

IMPLEMENTAÇÃO DE INTERPRETADORES PROLOG COMBINANDO EXPANSÃO EM PROFUNDIDADE COM EXPANSÃO EM AMPLITUDE

Marco Antonio Casanova* e Maria Emilia M. T. Walter**

*Centro Científico da IBM

**Departamento de Matemática da UnB e DHS/DITEC/SERPRO

SUMÁRIO

Este trabalho discute um algoritmo para interpretação de programas Prolog que utiliza a técnica de compartilhamento de estruturas e constrói árvores de refutação através de uma estratégia mista que combina expansão em profundidade com expansão em amplitude. A apresentação é acompanhada de uma análise informal do custo adicional de tempo e espaço incorrido por esta estratégia mista, permitindo uma melhor avaliação dos interpretadores Prolog que utilizam apenas expansão em profundidade.

1. INTRODUÇÃO

Dado um programa Prolog puro P e uma consulta Q , os interpretadores usuais para Prolog expandem a árvore de refutação para P e Q em profundidade, utilizando as cláusulas do programa em uma ordem fixada (a ordem com que foram listadas), até encontrar um ramo de sucesso, isto é, um ramo terminando com um nó rotulado com a cláusula vazia. Esta estratégia de expansão leva a implementações eficientes, mas não garante que um ramo de sucesso seja encontrado, caso exista algum, mesmo admitindo o uso de regras arbitrárias para seleção de literais (Lloyd [1984], sec.10).

Este trabalho explora então a construção de interpretadores que constroem árvores de refutação através de uma estratégia mista que combina expansão em profundidade com expansão em amplitude. A apresentação é acompanhada de uma análise informal do custo adicional de tempo e espaço incorrido por esta estratégia mista, permitindo uma melhor avaliação dos interpretadores Prolog que utilizam apenas expansão em profundidade. O ponto básico da discussão consiste em observar como a técnica de compartilhamento de estruturas pode ser utilizada para representar mais de um ramo da árvore de refutação ao mesmo tempo.

Este trabalho está organizado da seguinte forma. A seção 2 apresenta uma abstração do algoritmo padrão para interpretação de programas Prolog puros, resumindo as idéias principais de van Emden [1984] e Bruynooghe [1982]. Já a técnica descrita na seção 3 para implementação de estratégias mistas é nova e pode ser comparada favoravelmente com a técnica descrita em

Warren [1984]. A seção 4 contém observações sobre o desempenho do algoritmo.

2. O ALGORITMO BÁSICO

O algoritmo para interpretação de programas Prolog, chamado daqui em diante de Algoritmo Básico, constrói a árvore de refutação, para um programa P e uma consulta Q, em profundidade até encontrar um ramo de sucesso. O algoritmo possui três características importantes, discutidas resumidamente a seguir.

A construção da árvore em profundidade permite representar apenas um ramo da árvore de refutação de cada vez através de uma pilha, onde cada elemento acrescentado à pilha corresponde ao resolvente de um nó da árvore de refutação e uma das cláusulas do programa.

Segundo, a seleção do literal mais à esquerda de cada nó da árvore de refutação permite representar cada resolvente apenas pelo seu primeiro literal e por certos campos auxiliares dos elementos da pilha.

Terceiro, o algoritmo incorpora a técnica de compartilhamento de estruturas descrita em Boyer, Moore [1972] que armazena em cada elemento da pilha somente uma representação das unificações, indicando no programa, sempre que possível, as diferentes instâncias dos termos envolvidos numa particular unificação.

Assim, a partir do elemento acrescentado à pilha, podemos recuperar o nó que está sendo criado na árvore e, portanto, cada elemento da pilha representa um nó da árvore de refutação.

Cada elemento E da pilha tem os seguintes campos:

LIT e PROC representam, respectivamente, o literal que está sendo resolvido e o literal complementar ao que está sendo resolvido, pertencente a uma cláusula do programa.

ALIT aponta para o elemento da pilha que contém o ambiente onde as variáveis existentes em LIT podem ser encontradas.

AMB armazena as substituições de variáveis resultantes da unificação que gerou o resolvente representado por E.

REFAZ armazena alterações em ambientes de elementos anteriores a E provocadas pela unificação representada por E. Esta lista é fundamental em retrocessos, para colocar a pilha no estado anterior à criação de E.

A unificação ocorre entre o literal indicado por LIT, com as substituições indicadas no campo AMB do elemento apontado por ALIT, e o literal representado por PROC. A

unificação resulta em um novo ambiente, armazenado no campo AMB, para as variáveis da cláusula do programa indicada por PROC e em modificações em ambientes anteriores, armazenadas no campo REFAZ.

Descreveremos, a seguir, o Algoritmo Básico.

```
programa ( P , Q ) :
/*
    P : programa Prolog
    Q : consulta Prolog
*/
begin
    inicializa;

    A : l := selecionaliteral;

    if l = nil
    then
        PARE INDICANDO SUCESSO;

    B : c := selecionacláusula( l );

    if c <> nil
    then
        begin
            criaelemento( l,c );
            go to A
        end;

    C : if pilha só contém um elemento
    then
        PARE INDICANDO FRACASSO
    else
        begin
            retrocesso;
            go to B
        end;

end;
```

rotina inicializa;

Coloca como primeiro elemento da pilha uma representação de Q.

rotina selecionaliteral;

Retorna o primeiro literal não resolvido das cláusulas do programa apontadas por elementos da pilha. Se não houver nenhum literal, retorna nil, indicando que a cláusula vazia foi gerada.

rotina selecionaciáusula (l);

Retorna um apontador para a cláusula seguinte do programa cuja cabeça unifica com l. Se não houver esta cláusula, retorna nil.

rotina criaelemento(l, c);

Insere um novo elemento na pilha, que representa a unificação de l com a cabeça de c.

rotina retrocesso;

Retira o topo da pilha, utilizando a lista REFAZ para colocar a pilha no estado em que se encontrava antes do topo atual ter sido gerado, e atualiza l.

3. O ALGORITMO DE EXPANSÃO MISTA

O Algoritmo Básico apresentado na seção anterior expande uma árvore de refutação em profundidade, representando apenas um ramo da pilha. Porém, conforme mencionado na introdução, esta estratégia de expansão não necessariamente localiza um ramo de sucesso, caso exista algum, pois poderá encontrar um ramo infinito antes de expandir um ramo de sucesso.

Uma forma simples de contornar este problema consiste em combinar expansão em profundidade com expansão em amplitude. Mais precisamente, os ramos da árvore de refutação parcialmente expandida agora pertencerão a três categorias: ramos de sucesso, terminando em um nó rotulado com a cláusula vazia, ramos de fracasso, terminando em um nó não rotulado com a cláusula vazia e que não possui filhos, e ramos incompletos, terminando em um nó não rotulado com a cláusula vazia e que possui filhos. Dadas duas constantes k e K, uma estratégia de expansão mista seria, por exemplo, expandir a árvore em profundidade até que cada ramo: tenha comprimento k, ou seja um ramo de fracasso de comprimento menor ou igual a k ou seja um ramo de sucesso de comprimento menor ou igual a k. Se nenhum ramo de sucesso de comprimento menor ou igual a k for encontrado, a estratégia aumentaria o limite de expansão dos ramos para 2k e assim sucessivamente até o limite K. Naturalmente, outras funções mais sofisticadas para limitar o comprimento dos ramos poderiam ser implementadas.

Esta estratégia mista de expansão pode ser implementada de duas formas. A primeira alternativa seria simplesmente construir toda a árvore novamente após cada incremento de comprimento máximo. Isto poderia ser feito modificando o Algoritmo Básico para limitar o comprimento dos ramos incompletos. Esta alternativa é computacionalmente ineficiente pois força a reconstrução de toda a árvore já expandida a cada novo incremento do comprimento máximo.

A segunda alternativa seria armazenar simultaneamente todos os ramos incompletos. Se o comprimento fosse aumentado, bastaria expandir as folhas. Esta segunda alternativa será explorada detalhadamente nesta seção, resultando em um algoritmo chamado de Algoritmo de Expansão Mista.

O Algoritmo de Expansão Mista armazena a árvore de refutação através de uma combinação da representação de árvores genéricas por árvores binárias (Horowitz e Sahni [1976] , sec.5.6) com a técnica de compartilhamento de estruturas. Assim, a árvore passará a ser representada por uma multilista cujos elementos contêm, além dos campos descritos na seção 2, quatro novos campos:

PAI ponteiro para o elemento associado ao pai do nó associado a E;

FILHO ponteiro para o elemento associado ao filho mais à esquerda do nó associado a E;

IRMÃO ponteiro para o elemento associado ao irmão à direita do nó associado a E;

EXP será FALSO se o elemento corresponde a um nó cujos filhos ainda não foram expandidos, e VERDADEIRO em caso contrário.

O ponto fundamental do Algoritmo de Expansão Mista consiste na utilização da lista REFAZ de cada elemento para representar vários ramos incompletos de forma correta. Mais precisamente, o algoritmo utilizará a lista REFAZ para desfazer as modificações nos ambientes dos elementos associados aos ancestrais de um nó N, quando houver retrocesso para o pai de N, tanto no caso do elemento que o representa ser retirado da multilista por retrocesso, quanto no caso do retrocesso acontecer apenas para expansão de um novo ramo. Conversamente, quando um ramo for novamente percorrido, as substituições armazenadas nas listas REFAZ dos elementos associados a seus nós serão usadas para refazer os ambientes. Assim, as listas REFAZ permitirão representar vários ramos na multilista utilizando compartilhamento de estruturas. O exemplo abaixo ilustra este ponto.

Considere o seguinte programa Prolog P:

```
1. pai(a,b) <-  
2. pai(a,d) <-  
3. pai(b,c) <-  
4. anc(X,Y) <- pai(X,Y)  
5. anc(U,W) <- pai(U,V) & anc(V,W)
```

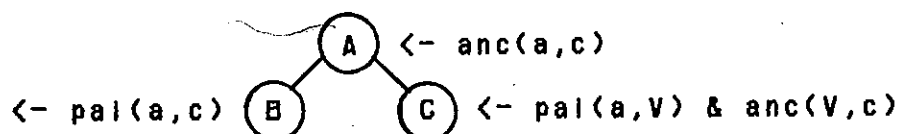
e a seguinte consulta Q:

```
<- anc(a,c)
```

Neste programa, as letras minúsculas indicam constantes e as maiúsculas representam variáveis.

Desenvolveremos uma expansão da árvore de refutação para P e Q utilizando a estratégia mista com $k=1$ e $K=10$. Para facilitar o entendimento do algoritmo, apresentaremos apenas os campos AMB, REFAZ, PAI, FILHO, IRMÃO e EXP de cada elemento da multilista, indicando também o nó associado da árvore, que será denotado por NA.

A primeira expansão da árvore será

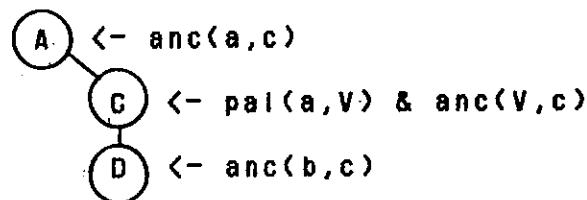


O nó A representa a consulta Q. O nó B foi derivado utilizando-se a cláusula 4 e substituindo-se as variáveis X e Y por a e c, respectivamente. O nó C foi derivado utilizando-se a cláusula 5 e substituindo-se as variáveis U e W por a e c, respectivamente.

A configuração da multilista neste ponto será:

	AMB	REFAZ	PAI	FILHO	IRMÃO	EXP	NA
1.	∅	∅	-	2	-	V	A
2.	{X/a, Y/c}	∅	1	-	3	F	B
3.	{U/a, V/V, W/c}	∅	1	-	-	F	C

Como não há mais nenhum ramo de comprimento 1 e a cadeia vazia não foi derivada, a árvore deverá ser expandida novamente até a altura $2k=2$.



Como o campo EXP do elemento 2 é FALSO, o algoritmo tenta expandir os filhos de B. Porém, descobre que o nó B não tem filhos e, portanto, é um nó de falha, devendo ser retirado da árvore. Como o campo REFAZ do elemento associado ao nó B é vazio, o algoritmo não necessita desfazer nenhuma substituição nos ambientes associados a ancestrais de B.

O nó D é derivado da cláusula que rotula C utilizando-se a cláusula 1 e substituindo-se a variável V de C por b. Esta substituição afeta o campo AMB do elemento associado a C e será registrado no campo REFAZ do elemento associado a D.

A configuração da multilista neste ponto será:

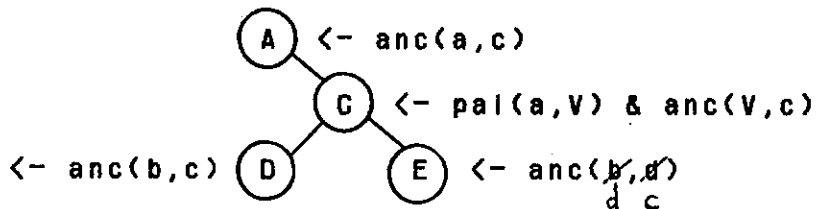
	AMB	REFAZ	PAI	FILHO	IRMÃO	EXP	NA
1.	\emptyset	\emptyset	-	3	-	V	A
3.	{U/a, V/b, W/c}	\emptyset	1	4	-	V	C
4.	\emptyset	{V/V/3}	3	-	-	F	D

Note que em cada substituição de um campo REFAZ existe um número que indica o elemento que teve o campo AMB alterado pela unificação representada neste elemento. Por exemplo, V/V/3 indica que a substituição V/b do campo AMB do elemento 3 deve ser trocada por V/V.

Como o ramo terminando em D tem comprimento 2, o algoritmo pára a expansão e retrocede ao pai de D para derivar seu próximo filho. A lista REFAZ do elemento associado a D é não vazia, indicando que deverão ser feitas alterações em elementos associados aos ancestrais de D. Neste caso, o ambiente associado a C deve ser modificado para retornar às substituições existentes no momento em que C foi criado. Esta operação é feita simplesmente trocando as substituições do ambiente associado a C pelas substituições indicadas na lista REFAZ do ambiente associado a D que atuam sobre a mesma variável. Então, a substituição V/b é trocada por V/V, como indica a configuração da multilista abaixo:

	AMB	REFAZ	PAI	FILHO	IRMÃO	EXP	NA
1.	\emptyset	\emptyset	-	3	-	V	A
3.	{U/a, V/V, W/c}	\emptyset	1	4	-	V	C
4.	\emptyset	{V/b/3}	3	-	-	F	D

A geração do irmão de D resultará em:



que será representada na multilista assim:

	AMB	REFAZ	PAI	FILHO	IRMÃO	EXP	NA
1.	\emptyset	\emptyset	-	3	-	V	A
3.	{U/a, V/d, W/c}	\emptyset	1	4	-	V	C
4.	\emptyset	{V/b/3}	3	-	5	F	D
5.	\emptyset	{V/V/3}	3	-	-	F	E

Como A e C não tem mais filhos e todos os ramos possuem comprimento 2, o algoritmo retrocede à raiz. O retrocesso desfaz o efeito da unificação ocorrida no nó E no ambiente do elemento 3, colocando a multilista no seguinte estado:

	AMB	REFAZ	PAI	FILHO	IRMÃO	EXP	NA
1.	0	0	-	3	-	V	A
3.	{U/a,V/V,W/c}	0	1	4	-	V	C
4.	0	{V/b/3}	3	-	5	F	D
5.	0	{V/d/3}	3	-	-	F	E

Em seguida, o algoritmo caminha em profundidade pela árvore já construída, tentando expandir os ramos até a altura $3k=3$. À medida que caminha até uma folha, o algoritmo refaz os ambientes dos ancestrais da folha usando as listas REFAZ dos elementos associados aos ancestrais ou do elemento associado à folha.

Percorrer a árvore até a folha D resultará então em uma multilista:

	AMB	REFAZ	PAI	FILHO	IRMÃO	EXP	NA
1.	0	0	-	3	-	V	A
3.	{U/a,V/b,W/c}	0	1	4	-	V	C
4.	0	{V/V/3}	3	-	5	F	D
5.	0	{V/d/3}	3	-	-	F	E

Note que o campo AMB do elemento 3 foi modificado de acordo com a informação do campo REFAZ do elemento 4.

O algoritmo agora expandirá os filhos de D como ilustrado anteriormente, já que o campo EXP do elemento 4 é FALSO. O resto da construção da árvore segue de forma semelhante.

O resto desta seção apresenta o algoritmo que implementa a estratégia mista. O algoritmo tem uma parte inicial que percorre árvores já construídas, pesquisando cada nó da forma descrita a seguir:

- . verifica se cn tem filho, o que implica que existe uma sub-árvore que tem cn como raiz;
- . verifica se cn ainda não foi expandido, isto é, cn não tem filhos porque a altura máxima foi atingida;
- . verifica se cn é nó de falha, devendo, portanto, ser removido.

Como dito anteriormente, se a árvore for totalmente criada até uma certa altura e a cadeia vazia não for gerada, o algoritmo expandirá as folhas, utilizando o Algoritmo Básico modificado para introduzir limitação de comprimento.

Os nós de falha serão removidos através de modificação dos ponteiros. Toda sub-árvore maximal cujas folhas são nós de falha será inteiramente removida.

Em seguida apresentamos o algoritmo para percorrer a parte já construída da árvore.

```
programa principal (k,K,P,Q):
```

```
  /*
```

```
    k : incremento do comprimento máximo dos ramos
```

```
    K : limite máximo do comprimento dos ramos
```

```
    P : programa
```

```
    Q : consulta
```

```
  */
```

```
begin
```

```
  cn := inicializa;
```

```
  n := k;
```

```
  while ( n <= K ) and ( cn <> nil )
```

```
    do
```

```
      begin
```

```
        percorrearvore ( cn,k );
```

```
        n := n + k
```

```
      end;
```

```
end;
```

```
rotina percorrearvore ( cn,k ) :
```

```
begin
```

```
  if cn = nil
```

```
    then
```

```
      return;
```

```
  refazambiente ( cn );
```

```
  percorrearvore ( cn.FILHO , k )
```

```
  if ¬cn.EXP
```

```
    then
```

```
      begin
```

```
        expandir ( cn , k );
```

```
        cn.EXP := TRUE
```

```
      end;
```

```
  desfazambiente ( cn );
```

```
  cni := cn.IRMÃO;
```

```
  tentaremove ( cn );
```

```
  percorrearvore ( cni , k )
```

```
end;
```

```

função inicializa:
    Constrói o elemento associado à raiz da
    árvore a partir de Q e retorna um apontador
    para este elemento. O campo EXP deste
    elemento recebe o valor FALSO.

rotina refazambiente ( cn );
    Faz as modificações indicadas na lista
    REFAZ de cn;

rotina desfazambiente ( cn );
    Desfaz as modificações indicadas na lista
    REFAZ de cn;

rotina tentare remover ( cn );
    If cn.FILHO = nil
        then
            retira o elemento associado a cn da
            multilista através da modificação
            dos ponteiros.

rotina expandir ( cn );
    Esta rotina é o Algoritmo Básico modificado
    para:
    .limitar o comprimento dos ramos
    .não retirar da multilista os elementos
    correspondentes a nós de ramos incompletos
    durante um retrocesso
    .manter os campos PAI, FILHO e IRMÃO dos
    elementos e associar ao campo EXP das
    folhas o valor FALSO

```

4. UMA AVALIAÇÃO DO ALGORITMO DE EXPANSÃO MISTA

Conforme comentado brevemente na seção 1, a estratégia de expansão das árvores de refutação em profundidade, utilizada pelas implementações usuais de Prolog, é um ponto controverso, devendo ser analisadas estratégias alternativas combinando expansão em profundidade com expansão em amplitude. Uma resposta parcial, mas objetiva, para esta discussão pode ser dada comparando-se o Algoritmo Básico da seção 2 com o Algoritmo de Expansão Mista da seção 3. Esta seção avalia então, de forma breve, estes dois algoritmos sob três aspectos: complexidade de código, complexidade de tempo e complexidade de espaço.

Sob o primeiro aspecto, os dois algoritmos são qualitativamente equivalentes. As rotinas adicionais do Algoritmo de Expansão Mista são simples, assim como os campos adicionais necessários. Isto significa que o Algoritmo Básico com pequenas modificações faz o trabalho principal do Algoritmo de Expansão Mista. Em termos práticos, as modificações necessárias em um interpretador padrão Prolog para implementar uma estratégia mista não seriam extensas.

A comparação em termos de tempo e espaço está

Intimamente ligada ao uso de compartilhamento de estruturas e à expansão simultânea de mais de um ramo. Compartilhamento de estruturas, por um lado, força o Algoritmo de Expansão Mista a consumir tempo adicional para desfazer e refazer ambientes ao expandir ramos alternados. Por outro lado, minimiza significativamente o custo adicional de espaço para representar simultaneamente vários ramos.

Portanto, a vantagem do Algoritmo Básico reside principalmente em armazenar apenas um ramo. Porém, esta vantagem é relativa pois o Algoritmo Básico poderá, em certos casos, expandir um ramo infinito antes de encontrar um ramo de sucesso. Nestes casos, o Algoritmo Básico irá parar por falta de memória, enquanto que o Algoritmo de Expansão Mista poderá encontrar um ramo de sucesso, se existir espaço suficiente para a expansão limitada dos ramos.

Finalmente, observamos que a técnica descrita em Warren [1984] evita o uso do campo REFAZ para refazer ambientes quando um novo ramo é expandido ou para desfazer/refazer ambientes quando o comprimento máximo é aumentado. Por outro lado, esta técnica exige caminhar por um ramo em direção à raiz a cada nova unificação, o que a torna ineficiente.

5. BIBLIOGRAFIA

- Boyer, R.S. e J.S. Moore [1972]. "The Sharing of Structure in Theorem-Proving Programs", em *Machine Intelligence* 7, Edinburgh, 101-116.
- Bruynooghe, M. [1982]. "The memory management of Prolog implementations", em *Logic Programming*, K.L. Clark and S.-A. Tarnlund (eds.), Edinburgh U. Press, Edinburgh, 83-98.
- Horowitz, E. e S. Sahni [1978]. *Fundamentals of Data Structures*, Computer Science Press, Inc., Potomac, Maryland.
- Lloyd, J.W. [1984]. *Fundamentals of Logic Programming*, Springer-Verlag.
- Van Emden, M.H. [1984]. "An interpreting algorithm for Prolog programs", em *Implementations of Prolog*, J.A. Campbell (ed.), John Wiley & Sons, 93-110.
- Warren, D.S. [1984]. "Efficient Prolog Memory Management for Flexible Control Strategies", em *New Generation Computing* 2, Springer-Verlag, 361-369.