {1} e {2} quando $x=x_o$ onde $S_1 \equiv S_3/\{k,y,z\}$. No procedimento iterativo no ponto {3} é executado $k:=0$ e atinge-se {4} com todas as hipóteses do lema 5.1 satisfeitas tomando-se $i=k=0$. Portanto pelo lema 5.1 alguma vez $z=f(b^o(x_o))= =f(x_o)$ e $k=i=0$ em {5} com S_5 de instantâneo onde $S_2 \equiv S_5/\{k,x,y, z\}$. Como $k=0$ o controle passa ao ponto {6} com o mesmo instantâneo S_5 e retorna o valor $z=f(x_o)$. □

Teorema 5.2

Se alguma vez $x=x_o$ e S_3 é o instantâneo em {3}, $f_1(x_o)$ está definida, e $S_1 \equiv S_3/\{k,y,z\}$ onde S_1 é o instantâneo em {1} na chamada $f(x_o)$ então esta chamada, $f(x_o)$, está definida.

Lema 5.2

Se alguma vez para $i \geq 0$, $x=b^i(x_o)$, $k=i$ e S_4 é o instantâneo em {4} e a computação no esquema iterativo termina então $f(b^i(x_o))$ está definida onde S_1, o instantâneo válido em {1} nesta chamada, é tal que $S_1 \equiv S_4/\{k,y,z\}$.

Prova

A prova é feita por indução em x_o sobre a ordem > induzida pela computação do esquema iterativo. Nesta ordem $x_o > x_o'$ onde x_o e x_o' são valores sucessivos de x em {4} durante

uma computação que termina do procedimento iterativo. Suponhamos que o lema é válido quando $x=b^j(x_o)$ onde $b^i(x_o)>b^j(x_o)$ e mostraremos que ele se verifica quando $x=b^i(x_o)$.

Base da indução: quando $p(b^i(x_o))$ é falso.

No antecedente é executado:

$p(b^i(x_o))$; ${}^xR_2[b^i(x_o)]$; $c(b^i(x_o))$

o que prova que $f(b^i(x_o))$ está definida.

Passo da indução: quando $p(b^i(x_o))$ é verdadeiro.

No consequente é executado:

{4} $p(b^i(x_o))$; ${}^xR_1[b^i(x_o)]$, {7} $k:=k+1$; $x:=b(b^i(x_o))$

retornando a {4} com $x=b^{i+1}(x_o)$, $k=i+1$ e instantâneo S_4'. No procedimento recursivo é executado:

{1} $p(b^i(x_o))$; ${}^xR_1[b^i(x_o)]$; {8} $a(f(b(b^i(x_o))))$.

Sendo S_1, S_4, S_7, S_8, S_9 os instantâneos em {1}, {4}, {7}, {8} e em {1} da chamada $f(b^{i+1}(x_o))$ respectivamente, podemos afirmar que

$S_7 \equiv S_8/\{k,y,z\}$ pois $S_1 \equiv S_4/\{k,y,z\}$ e foi executada a mesma sequência de comandos.

$S_7 \equiv S_4'/\{k,x\}$ pois os dois comandos que se interpõem entre S_7 e S_4' são atribuições a k e x (x assume o valor $b^{i+1}(x_o)$).

$S_8 \equiv S_9/\{x\}$ pois pela chamada de f, x assume o valor $b^{i+1}(x_o)$.

Portanto $S_4' \equiv S_9/\{k,y,z\}$. Observamos também, que pela ordem definida $b^i(x_o) > b^{i+1}(x_o)$ e como a computação no esquema iterativo de $b^i(x_o)$ termina a computação de $b^{i+1}(x_o)$ também termina. Podemos aplicar a hipótese de indução concluindo que $f(b^{i+1}(x_o))$ está definida, o que demonstra que $f(b^i(x_o))$ está definida. □

Prova do Teorema 5.2

O lema 5.2 implica obviamente no Teorema 5.2. □

Teorema 5.3

Se alguma vez $x=x_o$ e S_3 é o instantâneo em {3}, $f(x_o)$ está definida e se S_1 e S_2 são respectivamente os instantâneos associados a {1} e {2} quando $x=x_o$ onde $S_1 \equiv S_3/\{s,y,z\}$ então alguma vez $z=f(x_o)$ e S_6 é o instantâneo em {6} com $S_2 \equiv S_6/\{s,x,y,z\}$.

Lema 5.3

Se alguma vez $x=x_o$, $s=x_o.s'$ e S_4 é o instantâneo em {4}, $f(x_o)$ está definida e se S_1 e S_2 são respectivamente os instantâneos associados a {1} e {2} quando $x=x_o$ onde $S_1 \equiv S_4/\{s,y,z\}$ então alguma vez $z=f(x_o)$, $s=s'$ e S_5 é o instantâneo em {5} onde $S_2 \equiv S_5/\{s,y,z\}$.

Prova

Provemos por indução em x_o sobre a ordem > induzida pela computação no esquema recursivo. Vamos assumir que o lema vale quando $x=x_1$ e $s=x_1.s_1$ onde $x_o > x_1$ é mostraremos que ele é válido quando $x=x_o$ e $s=x_o.s'$.

Base da indução: quando $p(x_o)$ é falso.

No antecedente é executado:

$$\{1\} \quad p(x_o);\ {}^{x}R_2[x_o];\ d(x_o) \quad \{2\}$$

retornando o valor $d(x_o)$ com um instantâneo S_2 em {2}.

Se alguma vez em {4}, $x=x_o$ e $s=x_o.s'$, devido a $p(x_o)$ ser falso é executado:

$$\{4\} \quad p(x_o);\ {}^{x}R_2[x_o];\ z:=d(x_o);\ pop(x_o.s',x);\ \{5\}$$

atingindo-se {5} com $z=d(x_o)=f(x_o)$, $s=s'$ e instantâneo S_5.

Observa-se que nos dois casos foi executada a mesma sequência de comandos a menos da atribuição à variável z e do procedimento pop. Isto implica que se $S_1 \equiv S_4/\{s,y,z\}$ temos $S_2 \equiv S_5/\{s,x,y,z\}$.

Passo da indução: quando $p(x_o)$ é verdadeiro

No procedimento f é executado:

$$\{1\} \quad p(x_o);\ {}^{x}R_1[x_o];\ \{7\}\ a(f(b(x_o)),c(x_o));\ \{2\}$$

Por hipótese $f(x_o)$ está definida e como $x_o > b(x_o)$, $f(b(x_o))$ também está definida. Logo o valor que retorna é $a(f(b(x_o)),\ c(x_o))$.

Se alguma vez em {4}, $x=x_o$, $s=x_o.s'$ como $p(x_o)$ é verdadeiro é executado:

$$\{4\} \quad p(x_o);\ {}^{x}R_1[x_o];\ \{8\}\ x:=b(x_o);\ \{9\}\ {}^{s}push(s,x_o)[x_o.s']$$

retornando a {4} com $x=b(x_o)$, $s=b(x_o).(x_o.s')$ e instantâneo S_4'.

Sendo S_7, S_8, S_9, S_1' e S_2' os instantâneos em {7}, {8}, {9}, em {1} e {2} da chamada $f(b(x_o))$, respectivamente temos $S_7 \equiv S_8/\{s,y,z\}$ pois $S_1 \equiv S_4/\{s,y,z\}$ e foi executada nos dois casos a mesma sequência $p(x_o);\ {}^{x}R_1[x_o]$. Em seguida em ambos é executado $b(x_o)$ que é atribuído a x no primeiro caso pela chamada recursiva e no segundo pelo comando de atribui-

ção. Portanto $S_1' \equiv S_9/\{s,y,z\}$. O instantâneo S_4' difere de S_9 pelo valor de s. Logo $S_1' \equiv S_9/\{s,y,z\}$.

Por hipótese de indução alguma vez em {5} $z=f(b(x_o))$, $s=x_o.s'$ e S_5' é o instantâneo em {5} com $S_2' \equiv S_5/\{s,x,y,z\}$. Como o predicado $\neg$ empty $(x_o.s')$ é verdadeiro é executado ^{s}pop(s,x)$[x_o.s']$ fazendo $x=x_o$ e s=s'; em seguida é executado $z:=a(z,c(x_o))$ ou seja temos $z=a(f(b(x_o)),c(x_o))=f(x_o)$ em {5} com um instantâneo S_5. No esquema recursivo depois da volta da chamada recursiva também é executado $a(f(b(x_o)),\ c(x_o))$. Portanto $S_2 \equiv S_5/\{k,x,y,z\}$. □

Teorema 5.4

Se alguma vez $x=x_o$ e S_3 é o instantâneo em {3} , $f_1(x_o)$ está definida e $S_1 \equiv S_3/\{s,y,z\}$ onde S_1 é o instantâneo em {1} na chamada $f(x_o)$ então esta chamada está definida.

Lema 5.4

Se alguma vez $x=x_o$, $s=x_o.s'$ e S_4 é o instantâneo em {4} e a computação no esquema iterativo termina então $f(x_o)$ está definida onde S_1, o instantâneo válido em {1} nesta chamada, é tal que $S_1 \equiv S_4/\{k,y,z\}$.

Prova

A prova pode ser feita de modo análogo à prova do lema 5.2, considerando a ordem > induzida pela computação do esquema iterativo do seguinte modo: $(x_1\ s_1) > (x_2\ s_2)$ onde x_1 e x_2 são valores sucessivos de x e {4} e s_1 e s_2 são valores sucessivos de s em {4} durante uma computação que termine do esquema iterativo. □

Prova do Teorema 5.4

O lema 5.4 implica obviamente no teorema 5.4. □

6. ARITMETIZAÇÃO DO CONTROLE

É possível efetuar-se a aritmetização do controle em certos esquemas de procedimento recursivo em que existe um parâmetro que funciona como um contador controlando a recursão e a execução de sequências de comandos, $R_1, R_2, \ldots, R_k$. Para esses esquemas é possível determinar um esquema iterativo que efetua a mesma execução das sequências R_1, $R_2, \ldots,$ R_k definindo certas funções injetoras com as quais se obtem todas as informações relevantes para determinar aquelas execuções.

Para exemplificar vamos desenvolver e provar a regra III.28. Nessa regra podemos aritmetizar o controle visto que no antecedente da mesma a recursão controla a execução dos meta-símbolos R_j, $0 \le j \le k+1$ que representam sequências de comandos. Para saber quantos R_j são executados, vamos introduzir nesse esquema um contador inteiro, global i, que não <u>ocorre livre</u> (isto é, se ocorrer é declarado localmente) em R_j, $0 \le j \le k+1$, que será inicializado por zero e será incrementado de 1, antes da execução de cada R_j, $1 \le j \le k-1$. Obtemos, então, o esquema:

```
begin int i:=0;
  proc f=(int x) void:
    if x>0 then R_o;
              {0}f(x-1);i:=i+1;{1}R_1;f(x-1);...;i:=i+1;
(17)          {k-1}R_{k-1};f(x-1);
                 R_k
           else R_{k+1} fi
end
```

Nos lemas que se seguem vamos supor que no momento de uma chamada f(x) o valor corrente de i é i_o.

Lema 6.1

Cada chamada de $f(x)$ de (17), com $x \geq 0$, aumenta o valor de i de $k^x - 1$.

Prova

Vamos provar por indução sobre o conjunto dos naturais, com a ordem usual.

Base da indução: $x=0$

Na chamada $f(0)$ nada é executado. Portanto o valor de i depois da execução é $i=i_o=i_o+2^o-1$.

Passo de indução: $x>0$

A chamada $f(x)$ é equivalente por expansão a:

$$R_o; f(x-1); i:=i+1; R_1; f(x-1); \ldots; i:=i+1; R_{k-1}; f(x-1); R_k$$

Por hipótese de indução cada chamada $f(x-1)$ aumenta o valor de i de $k^{x-1}-1$. Portanto como temos k chamadas de $f(x-1)$ i será aumentado de $k.(k^{x-1}-1)$ e, ainda, pela execução do comando $i:=i+1$, $(k-1)$ vezes temos depois de R_k, $i=i_o+k-1+$ $+k.(k^{x-1}-1)=i_o+k^x-1$. □

Corolário 6.1

Se antes de uma chamada $f(x_o)$ para $x_o \geq 0$ o valor de i é 0 então depois da chamada $f(x_o)$ o valor de i é $k^{x_o}-1$.

O próximo lema estabelece a relação que existe entre os valores correntes de x, i e j no momento em que se vai executar um R_j, $1 \leq j \leq k-1$.

Lema 6.2

Na execução de uma chamada f(x), em cada ponto {j} de (17), para $1 \leq j \leq k-1$, $i=i_o+j.k^{x-1}$.

Prova

Vamos provar por indução em j.

Base da indução: j=1

Pelo lema 6.1 depois da execução da primeira chamada da f(x-1) o valor de i é $i_o+k^{x-1}-1$; depois da atribuição o valor de i é aumentado de 1. Portanto em {1}, $i=i_o+k^{x-1}-1+1= =i_o+k^{x-1}$.

Passo da indução: $1<j \leq k-1$

Consideremos a sequência

$$\{j-1\}R_{j-1};\ f(x-1);\ i:=i+1;\ \{j\}\ R_j$$

Por hipótese de indução em {j-1} o valor de i é $i_o+(j-1).k^{x-1}$. Depois da chamada f(x-1), pelo lema 6.1, $i = = i_o+(j-1).k^{x-1}+k^{x-1}-1$. Portanto em {j}, $i=i_o+(j-1).k^{x-1}+k^{x-1}- -1+1=i_o+j.k^{x-1}$. □

Lema 6.3

(a) Em {0} de (17) é verdadeira a condição:

$$C_1[x]:\ \exists\ n \in \mathbb{N}:\ i=n.k^{x-1}$$

(b) Em {j} , para $1 \leq j \leq k-1$ é verdadeira a condição:

$$C_2[x]:\ C_1[x] \wedge k \not| n \wedge j=n \bmod k$$

onde $k \not| p$ denota que k não divide p.

Prova

Vamos provar por indução em x:

Base da indução: $x=x_o$ (isto é, numa chamada externa de f com argumento x_o para a qual $i_o=0$).

Por expansão $f(x_o)$ é equivalente:

$$^{x}(R_o;\{0\}f(x-1);i:=i+1;\{1\}R_1;f(x-1);\ldots;i:=i+1;\{j\}R_j;f(x-1);\ldots;$$
$$f(x-1);i:=i-1;\{k-1\}R_{k-1};f(x-1);R_k)[x_o]$$

a) Em $\{0\}$ $i=0=0.k^{x_o-1}$ o que mostra que $C_1[x_o]$ é verdadeira tomando-se n=0

b) Em cada $\{j\}$, para $1 \le j \le k-1$, pelo lema 6.2 $i= i_o+ j.k^{x_o-1}=j.k^{x_o-1}$. Portanto tomando-se n=j temos $C_1[x_o]$ verdadeira. Verifica-se, também, que $k \not| n$ pois $n=j<k$ e $j=n \bmod k$ pois n=j. Temos, então, que a condição $C_2[x_o]$ é verificada.

Passo da indução: $x=x_1$ com $0<x_1<x_o$ (isto é, na execução de $f(x_1+1)$ uma chamada de f com argumento x_1)

Por hipótese de indução $C_1[x_1+1]$ é verdadeiro antes de $^{x}R_j[x_1+1]$, para $1 \le j \le k-1$, e como R_j não altera i e x temos que $C_1[x_1+1]$ é verdadeira antes das chamadas $f(x_1)$ que ocorrem na execução de $f(x_1+1)$. Portanto se antes da chamada $f(x_1)$, $\exists$ $n' \in \mathbb{N}$: $i=n'.k^{x_1}$ temos $i_o=n'.k^{x_1}$. A chamada $f(x_1)$ por expansão é equivalente a:

$$^{x}(R_o;\{0\}f(x-1);i:=i+1;\{1\}R_1;f(x-1);\ldots;i:=i+1;\{j\}R_j;f(x-1);\ldots;$$
$$f(x-1);i:=i-1;\{k-1\}R_{k-1};f(x-1);R_k)[x_1]$$

a) Como R_o não altera o valor de x e i, em {0} vale $C_1[x_1+1]$, isto é, $\exists n' \in \mathbb{N} : i=n'.k^{x_1}$ o que demonstra que $C_1[x_1]$ é verdadeiro para $n=n'.k$.

b) Em cada {j} para $1 \le j \le k-1$, pelo lema 6.2 temos $i=i_o+j.k^{x_1-1} = n'.k^{x_1}+j.k^{x_1-1} = (k.n'+j).k^{x_1-1}$, o que verifica $C_1[x_1]$ para $n=k.n'+j$. Como $k \nmid (k.n'+j)$ pois $j<k$ e $(k.n'+j) \bmod k=j$ temos que $C_2[x_1]$ é valida para $n=k.n'+j$. □

Pelos últimos tres lemas que acabamos de provar, estabelecemos que numa chamada $f(x_o)$ executa-se $k^{x_o}-1$ $R_j^{'}s$, com $1 \le j \le k-1$; se ${}^{x}R_j[x_1]$ é a i-ésima sequência executada podemos obter os valores de j e x_1 a partir da função:

$$g: i \to (j, x_1) \text{ onde } \exists n \in \mathbb{N} : i=n.k^{x_1-1} \wedge k \nmid n \wedge j=n \bmod k.$$

Podemos definir um procedimento g que computa esta função da seguinte maneira:

```
proc g (int i, ref int j, m) void:
  begin int n:=i;
(18)     m:=1;
         while n mod k=0 do (m,n):=(m+1,n÷k) od;
         j:=n mod k
  end
```

<u>Definição 6.1</u>

Seja ${}_{x_o}R_o$ sequência ${}^{x}R_o[x_o]; {}^{x}R_o[x_o-1]; \ldots ; {}^{x}R_o[1]$ para $x_o>0$ e ${}_oR_o$ a sequência vazia.

<u>Definição 6.2</u>

Seja ${}_{x_o}R_k$ a sequência ${}^{x}R_k[1]; {}^{x}R_k[2]; \ldots ; {}^{x}R_k[x_o]$ para $x_o>0$ e ${}_oR_k$ a sequência vazia.

Definição 6.3

Vamos definir indutivamente $G[x_o]$ para $x_o \geq 0$ como sendo:

i) ${}^{x}R_{k+1}[0]$ se $x_o=0$

ii) $G[x_o-1]; {}_{x_o-1}R_k;$

${}^{x}(R_1; f(x-1); R_2; \ldots; f(x-1); R_{k-1})[x_o]$

${}_{x_o-1}R_o; G[x_o-1]$

Nos lemas que se seguem estamos nos referindo ao antecedente da regra III.28.

Lema 6.4

Na chamada $f(x_o)$ executa-se

$${}_{x_o}R_o; G[x_o]; {}_{x_o}R_k$$

Prova

Indução em x_o.

Base da indução: $x_o=0$

A chamada $f(0)$ é equivalente por expansão a ${}^{x}R_{k+1}[0]$. Portanto, pelas definições 6.1, 6.2, 6.3 $f(0)$ é equivalente a ${}_oR_o; G[0]; {}_oR_k$.

Passo da indução: $x_o>0$

A chamada $f(x_o)$ por expansão é equivalente a:

$${}^{x}(R_o; f(x-1); R_1; f(x-1); \ldots; R_{k-1}; f(x-1); R_k)[x_o]$$

Aplicando-se a hipótese de indução na primeira e na última chamada de $f(x_o-1)$ temos:

$$^{x}R_o[x_o]; {}_{x_o-1}R_o; G[x_o-1]; {}_{x_o-1}R_k; {}^{x}(R_1; f(x-1); \ldots; R_{k-1})[x_o];$$

$$_{x_o-1}R_o; G[x_o-1]; {}_{x_o-1}R_k; {}^{x}R_k[x_o]$$

que pelas definições 6.1 e 6.2 é equivalente a:

$$_{x_o-1}R_o; G[x_o-1]; {}_{x_o-1}R_k; {}^{x}(R_1; f(x-1); \ldots; R_{k-1})[x_o];$$

$$_{x_o-1}R_o; G[x_o-1]; {}_{x_o}R_k$$

que é pela definição 6.2 equivalente a

$$_{x_o-1}R_o; G[x_o]; {}_{x_o-1}R_k \;. \;\square$$

<u>Lema 6.5</u>

Imediatamente antes de executarmos ${}^{x}R_j[x_o]$, executa-se ${}_{x_o-1}R_k$, e imediatamente após, executa-se ${}_{x_o-1}R_o$.

<u>Prova</u>

Antes e depois de ${}^{x}R_j[x_o]$, executa-se uma chamada de $f(x_o-1)$:

$$\ldots f(x_o-1);\; {}^{x}R_j[x_o];\; f(x_o-1); \ldots$$

Portanto pelo lema 6.4 temos:

$$\ldots {}_{x_o-1}R_o; G[x_o-1]; {}_{x_o-1}R_k; {}^{x}R_j[x_o]; {}_{x_o-1}R_o; G[x_o-1]; {}_{x_o-1}R_k; \ldots$$

o que demonstra o lema. □

20004

Lema 6.6

A primeira sequência executada em $G[x_1]$ para $x_1 \geq 0$ é ${}^{x}R_{k+1}[0]$.

Prova

Indução em x_o.

Base da indução $x_1=0$

Óbvio, pela definição 6.3.

Passo da indução:

Pela definição 6.3 a primeira sequência executada de $G[x_1]$ é a primeira sequência executada em $G[x_1-1]$. Por hipótese de indução a primeira sequência executada em $G[x_1-1]$ é ${}^{x}R_{k+1}[0]$. Logo ${}^{x}R_{k+1}[0]$ é a primeira sequência executada em $G[x_1]$. □

Definição 6.4

Uma sequência ${}_{x_o}R_o$ é completa se imediatamente antes de sua execução não é executado ${}^{x}R_o[x_o+1]$.

Lema 6.7

Quando da execução de uma chamada $f(x_o)$, depois de execução de cada sequência completa ${}_{x}R_o$, para $0 \leq x \leq x_o$, executa-se ${}^{x}R_{k+1}[0]$.

Prova

Indução em x_o.

Base da indução: $x_o=0$

Pelo lema 6.4, $f(0)$ é equivalente a ${}_{o}R_o$; $G[0]$; ${}_{o}R_k$ que pela definição 6.3 é equivalente a ${}_{o}R_o$; ${}^{x}R_{k+1}[0]$; ${}_{o}R_k$ o

que verifica a tese.

Passo da indução: $x_o>0$

Pelo lema 6.4 a chamada $f(x_o)$ é equivalente a ${}_{x_o}R_o$; $G[x_o]$; ${}_{x_o}R_k$ que pela definição 6.3 é equivalente a:

$${}_{x_o}R_o;\ G[x_o-1];\ {}_{x_o-1}R_k;$$

(19) $$^{x}(R_1;f(x-1);R_2;\ldots;f(x-1);R_{k-1})\ [x_o]$$

$${}_{x_o-1}R_o;\ G[x_o-1];\ {}_{x_o}R_k$$

As sequências completas ${}_{x_o}R_o$ e ${}_{x_o-1}R_o$ que aparecem em (19) são seguidas por $G[x_o-1]$. Portanto a primeira sequência executada depois de ${}_{x_o}R_o$ e ${}_{x_o-1}R_o$ é a primeira sequência executada de $G[x_o-1]$ que, pelo lema 6.6, é ${}^{x}R_{k+1}[0]$. Por hipótese de indução, depois de cada sequência completa ${}_{x_1}R_o$ para $0\le x_1<x_o$ que ocorre na execução das chamadas $f(x_o-1)$ de (19) executa-se ${}^{x}R_{k+1}[0]$. □

Examinemos, agora, o consequente da regra III.28 , omitindo a declaração do procedimento g declarado em (18).

```
         proc f=(int y) void:
           begin int i,j,m,x;
(i)          for x from y by -1 to 1 do R_o od;
(ii)         x:=0;R_k+1;
(iii)        for i to k↑j-1
(iii.1)       do  g(i,j,m);
(iii.2)           for x from 1 by 1 to m-1 do R_k od;
                  x:=n;
(iii.3)           case j in R_1,R_2,...,R_k-1 esac;
(iii.4)           for x from m-1 by -1 to 1 do R_o od;
(iii.5)           x:=0; R_k+1 od;
(iv)         for x from 1 by 1 to y do R_k od;
         end
```

Teorema 6.1

A regra III.28 é consistente.

Prova

A tabela que se segue justifica o teorema:

Lema 6.1 - (iii)
Lema 6.3 - (iii.1), (iii.3)
Lema 6.4 - (i), (iv)
Lema 6.5 - (iii.2), (iii.4)
Lema 6.7 - (ii), (iii.5) . □

BIBLIOGRAFIA

/1/ AUSLANDER, M.A. & STRONG JR., H.R. Systematic recursion removal. CACM, 21 (2):127-134, fev 1978.

/2/ BAUER, F.L. Programming as an evolucionary process. In: INT. CONF. ON SOFTWARE ENGINEERING, 2, San Francisco, 1976, p. 223-234.

/3/ BAUER, F.L.; BROY, M.;GNATZ, R.; HESSE, W.;BRÜCKNER,B.K. Notes on the project CIP: towards a wide spectrum langua ge to support program development by transformation. TUM-INFO, 7722, jul 1977.

/4/ BAUER, F.L.; PARTSCH, H.; PEPPER, P.; WÖSSNER, H. Notes on the project CIP: outline of a transformation system. TUM-INFO, 7729, jun 1977.

/5/ BAUER, F.L.; PARTSCH, H.; PEPPER, P.; WÖSSNER, H. Techni ques for program development: software engineering techni ques. Infotech State of the Art Report, New Series,1977.

/6/ BIRD, R.S. Notes on recursion elimitation. CACM, 20 (6): 434-439, jun 1977.

/7/ BROY, M. Program development: for Steinhaus type permuta tion generating programs. TUM-INFO, 7701, jun 1977.

/8/ BROY, M. Program development: the Ackermann function as an example. TUM-INFO, 7716, jun 1977.

/9/ BURSTALL, R.M. Program proving as hand simulation with a little induction. In: IFIP CONGRESS 74. Proceedingns. Amsterdan, North-Holland, 1974, p. 308-312.

/10/ BURSTALL, R.M. & DARLINGTON, J. A transformation system for developing recursive programs. JACM, 24 (1) : 44-67, jan 1977.

/11/ COOPER, D.C. The equivalence of certain computations . Computer Journal, 9:45-52, 1966.

/12/ DARLINGTON, J. Applications of program transformation to program synthesis. In: SYMPOSIUM ON PROVING AND IMPROVING PROGRAMS, Arc-et-Sennes, 1975. Proceedings, p. 133-144.

/13/ DARLINGTON, J. & BURSTALL, R.M. A system which automa tically improves programs. Acta Informatica, 6:41-60,1976.

/14/ DIJKSTRA, E.W. A simple axiomatic basis for programming languages constructs. Report EWD, 372-0, 1973.

/15/ FLOYD, R.W. Assigning meaning to programs. In: SYMPOSIUM IN APPLIED MATEMATICS, New York, 1966.Mathematical aspects of computer science, ed. Schwartz, J.T., Providence, AMS, 1967, p. 19-32.

/16/ GNATZ, R. & PEPPER, P. Fusc: an example in program deve- lopment. TUM-INFO - 7711, mai 1977.

/17/ HUET, G. & LANG, B. Proving and applyng program transforma tions expressed with second-order patterns. Acta Informa- tica, 11:31-55, 1978.

/18/ KNUTH, D.E. Structured programming with go to statements. Computing Surveys, 6 (4):261-301, dec 1974.

/19/ KRÖGER, H. Bemerkungen zur Auflösung von Rekursionen. In: SEMINAR ÜBER METHODIK DES PROGRAMMIERENS, München,TUM, 1974, p. 45-62 (TUM. ABTEILUNG MATHEMATIK).

/20/ LINDSAY, C.H. & VAN DER MEULEN, S.G. Informal introduction to ALGOL 68. Revised edition. Amesterdan,North-Holland , 1977.

/21/ MANNA, Z. Mathematical theory of computation, New York , McGraw-Hill, 1974.

/22/ MANNA, Z. & WALDINGER, R. Is "sometime" sometimes better then "always"? CACM, 21 (2):159-172, fev 1978.

/23/ McCarthy, J. Towards a mathematical science of compution. In: IFIP CONGRESS, Munich, 1962. Proceedings. Amsterdan, North-Holland, 1963, p. 21-28.

/24/ McCarthy, J. A basis for a Mathematical science of computION. In: BRAFFORT, P. & HIRSHBERG, D., ed. Computer programming and formal systems. Amsterdan,North-Holland, 1963, p. 33-70.

/25/ MORRIS JR., J. Another recursion principle.CACM, 14 (5): 351-354, mai 1971.

/26/ PARTSCH, H. & PEPPER, P. A family of rules for recursion removal related to towers of Hanoi problem. Inf. Proc. Letters, 5 (6):174-177, 1976.

/27/ PARTSCH, H. & PEPPER, P. Program transformation on different levels of programming. TUM-INFO, 7715, jun 1977.

/28/ PATERSON, M.S. & HEWITT, C.E. Comparative schematology, In: CONFERENCE ON CONCURRENT SYSTEMS AND PARALLEL COMPUTATION. Woods Hole, Massachusetts, 1970. Records of the project MAC, p. 119-127.

/29/ RICE, H.G. Recursion and iteration. CACM, 8 (2):114-115, fev 1965 .

/30/ STRONG JR., H.R. Translating equations into flow charts. JCSS, 5:254-285, 1971.

/31/ STEINBRÜGGEN, R. Equivalent recursive definitions of certain number theoretical functions. TUM-INFO, 7714,jun 1977.

/32/ WALTER, S.A. & STRONG JR., H. R. Characterizations of flowchartable recursions. JCSS, 7: 404-407, 1973.

/33/ WÖSSNER, H. Rekursionauflösung für gewisse Prozedur-Klassen. In: SEMINAR ÜBER METHODIK DES PROGRAMMIERENS, München , TUM, 1974, p. 69-81 (TUM. ABTEILUNG MATHEMATIK).

/34/ WÖSSNER, H. Transformation into repetitive form. In: INTERNATIONAL SUMMER SCHOOL ON PROGRAM CONSTRUCTION, Munich, 1978.