COHN

PROGRAMAÇÃO EM LÓGICA, PROLOG E RESTRIÇÕES : PODER DE EXPRESSÃO vs EFICIÊNCIA

Este exemplar corresponde à redação final da tese devidamente corrigida e defendida pelo Sr. Paulo Gomide Cohn e aprovada pela Comissão Julgadora.

Campinas, 18 de Julho de 1991.

Antonio Eduardo Costa Pereira

Prof. Dr. Antonio Eduardo Costa Pereira
Orientador

Dissertação apresentada ao Instituto de Matemática, Estatística e Ciência da Computação, UNICAMP, como requisito parcial para obtenção do Título de Mestre em Ciência da Computação.

UNIVERSIDADE ESTADUAL DE CAMPINAS

INSTITUTO DE MATEMÁTICA ESTATÍSTICA E CIÊNCIA DA COMPUTAÇÃO

DEPARTAMENTO DE CIÊNCIA DA COMPUTAÇÃO

PROGRAMAÇÃO EM LÓGICA, PROLOG E RESTRIÇÕES : PODER DE EXPRESSÃO vs. EFICIÊNCIA.

Autor: Paulo Gomide Cohn

Orientador: Prof. Dr. Antonio E. Costa Pereira

Co-orientador: Prof. Dr. Tomasz Kowaltowski

RESUMO

Apresenta-se inicialmente uma introdução à programação em lógica através de uma abordagem evolutiva. Começando de um sistema formal de primeira ordem bastante complexo, descreve-se o conceito de prova de teoremas e sua automação. A partir daí apresenta-se a idéia de eficiência da prova. Os principais avanços obtidos durante o século XX nesta área são apontados, dando-se ênfase ao princípio de resolução de Robinson. Ao restringir a linguagem do sistema formal às sentenças de Horn, obtem-se uma grande melhora da eficiência do mecanismo de prova, preservando razoável poder de expressão. Alguns problemas relativos à expressividade da linguagem são apontados assim como formas em que têm sido abordados na atualidade. Uma delas é a programação por restrições.

Na segunda parte do trabalho é apresentada a implementação de um interpretador/sistema de execução para a linguagem Prolog. A partir de uma especificação breve, de alto nivel, são introduzidos, incrementalmente, os detalhes de implementação de nivel mais baixo (estruturas de dados, controle de execução) até se obter um programa puramente procedimental escrito numa linguagem convencional (C).

Finalmente, os conceitos de programação por restrições mencionados no início do trabalho são apresentados de forma mais detalhada para que se possa ter uma visão mais clara do seu funcionamento na prática. Assim, adotando uma abordagem semelhante à utilizada para descrever o compilador Prolog, é apresentado um interpretador de alto nivel para programas de restrições e, após alguns refinamentos, é obtido um conjunto de instruções que pode ser implementado numa linguagem de programação convencional.

ABSTRACT

An introduction to logic programming is initially presented by means of an evolutionary approach. Starting with a fairly complex first order formal system, the concepts of theorem proving and its automatization are described. From there the idea of proof efficiency is presented. The most significant advances obtained during the 20th century in this area are pointed out emphasizing Robinson´s resolution principle. Restraining the formal system´s language to Horn sentences, a major improvement of the proof mechanism is obtained, preserving reasonable expressive power. Some problems related to language expressiveness and the ways in which they have been recently approached are shown. One of these approaches is constraint programming.

In the second part of the work, the implementation of an interpreter/runtime system for the Prolog language is shown. From a brief, high-level specification, lower-level implementation details (data structures, execution control) are introduced until a purely procedural program written in a conventional language (C) is obtained.

Finally, the contraint programming concepts mentioned in the first part are presented in a more detailed form in order to allow for a clearer vision of how it works in practice. Thus, by means of an approach similar to that used to describe the Prolog compiler, a high-level interpreter for constraint programs is presented and, after some refinements, an instruction set that may be implemented in a conventional programming language is obtained.

CONTEÚDO

DEDICATÓRIA

Aos meus pais.

AGRADECIMENTOS

Se tivesse que descrever em uma palavra qual o objeto mais importante deste trabalho, sem hesitar diria que é o tempo, o recurso absoluto, não renovável e inesgotável tão precioso que precisamos sempre encontrar a melhor maneira possível de utilizá-lo. Este trabalho fala principalmente de tempo: o tempo que as pessoas gastam em expressar seus conhecimentos de uma forma suficientemente precisa para que possam ser manipulados por máquinas e o tempo que as máquinas gastam para realizar tais manipulações. Como veremos, foi necessário investir muito tempo para que atingissemos um estágio no qual fosse possível obter algum resultado útil desta "Engenharia do Conhecimento" e, no entanto, sabemos que resta ainda muito por fazer.

Muitas pessoas generosamente cederam uma parcela desse recurso mais precioso para me ajudar, de inúmeras formas, a levar adiante este trabalho e a elas gostaria agora de expressar meu mais profundo agradecimento:

Ao Prof. Antonio E. Costa Pereira, por me mostrar a programação em lógica, o Prolog, a Inteligência Artificial. Por me ensinar como tudo isto funciona e o que se pode fazer. Por me indicar o caminho e pelo grande incentivo. E principalmente pela infinita paciência.

Ao Prof. Walter A. Carnielli, por me mostrar como tudo isto faz sentido: as origens de todas estas idéias e seus embasamentos. Por me fazer entender o que é a lógica e a computabilidade e me fornecer recursos para saber decidir, em cada momento, se o que estava fazendo era correto. Pelo incentivo e pela infinita paciência.

À Profa. Dra. Carolina Monard por começar tudo. Pela motivação, apoio e incentivo.

À Profa. Dra. Elza Gomide, a tia Elza, por me ajudar a "decifrar" as cláusulas de Horn.

Ao Departamento de Ciência da Computação da UNICAMP, em especial aos Profs. Tomász Kowaltowski e Hans Liesenberg, por me ajudar a superar os mais diversos entraves e barreiras no longo caminho que leva um aluno especial a aluno regular e, daí, a Mestre em Ciência da Computação.

Aos Sres. Sinésio e José Ignácio Fernandes, diretores da Infortec Informática e Tecnologia, meu local de trabalho, pela flexibilidade na alocação do meu tempo, e pela cessão de tempo e espaço de computador, indispensáveis para a execução deste trabalho.

À Expersystems Informática Inteligente, por me ajudar a encontrar um sentido "prático" para este trabalho.

A tantos outros colegas e amigos, em especial a Artemis Moroni, pelo apoio, incentivo, trocas de idéias e artigos, utilizados como referência e complemento bibliográfico neste projeto.

A todos, Muito Obrigado.

Campinas, Junho de 1991.

Capítulo I. Programação em Lógica

1.1 Objetivo da Tese.

Neste trabalho, descreveremos inicialmente a programação em lógica : sua fundamentação e seu desenvolvimento ao longo deste século. Sabe-se pelo Teorema de Church [Epstein, 1989] que os sistemas lógicos de primeira ordem são indecidíveis, portanto a implementação de um sistema de programação em lógica é na verdade a busca de um sistema lógico onde o equilibrio de clareza, completude, corretude e eficiência das regras de inferência utilizadas assim como do conjunto de sentenças válidas na linguagem resulte numa ferramenta útil para a programação de computadores.

Posteriormente abordaremos a interpretação e compilação de programas em Prolog. A razão para isto é a seguinte: Os trabalhos sobre compilação de Prolog que levam em conta as otimizações indispensáveis (tais como recursividade de cauda) não apresentam de forma satisfatóriamente clara os algoritmos utilizados [Warren, D.H.D. 1977; Bruynooghe, 1980; Clocksin, 1981; van Emden, 1984; Bowen, 1983; Warren, D.H.D. 1983]. Existem, por outro lado, vários trabalhos em que a execução de Prolog é apresentada de maneira descritiva e clara [Warren, D.S. 1983; Bowen, K. 1985]. Infelizmente, estes últimos trabalhos não mostram como implementar as otimizações indispensáveis para que Prolog consiga realizar algo tão simples quanto um laço. Assim sendo, decidimos começar por fornecer ao leitor os subsídios indispensáveis para que atinja, da maneira mais rápida, a última palavra no que se refere a execução de Prolog. Isto será feito no Capítulo II.

O passo seguinte será apresentar um sistema de computação lógica mais descritivo do que o Prolog, ou seja, que seja capaz de descobrir automáticamente uma parcela maior do controle necessário à execução. Nosso sistema se limitará a trabalhar com números e usará lógica de restrições. Assim sendo, será similar ao CLP(R)

[Lassez, 1990] e ao Prolog III [Colmerauer, 1990]. Apesar disto, será suficiente para demonstrar os princípios sobre os quais estamos falando. Proporemos também um modo de compilar programas escritos em linguagens com restrições. Não temos conhecimento de trabalhos nesta área visto que tanto os autores de CLP(R) quanto Colmerauer se limitam a trabalhar com interpretadores.

1.2 Introdução.

Os princípios que regem a construção e execução de programas lógicos têm sua origem no cálculo de predicados de primeira ordem [Kowalski, 1974] e é nele que vamos buscar sua fundamentação. Desde os primórdios da prova automática de teoremas, sabia-se que se podia utilizar mecanismos de inferência sobre conjuntos de sentenças da lógica de primeira ordem para se obter os mais diversos resultados úteis à computação. Chang e Lee [Chang,1973] mostraram que provadores de teoremas podiam ser usados para gerar programas de computador. Em tempos mais recentes, Wos et allii [Wos,1984] usaram provadores para resolver os mais diversos problemas de engenharia, matemática e computação.

O cálculo de predicados é, nos tempos modernos, sempre apresentado como um sistema formal. Um sistema formal é composto de uma linguagem, um conjunto de axiomas e um conjunto de regras de inferência. No caso particular do Cálculo de Predicados, a linguagem define fórmulas bem formadas e sua gramática é a seguinte:

```
fórmula ::= literal
fórmula ::= ¬ fórmula
fórmula ::= fórmula ∧ fórmula
fórmula ::= fórmula ∨ fórmula
fórmula ::= fórmula → fórmula
fórmula ::= fórmula ↔ fórmula
fórmula ::= ∀ variável fórmula
fórmula ::= ∃ variável fórmula

literal ::= pred ( termo termos )

termo ::= variável
termo ::= constante
termo ::= função( termo termos )

termos ::= , termo termos
termos ::= ε
```

Figura 1. Gramática para fórmulas bem formadas de primeira ordem.

Por exemplo, as frases

Beethoven compôs a nona sinfonía.

Qualquer composição de Beethoven é clássica.

Beethoven compôs algumas sonatas e concertos.

poderiam ser expressas na linguagem de primeira ordem, como segue:

compôs(Beethoven, nona_sinfonía)

$\forall x$ compôs(Beethoven, x) $\rightarrow$ clássica(x)

$\exists x$ sonata(x) $\wedge$ compôs(Beethoven, x) $\wedge$ $\exists y$ concerto(y) $\wedge$ compôs(Beethoven,y)

Em geral, os diversos sistemas utilizados com o cálculo de predicados não divergem muito quanto à linguagem. A principal diferença entre eles está na escolha dos axiomas e das regras de inferência. Para dar uma idéia mais clara do que são estes sistemas, vamos apresentar a seguir o Sistema de Dedução Natural (SDN) [Gries, 1981], concebido por Gerhard Gentzen com o objetivo de capturar nossos padrões "naturais" de raciocínio. Neste sistema, introduz-se a idéia de que o raciocínio sobre sentenças da linguagem natural pode ser formalizado por meio de mecanismos de prova de teoremas onde cada sentença introduzida durante a prova pode ser justificada através de sentenças préviamente enunciadas e regras de inferência que permitem que novas sentenças

sejam derivadas.

As regras de inferência do SDN são quatorze: uma regra de inserção e uma regra de eliminação para cada um dos conectores lógicos **não**, **e**, **ou**, **implica**, **igual** e para os quantificadores $\exists$ e $\forall$. Para entender a estrutura das regras, examinemos a primeira delas. O nome da regra é ^_I e aparece à esquerda do dois pontos. E_1, E_2, ..., E_n são sentenças préviamente enunciadas e, consequentemente, deduzidas. $E_1 \wedge E_2 \wedge \ldots \wedge E_n$ é a sentença que se pode deduzir pela aplicação da regra. Na segunda regra, está envolvida uma única sentença préviamente enunciada, a saber: $E_1 \wedge E_2 \wedge \ldots \wedge E_n$. Dela se pode deduzir E_i onde i é um inteiro entre 1 e n. As outras regras têm semântica semelhante. Numa regra de inferência, costuma-se chamar as sentenças que estão acima do traço de prótese e as que estão abaixo de conclusão ou apódose.

$$(1)\ \wedge\_I:\ \frac{E_1,\ \ldots,\ E_n}{E_1 \wedge \ldots \wedge E_n}$$

$$(2)\ \wedge\_E:\ \frac{E_1 \wedge \ldots \wedge E_n}{E_i}$$

$$(3)\ \vee\_I:\ \frac{E_i}{E_1 \vee \ldots \vee E_n}$$

$$(4)\ \vee\_E:\ \frac{E_1 \vee \ldots \vee E_n,\ E_1 \rightarrow E,\ \ldots,\ E_n \rightarrow E}{E}$$

(5) $\Rightarrow$_I: $\dfrac{\text{De } E_1, \ldots, E_n \text{ infira } E}{(E_1 \wedge \ldots \wedge E_n) \Rightarrow E_i}$

(6) $\Rightarrow$_E : $\dfrac{E_1 \Rightarrow E_2,\ E_1}{E_2}$ (modus ponens)

(7) =_I : $\dfrac{E_1 \Rightarrow E_2,\ E_2 \Rightarrow E_1}{E_1 = E_2}$

(8) =_E : $\dfrac{E_1 = E_2}{E_1 \Rightarrow E_2,\ E_2 \Rightarrow E_1}$

(9) $\neg$_I : $\dfrac{\text{De } E \text{ infira } E_1 \wedge \neg E_1}{\neg E}$

(10) $\neg$_E : $\dfrac{\text{De } \neg E \text{ infira } E_1 \wedge \neg E_1}{E}$

(11) $\forall$_I : $\dfrac{R \Rightarrow E}{\forall i : R : E}$ (i é uma variável não presente em E)

(12) $\forall$_E : $\dfrac{\forall i : R : E}{R^i_e \Rightarrow E^i_e}$

$$(13)\ \exists\_I : \frac{\forall i : R : E}{\neg(\exists i : R : \neg E)}$$

$$(14)\ \exists\_E : \frac{\exists i : R : E}{\neg(\forall i : R : \neg E)}$$

Os teoremas a serem provados no SDN têm a forma :

De s_1, s_2, ..., s_n infira s

onde s_1, s_2, ..., s_n são predicados chamados premissas e s é um predicado chamado conclusão. A prova de um teorema no SDN terá a seguinte forma :

De s_1, s_2, ..., s_n infira s

1	s_1	premissa 1
2	s_2	premissa 2
	:	
n	s_n	premissa n
i	s_i	justificativa i
i+1	s_{i+1}	justificativa i+1
	:	
	:	
m	s	justificativa m

Na prova acima, cada linha é verdadeira e é justificada pela aplicação de uma regra de inferência sobre um subconjunto das linhas anteriores. A justificativa aparece diante da expressão s_j. As linhas de 1 a n são justificadas pelas premissas.

Vejamos, para exemplificar, alguns exemplos apresentados em [Gries,1981]. Admitamos que queiramos provar que de $p \wedge q$ pode-se deduzir $p \wedge (r \vee q)$. A prova fica assim:

De p ∧ q infira p ∧ (r ∨ q)		
1	p ∧ q	premissa 1
2	p	∧_E, 1
3	q	∧_E, 1
4	r ∨ q	∨_I, 3
5	p ∧ (r ∨ q)	∧_I, 2, 4

Observe, na prova acima, que a justificativa da veracidade de uma sentença sempre contém a regra de inferência e as linhas sobre as quais ela foi usada. Por exemplo, a linha p ∧ (r ∨ q) é verdadeira pela regra de inferência ∧_I aplicada às linhas 2 e 4. Isto é expresso assim: ∧_I, 2, 4.

Note o leitor que nas regras (5), (9) e (10) um enunciado de teorema aparece na prótese da regra. Isto deve ser interpretado assim: se o teorema da prótese for demonstrado usando as outras regras, pode-se concluir a apódose.

Frequentemente convém utilizar-se de subteoremas nas provas. Seja, por exemplo, provar que (p ∧ q) = (q ∧ p). Podemos começar provando que de p ∧ q pode-se deduzir q ∧ p. Assim:

Teorema 1.

De p ∧ q infira q ∧ p		
1	p	∧_E, premissa 1
2	q	∧_E, premissa 1
3	q ∧ p	∧_I, 1, 2

E depois provar que de q ∧ p pode-se deduzir p ∧ q:

Teorema 2.

De q ∧ p infira p ∧ q		
1	p	∧_E, premissa 1
2	q	∧_E, premissa 1
3	p ∧ q	∧_I, 1, 2

Agora podemos realizar a prova final, apresentada a seguir.

infira (p ∧ q) = (q ∧ p)

1	(p ∧ q) ⇒ (q ∧ p)	⇒_I, Teorema 1
2	(q ∧ p) ⇒ (p ∧ q)	⇒_I, Teorema 2
3	(p ∧ q) = (q ∧ p)	=_I, 1, 2

Provas que utilizam as regras de inclusão e exclusão do não denominam-se provas por contradição. Em geral, elas se processam da seguinte maneira: Nega-se a tese que se quer provar. Caso, com isto, chegue-se a uma contradição, ou seja, caso conclua-se que algo é verdadeiro e falso ao mesmo tempo, conclui-se que a tese negada era falsa e que, portanto, a tese é verdadeira. Seja, por exemplo, provar que de p deduz-se ¬¬p.

De p infira ¬¬ p

1	p	premissa 1
2	**De ¬p infira p ∧ ¬p**	
	2.1 | p ∧ ¬p	∧_I, 1, premissa 1
3	¬¬p	¬_I, 2

Um outro exemplo interessante é que a partir de uma contradição é possível provar qualquer coisa. Assim:

De p, ¬p infira q

1	p	premissa 1
2	¬p	premissa 2
3	**De ¬q infira p ∧ ¬p**	
	2.1 | p ∧ ¬p	∧_I, 1, 2
4	q	¬_E, 3

A tarefa de um provador automático de teoremas para o SDN seria a de executar um processo de busca no qual ele poderia olhar três coisas :

1) Combinar as premissas ou derivar sub-predicados a partir das premissas, visando produzir a conclusão ou algo

semelhante a ela.

2) Investigar a própria conclusão. Já que deverá ser usada uma regra de inferência, a forma da conclusão deveria nos ajudar a decidir qual regra usar.
3) Olhar as regras de inferência. Existem quatorze, o que abre um número grande de possibilidades. Felizmente, não todas serão aplicáveis a uma certa tese já que esta deverá ter a forma da conclusão da regra usada para justificá-la.

Podemos concluir, observando os exemplos dados, que os métodos passíveis de utilização para a automatização da prova de teoremas com este sistema são inerentemente ineficientes. Este problema, o do grau de complexidade da prova em função do teorema a ser provado e das regras de inferência disponíveis, direcionou a pesquisa em prova automática de teoremas dando origem a alguns refinamentos que diminuem tal complexidade de ordens exponenciais para ordens lineares.

Herbrand adotou uma abordagem importante nesta direção [Chang, 1973]: êle propôs um algoritmo que encontra uma interpretação, ou seja, um conjunto de valores para as variáveis presentes numa fórmula, que a tornam falsa. Porém, se a fórmula for válida, (verdadeira para qualquer interpretação), então o algoritmo irá parar depois de um número finito de passos. A primeira tentativa de utilizar o algoritmo de Herbrand como base para um provador automático de teoremas foi realizada por Gilmore [Chang, 1973]. Sua idéia era básicamente a seguinte: ele negava a fórmula ou parte dela. Procurava, então, uma interpretação que tornasse falsa a negação da fórmula. Se tal interpretação fosse encontrada, por redutio ad absurdum, a fórmula seria verdadeira.

O método de Gilmore, porém, era ineficiente e [Davis e Putnam, 1960] introduziram um refinamento obtido a partir de transformações sintáticas nas sentenças de primeira ordem que as tornam estruturalmente mais simples conservando, no entanto, o poder de expressão da linguagem como um todo. Sobre esta linguagem simplificada, podem ser aplicadas quatro regras

(lógicamente corretas) que reduzem o conjunto de sentenças até que seja obtida uma refutação ou até que não hajam mais reduções possíveis. A seguir falaremos sobre as transformações propostas por Davis e Putnam.

Qualquer sentença do Cálculo de Predicados de Primeira Ordem pode ser reescrita na forma de uma conjunção de disjunções, chamada de forma clausal ou forma padrão, na qual só existem quantificadores universais de escopo máximo, ou seja, têm a seguinte forma:

$$\forall x \forall y \forall z \ldots ((L_{11} \vee L_{12} \vee L_{13} \ldots) \wedge (L_{21} \vee L_{22} \vee L_{23} \ldots) \wedge (L_{31} \vee L_{32} \vee L_{33} \ldots) \wedge \ldots)$$

onde L_{ij} são literais, isto é, predicados, ou negações de literais e as únicas variáveis que aparecem em L_{ij} são aquelas introduzidas pelos quantificadores (i.e., não existem variáveis livres). A gramática, em notação BNF, para sentenças na forma de conjunção de disjunções é fornecida na Figura 2.

```
sentença ::= quant ( conjunção )

quant ::= ∀ variável quant
quant ::= ε

conjunção ::= disjunção conj

conj ::= ∧ disjunção conj
conj ::= ε

disjunção ::= ( literal disj )

termo ::= variável
termo ::= constante
termo ::= função( termo termos )

termos ::= , termo termos
termos ::= ε

literal ::= pred ( termo termos )
literal ::= ¬ pred ( termo termos )

disj ::= ∨ literal disj
disj ::= ε
```

Figura 2. Gramática para sentenças em forma padrão.

A conversão, proposta por Davis e Putnam, de uma fórmula do cálculo de predicados para a forma padrão envolve seis passos (Em [Clocksin, 1981] é mostrado um programa em Prolog que efetúa esses passos.) :

1. Retirar as implicações.

$\alpha \Rightarrow \beta$	$(\neg\alpha) \vee \beta$
$\alpha \Leftrightarrow \beta$	$(\alpha \wedge \beta) \vee (\neg\alpha \wedge \neg\beta)$

2. Distribuir as negações.

$\neg(\alpha \vee \beta)$	$\neg\alpha \wedge \neg\beta$
$\neg(\alpha \wedge \beta)$	$\neg\alpha \vee \neg\beta$

3. Skolemização (Eliminar os quantificadores existenciais).
   Substituir as variáveis introduzidas por quantificadores existenciais por constantes ou funções novas (existe uma interpretação dos simbolos de uma fórmula que a torna verdadeira se e somente se existir uma interpretação para a versão Skolemizada da fórmula).

$\exists x(p(x))$	$p(g)$
$\forall x(p(x) \Rightarrow \exists y(q(x,y)))$	$\forall x(p(x) \Rightarrow q(x,g(x)))$

4. Maximizar o escopo dos quantificadores universais.
   Reescreve-se a fórmula de modo que toda variável seja quantificada universalmente e o escopo da quantificação seja a fórmula inteira. Para facilitar a identificação das variáveis dentro da fórmula, costuma-se usar uma notação especial (e.g. todas as variáveis são sequências de caracteres iniciadas por maiúsculas).

5. Distribuir conjunções sobre disjunções (Forma normal conjuntiva).

$(\alpha \wedge \beta) \vee \gamma$	$(\alpha \vee \gamma) \wedge (\beta \vee \gamma)$
$\alpha \vee (\beta \wedge \gamma)$	$(\alpha \vee \beta) \wedge (\alpha \vee \gamma)$

6. Colocar em cláusulas (Forma clausal, forma padrão)
   A conjunção é transformada numa coleção.

$\alpha \wedge \beta \wedge \gamma \wedge \ldots$	$\{\alpha, \beta, \gamma, \ldots\}$

Nossa sentença original do cálculo de predicados de primeira ordem ficou sendo representada por um conjunto de cláusulas :

$$\{ (L_{11} \vee L_{12} \vee L_{13} \vee \dots), \\ (L_{21} \vee L_{22} \vee L_{23} \vee \dots), \dots \}$$

O principal problema com o uso de procedimentos baseados no método de Herbrand é que eles requerem a geração de conjuntos de valores para as variáveis presentes nas cláusulas e, na maioria dos casos, esses conjuntos crescem exponencialmente. Robinson [1963] expôs uma análise da característica explosiva destes procedimentos e posteriormente [Robinson, 1965] introduziu um novo procedimento, chamado de **resolução**, que, em conjunto com o algoritmo de **unificação**, proporciona o conjunto de valores que devem ser atribuidos às variáveis para tornar um conjunto de cláusulas insatisfatível, se tais valores existirem.

1.3 Princípio de Resolução.

O procedimento usado tanto por Gilmore e Davis-Putnam como por Robinson é o de **refutação** ou **prova por redução ao absurdo**, análogo à prova por contradição discutida anteriormente no SDN. O que se faz é demonstrar que a inclusão da negação de uma cláusula (chamada tese) em um conjunto de cláusulas (chamado hipótese) provoca uma contradição (i.e., faz com que o novo conjunto de cláusulas seja insatisfatível). Portanto, pela lei do terceiro excluído, se a negação da tese provoca uma contradição, a tese original tem que ser consistente com a hipótese. No SDN, esses conceitos estão embutidos nas regras ¬_I e ¬_E : se a partir de uma cláusula for possível inferir uma contradição, então pode-se concluir a negação da cláusula; e se a partir da negação de uma cláusula for possível inferir uma contradição então conclui-se a cláusula não negada. Para entender isto, vamos examinar o algoritmo de resolução frequentemente usado em tais provas :

1. Negue a tese e coloque-a na forma clausal.
2. Faça a união do conjunto tese com o conjunto hipótese.

3. Aplique a regra de inferência conhecida como resolução até obter uma contradição.

A regra (de inferência) de resolução pode ser expressa no sistema de dedução natural da seguinte forma :

De $P \vee Q$, $\neg P \vee R$ **infira** $Q \vee R$

Para prová-la, basta observar que através da regra de inserção do ou ($\vee$_I) podemos obter duas disjunções iguais exceto por um literal que aparece negado em uma delas, mas não na outra. Depois de algumas manipulações é possível eliminar esses literais produzindo então uma nova disjunção, chamada **resolvente**, que corresponde a uma nova linha da prova :

De $P \vee Q$, $\neg P \vee R$ **infira** $Q \vee R$		
1	$P \vee Q \vee R$	$\vee$_I, pr. 1
2	$\neg P \vee Q \vee R$	$\vee$_I, pr. 2
3	$(P \vee Q \vee R) \wedge (\neg P \vee Q \vee R)$	$\wedge$_I, 1, 2
4	$(Q \vee R) \vee (P \wedge \neg P)$	lei distributiva, 3
5	$Q \vee R$	lei da simplificação do ou

Nota : A forma usada da lei distributiva é
$(a \vee b) \wedge (a \vee c) = a \vee (b \wedge c)$
e a da lei da simplificação do ou é
$a \vee (b \wedge \neg b) = a$.
A prova destas leis pode ser encontrada em [Gries, 1981].

Vejamos um exemplo de prova usando o principio de resolução. Suponhamos que nos são dados os seguintes identificadores e suas interpretações :

hu(x)	X é humano
h(x)	X é homem
m(x)	X é mulher

e que temos o seguinte conjunto Hipótese :

$hu(x) \vee \neg h(x)$	(X é humano ou X não é homem)
$hu(x) \vee \neg m(x)$	(X é humano ou X não é mulher)
$h(wang) \vee m(wang)$	(Wang é homem ou Wang é mulher)

desejamos provar a seguinte Tese :

$\exists y\ hu(y)$	(Alguém é humano)

1. Negar a tese e colocá-la em forma clausal :

   $\neg\exists y\ hu(y)$

   $\forall y\ \neg hu(y)$

   $\neg hu(y)$

2. Faça a união do conjunto tese com o conjunto hipótese.

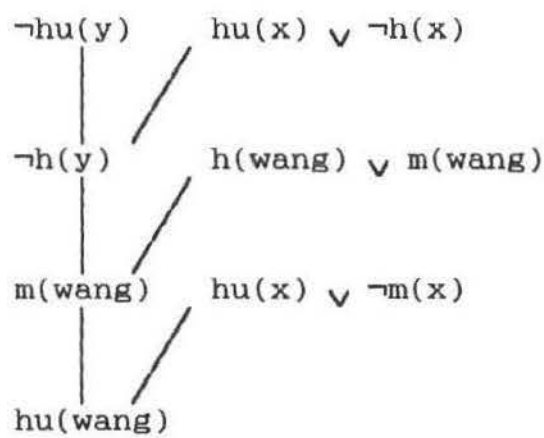
   $hu(x) \vee \neg h(x)$

   $hu(x) \vee \neg m(x)$

   $h(wang) \vee m(wang)$

   $\neg hu(y)$

3. Aplique a regra de inferência conhecida como resolução até obter uma contradição.



contradição

Por exclusão do não (¬_E) ¬hu(y) é falso e hu(y) é verdadeiro.

1.4 Eficiência e Decidibilidade.

Seja S um conjunto de cláusulas. Seja $\mathbb{R}(S)$ o conjunto união de S com o conjunto de todos os resolventes de S. $\mathbb{R}^n(S)$, neste caso, é $\mathbb{R}(\mathbb{R}^{n-1}(S))$. Um provador automático de teoremas baseado no procedimento proposto originalmente por Robinson, chamado de **resolução por saturação**, consistiria de um algoritmo para calcular, dado o conjunto S de cláusulas de entrada, a sequência de conjuntos S, $\mathbb{R}(S)$, $\mathbb{R}^2(S)$, ..., até que fosse encontrado, digamos, $\mathbb{R}^n(S)$ que contivesse a cláusula vazia ou, caso contrário, fosse igual ao seu sucessor $\mathbb{R}^{n+1}(S)$. No primeiro caso, uma refutação de S é obtida descobrindo a sequência de resolventes que geraram a cláusula vazia; no outro caso, a conclusão é que S é satisfatível. Pelo Teorema de Church, sabe-se que para algumas entradas S este procedimento, e em geral todos os procedimentos corretos de refutação não irão terminar em nenhum desses dois casos, mas continuarão calculando indefinidamente. Um exemplo destes casos está associado à verificação de ocorrência presente em algumas implementações de Prolog. Consideremos o seguinte conjunto de cláusulas :

$$\{Q(a),\ \neg Q(x) \vee Q(f(x))\}$$

É fácil ver que para este exemplo o procedimento descrito acima geraria sucessivamente os resolventes

$$Q(f(a))$$
$$Q(f(f(a)))$$
$$Q(f(f(f(a))))$$
$$\vdots$$

"ad infinitum". Robinson sugeriu o "Principio de Pureza" através do qual cláusulas poderiam ser removidas a priori do conjunto de entrada e assim evitar este caso de indecidibilidade.

Outro principio proposto por Robinson é o de "Subjugação" que favorece a velocidade de convergência do procedimento. A idéia é que se uma cláusula C subjuga uma cláusula D, isto é, existe uma substituição σ tal que $C\sigma \subseteq D$, então D pode ser eliminada.

Na verdade estes principios, chamados de principios de busca, para diferenciá-los dos principios de inferência, propostos como melhoras para o procedimento de refutação, não proporcionam os resultados desejados já que sua implementação é complicada e, por sua vez, ineficiente.

Algoritmos como o apresentado acima, levam, como é fácil verificar, a uma explosão combinatória : suponhamos que existem n "tipos" de literais num certo conjunto $\mathbb{S}$ de cláusulas; suponhamos também que uma certa cláusula $\mathbb{C}_i \in \mathbb{S}$ contem i literais. É possível, então gerar $k_{ij} \geq 0$ resolventes de tamanho j para $\mathbb{C}_i$, onde $i - 1 \leq j \leq n - 1$. Se $\mathbb{S}$ contiver m cláusulas, então poderão ser geradas

$$R = \sum_{i=1}^{m} \sum_{j=i-1}^{n-1} k_{ij}$$

cláusulas resolventes a partir de $\mathbb{S}$ e $\mathbb{R}(\mathbb{S})$ conterá $R + m$ cláusulas. De maneira semelhante podemos obter o tamanho de $\mathbb{R}^2(\mathbb{S})$:

$$R_2 = \sum_{i=1}^{R} \sum_{j=i-1}^{n-1} k_{ij}$$

onde o número de cláusulas candidatas agora é provávelmente muito maior que na iteração anterior. Evidentemente o algoritmo pode ser melhorado. Pode-se, por exemplo, utilizar-se de idéias tais como conjunto de suporte, filtragem, preferência unitária e resolução linear [Chang, 1973]. Vamos ver, por exemplo, o algoritmo da resolução linear.

A estratégia adotada na resolução linear é análoga ao raciocinio em cadeia utilizado quando se quer provar uma identidade : começando pelo lado esquerdo da igualdade, aplicamos uma regra de transformação para obter uma nova expressão, sobre a qual aplicamos outra regra e repetimos o processo até obter uma expressão idêntica à do lado direito da identidade : dado um conjunto de cláusulas S e uma cláusula C_o em S, uma dedução linear

de C_n a partir de S com cláusula topo C_0 é uma dedução da forma mostrada na Figura 3, onde

1. para i = 0, 1, ..., n - 1, C_{i+1} é um resolvente de C_i (chamada de cláusula central) e B_i (chamada de cláusula lateral), e

2. cada B_i, ou está em S, ou é uma C_j, para j < i.

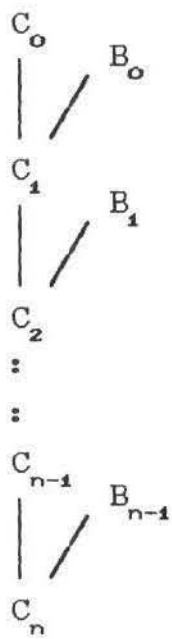


Figura 3. Estrutura de uma dedução linear.

Na realidade, a resolução linear pode ser vista como uma mudança de estratégia de pesquisa dentro de uma árvore, chamada de árvore de prova: enquanto que na resolução por saturação se faz uma pesquisa em largura de **todas** as árvores possíveis **simultâneamente**, na resolução linear se faz uma pesquisa em profundidade de **cada uma** destas árvores.

Intuitivamente, podemos concluir que, em termos de consumo de memória (para armazenar os diversos resolventes gerados), a resolução linear tem um comportamento controlado, enquanto que a resolução por saturação é explosiva. Em termos de tempo de execução (até chegar a uma refutação se ela existir) a eficiência de cada estratégia dependerá do critério de seleção da cláusula inicial e do critério de seleção da cláusula a ser usada para gerar o próximo resolvente. Em alguns casos é possível caracterizar as classes de árvores que poderão ser geradas e determinar funções heurísticas para a seleção de cláusulas que melhoram o desempenho do procedimento para uma ou outra classe específica de árvore. Em outras palavras, o tempo de execução é não determinístico e nos defrontamos com o problema $P \stackrel{?}{=} NP$ quando tentamos estabelecer comparações. O procedimento de resolução Linear com função de Seleção, ou resolução-SL, é amplamente descrito em [Kowalski e Kuehner, 1971].

1.5 Sentenças de Horn.

Dada a utilidade de se usar lógica e prova de teoremas em computação e a impossibilidade de se realizar tais provas em conjuntos de cláusulas quaisquer, procurou-se conjuntos particulares que ainda mantivessem uma completude suficiente para ser útil. Evidentemente, tais conjuntos devem ter propriedades tais que diminuam as possibilidade de resolução. Um modo de se conseguir isto é permitir que apenas um dos literais de cada sentença seja positivo. Alfred Horn [1951] estudou propriedades matemáticas em cláusulas com no máximo um literal positivo e, por isso, tais cláusulas receberam o nome de sentenças de Horn [Cohn, 1965].

Além das propriedades matemáticas e de possibilitarem algoritmos eficientes de prova, as sentenças de Horn possuem uma outra fonte de apelo que as tornam candidatas fortes a se constituirem em um sistema de representação computacional em substituição às cláusulas gerais do Cálculo de Predicados. Elas

podem ser postas na forma :

$$\{ C_1 \leftarrow +L_{11} \wedge +L_{12} \wedge +L_{13} \wedge \ldots ,$$
$$C_2 \leftarrow +L_{21} \wedge +L_{22} \wedge +L_{23} \wedge \ldots , \ldots \}$$

onde "$\leftarrow$" é o símbolo de implicação e $+L_{ij}$ é a negação dos literais negativos. Para isto, basta aplicar a regra de DeMorgan a todas as sentenças. Ora, neste último formato, o símbolo de implicação pode ser lido como "se", usado para introduzir condições em línguas humanas. Com tal leitura, as regras podem ser interpretadas como as regras de algibeira utilizadas pelos seres humanos para expressar conhecimentos em diversas áreas

$$\{ C_1 \text{ se } +L_{11} \text{ e } +L_{12} \text{ e } \ldots,$$
$$C_2 \text{ se } +L_{21} \text{ e } +L_{22} \text{ e } \ldots, \ldots \}$$

Do que foi dito, não é de se admirar que a primeira tentativa de se usar lógica com sucesso em um sistema computacional tenha lançado mão das sentenças de Horn. Esta tentativa deu origem à linguagem Prolog. O principal problema enfrentado pelos projetistas de Prolog foi que as sentenças de Horn não eram suficientemente completas para a maioria das aplicações. Este problema foi resolvido introduzindo-se mecanismos de controle que permitiam ao programador criar procedimentos na representação que compensassem as limitações das sentenças de Horn. Estes mecanismos podem ser divididos em quatro grupos :

1. Métodos de modificar a direção da prova. O mais conhecido destes mecanismos é o corte, mas existem outros (snip, if...then...else, etc).
2. Ordem fixa na escolha dos literais negativos a serem resolvidos na sentença.
3. Ordem fixa na escolha das sentenças.
4. Uso de procedimentos pré-definidos ou escritos em outras linguagens.

Há muito tempo se tornou óbvio para pessoas envolvidas com programação lógica que a grande fraqueza de sistemas como Prolog é forçar o usuário a introduzir controle para compensar as limitações das sentenças de Horn. Evidentemente, a introdução do controle aproxima Prolog das linguagens convencionais e dificulta sua utilização como meio de representação pura de conhecimento. Podemos dizer que a direção a seguir na busca de uma linguagem lógica ideal é aquela que diminui a quantidade de controle exigida do programador. Devemos então buscar algorítmos e heurísticas que obtenham controle da forma mais automática possível.

Podemos dizer que um sistema computacional típico possui três componentes, a saber [Lucena, 1990] :

1. Componente descritivo.
2. Componente procedimental.
3. Componente teleológico (para quê ?).

O conjunto de sentenças lógicas bem que pode formar o componente descritivo. Idealmente, os mecanismos de prova automática deveriam ser capazes de obter os componentes procedimentais. De fato, o princípio de resolução trabalhando com cláusulas gerais deve ser capaz de gerar qualquer procedimento. Infelizmente vimos que isto não é prático. A solução procurada até hoje foi diminuir a expressividade do componente descritivo e deixar a cargo do programador completar a parte do componente procedimental que o sistema não conseguisse realizar por falta de descrição adequada. O programador usa, para isto, seus conhecimentos da finalidade do sistema, ou seja, usa o componente teleológico. Aparentemente parte das pesquisas atuais estão indo na direção de dar aos algoritmos de prova um componente teleológico que lhes permita contribuir de modo mais significativo para o componente procedimental [Yamaki, 1990; de Barros, 1991]. Com isto surgiram as linguagens lógicas com restrição, as quais discutiremos a seguir.

1.6 Programação por Restrições.

Vamos examinar a função das restrições na programação lógica através de exemplos. Seja um programa para calcular o fatorial escrito em Prolog. Tal programa tem a seguinte forma:

```
fat(0,1) :- !.
fat(N,F) :- N1 is N - 1, fat(N1, F1), F is N * F1.
```

É fácil ver, por ele, que Prolog exige a introdução de uma enorme quantidade de controle (procedimentos). O corte que aparece na primeira sentença é um destes controles. A ordem das sentenças é outro mecanismo de controle. De fato, a primeira sentença não poderia vir depois da segunda pois o sistema entraria em um laço infinito por não encontrar a condição de parada. Em terceiro lugar, a ordem dos literais na cauda da segunda sentença deve ser a mostrada pois, no caso contrário, os dados necessários ao procedimentos, *N1 is N - 1* e *F is N * F1* não estariam disponiveis no momento em que se tornassem necessários. Além do que foi dito, o programa permite que se calcule o fatorial de um número, mas não que se ache um número cujo fatorial é dado. Em outras palavras, *?- fat(6,F)* teria resposta. Já *?- fat(N,120)* ficaria sem resposta. Isto porque, apesar de ter o aspecto de uma especificação, ou seja, de uma descrição do que é fatorial, nosso programa é na realidade um procedimento. Ele tem aspecto de componente descritivo, mas é na realidade um componente procedimental do esquema proposto por Lucena.

Uma linguagem baseada em restrições veria cada equação que aparece no programa como uma restrição matemática. Para ser preciso, cada componente não lógico é visto como uma restrição que deve ser obedecida durante a prova. O provador, ao encontrar a restrição, coloca-a em uma pilha sem alterá-la. Concorrentemente, (ou paralelamente) um sistema separado do programador tentaria verificar se as restrições estão sendo satisfeitas. Para isto, tal sistema resolveria as equações da pilha até onde isto fosse possível. Se as equações não forem sufientes para se encontrar o

valor de todas as variáveis, o sistema de satisfação de restrições consideraria que não há provas de que o provador está no caminho errado e deixá-lo-ia continuar. Se for possível resolver as equações, o sistema de satisfação de restrições acha os valores das variáveis e os fornece ao provador. Se, por outro lado, as restrições não forem satisfactíveis, o provador é forçado a mudar a direção da prova. Neste esquema, o programa de fatorial teria a forma:

```
fat(0,1).
fat(N,F) :- N >0, N1 = N-1, F = N*F1, fat(N1,F1).
```

Aí a ordem das sentenças ou dos literais da cauda não interessa. Além disso, podemos obter respostas para qualquer consulta abaixo:

```
?- fat(5,F).
?- fat(N,120).
?- fat(N,F).
```

Capítulo II. Prolog

2.1 Introdução.

Conforme vimos no Capítulo I, um programa lógico é uma sentença lógica a ser provada. Normalmente, tal sentença é representada em forma clausal e o provador de teoremas é baseado no princípio de resolução. No caso específico de Prolog, a linguagem é restrita às cláusulas de Horn e o provador é uma implementação da resolução SLD onde o critério de seleção de cláusulas é "de cima para baixo" e o de seleção de resolventes é "da esquerda para a direita". Para melhorar o desempenho dos programas Prolog além do obtido com todos os refinamentos feitos sobre a linguagem e o mecanismo de prova, permite-se ao programador introduzir controle através de mecanismos especiais.

Neste capítulo vamos tratar do desenvolvimento de um provador de teoremas com as características do Prolog que possa ser executado numa máquina von Neumann convencional e das transformações que podem ser feitas aos programas Prolog levando em consideração esse sistema de execução específico, isto é, a compilação desses programas para linguagens mais próximas da linguagem de máquina de um computador convencional.

Dentro desse contexto puramente procedimental, uma mudança de nomenclatura é conveniente. Assim, ao literal positivo presente em cada cláusula chamaremos de "cabeça da cláusula". A coleção de cláusulas cujos literais positivos, isto é, cujas cabeças, têm o mesmo nome e a mesma aridade (quantidade de argumentos) será chamada de "predicado" ou "procedimento". A cada literal negativo de uma cláusula daremos o nome de "meta" ou "chamada" e a coleção de literais negativos (metas) da cláusula será chamada de corpo da cláusula. O teorema a ser provado, por conter somente literais negativos, chamaremos também de "meta" ou "meta inicial".

2.2 Especificação de um Interpretador Prolog

A tarefa de um interpretador Prolog é, dada uma coleção de procedimentos e uma meta inicial, tentar chegar a uma refutação. O primeiro passo será portanto encontrar o predicado correspondente a esta meta. O seguinte procedimento (escrito em Prolog) faz isso :

```
clausulas(Meta, Clausulas) :-
        findall((Meta :- Corpo), clause(Meta,Corpo), Clausulas).
```

A primitiva *clause* obtem, uma a uma, as cláusulas cujas cabeças correspondem (unificam) à *Meta*[1] e a primitiva *findall* reune a coleção destas cláusulas na lista *Clausulas*. Se não houverem cláusulas que satisfaçam a *Meta*, esta lista ficará vazia.

O próximo passo será usar cada uma das cláusulas obtidas, de acordo com o critério de cima para baixo, para tentar satisfazer a meta :

```
tente(Meta, [(Cabeca :- Corpo)|Clausulas]) :-
        unifique(Meta, Cabeca),
        execute(Corpo) ;
        tente(Meta, Clausulas).

unifique(X, X).
```

Este procedimento não-determinístico (devido à presença do operador ';') escolhe a primeira cláusula da lista (a cláusula de cima) como candidata a satisfazer a *Meta* e tenta executar o seu *Corpo* se tiver conseguido unificar a *Cabeca* com a *Meta*. No caso da unificação ou da execução falhar, as cláusulas restantes da

---

Nota :

1 Ao encontrar uma cláusula unitária *clause* gera uma ocorrência da constante *true* no seu segundo argumento.

lista (as de baixo) são tentadas da mesma forma.

Resta, finalmente, executar as metas que compõem o corpo da cláusula escolhida da esquerda para a direita, ou seja, repetir os passos até agora efetuados com a meta inicial para a nova sequência de metas:

```
execute(true) :- !.
execute(Meta) :-
        separe(Meta, Esquerda, Direita),
        chame(Esquerda),
        execute(Direita).

chame(Meta) :-
        primitiva(Meta), !,
        call(Meta).
chame(Meta) :-
        clausulas(Meta, Clausulas),
        tente(Meta, Clausulas).

separe((Esq, Dir), Esq, Dir) :- !.
separe(Meta, Meta, true).
```

Com estes procedimentos fica completo o nosso interpretador. O predicado *execute* é na verdade o laço principal deste programa, o qual parará quando lhe for apresentada a constante *true* como meta. Outra metas serão desmembradas de tal forma que a primeira (a da esquerda) seja resolvida completamente e posteriormente sejam executadas as restantes (as da direita). Se a *Meta* escolhida corresponder a uma rotina intrínseca ao sistema de execução, êle é chamado; caso contrário, o predicado correspondente é procurado e as cláusulas que o compõem são tentadas uma a uma.

Estamos agora de posse de um interpretador Prolog, escrito em Prolog e, portanto, que pode ser executado através do auxílio de um outro interpretador Prolog escrito na linguagem de uma máquina

tipo von Neumann, ou então compilado para tal linguagem.

Esta especificação, embora elegante, é deficiente quando analisada em termos do nosso objetivo inicial que é o de descrever um sistema de execução para Prolog: nosso dever é justamente tornar explícitas, em termos de computação convencional, todas as operações que estão envolvidas. Sob esta ótica, são três as deficiências desta especificação :

1) É usado o predicado *tente* que é não-determinístico, e seu uso oculta o mecanismo de gerenciamento do retrocesso do Prolog.

2) O predicado *execute*, por ser duplamente recursivo, é explosivo em termos de consumo de memória.

3) As metas a serem provadas contêm variáveis para as quais podem ser geradas ocorrências ou que podem ser unificadas entre si. Estas operações e seu gerenciamento também ficam ocultos no procedimento *unifique*.

2.3 Otimização do Interpretador

Uma vez que a especificação dada, apesar de deficiente, é correta, nossa abordagem será a de transformá-la sintáticamente, para que, preservando sua semântica, tais deficiências sejam eliminadas.

As três características deficientes do nosso interpretador/especificação inicial dizem respeito a dois problemas inerentes à implementação de qualquer linguagem de programação: o controle do fluxo de execução e o gerenciamento da memória alocada para variáveis. Em Prolog, como em outras linguagens, estes dois problemas podem ser abordados separadamente para efeitos de explanação, embora numa implementação, as duas questões fiquem fortemente acopladas. Assim sendo, vamos tratar inicialmente da questão do controle de fluxo de execução e posteriormente da alocação de memória. O fluxo de execução, por sua vez, será abordado em duas etapas: as sucessivas "chamadas" de metas até chegar a uma solução do problema e os eventuais retrocessos que

possam ocorrer quando a alternativa escolhida falha.

Por razões práticas de legibilidade, vamos tratar primeiro da questão da explosividade gerada pela dupla recursividade do predicado *execute*. Este problema é fácilmente resolvido lembrando que todo algoritmo recursivo pode ser transformado num algoritmo iterativo equivalente através da introdução de uma pilha auxiliar, isto é,

```
r(true) :- !.
r(X) :-
        p(X,A,B),
        r(A),
        r(B).
```

é equivalente a :

```
r(true,[]) :- !.
r(true,[A|B]) :- !,
        r(A,B).
r(X,C) :-
        p(X,A,B),
        r(A,[B|C]).
```

Mas este novo programa não é recursivo ? Sim, na sua forma. Não no seu comportamento. Sendo que todas as chamadas recursivas são determinísticas e são as últimas metas a serem executadas nas suas respectivas cláusulas, elas se comportam como se fossem chamadas iterativas que não gastam memoria adicional.

Nossa especificação transformada fica como segue :

```
%
%    execute1 - elimina a recursividade de execute
%
execute1(true,[]) :- !.
execute1(true, [Cont|Continuacao]) :- !,
     execute1(Cont, Continuacao).
```

```
execute1(Meta, Continuacao) :-
        separe(Meta, Esquerda, Direita),
        chame1(Esquerda, [Direita|Continuacao]).
chame1(Meta, Cont) :-
        clausulas(Meta, Clausulas),
        tente1(Meta, Clausulas, Cont).
tente1(Meta, [(Cabeca :- Corpo)|Clausulas], Continuacao) :-
        unifique(Meta, Cabeca),
        execute1(Corpo, Continuacao).
tente1(Meta, [Clausula|Clausulas], C) :-
        tente1(Meta, Clausulas, C).
```

Esta transformação nos sugere a primeira estrutura de dados para a implementação do interpretador: uma pilha dos pontos de continuação, isto é, os locais aonde deve prosseguir a execução do programa uma vez que sejam satisfeitas as metas que compõem as cláusulas. Quando o corpo de uma cláusula é composto por mais de uma meta, é feita a tentativa de satisfazer aquela que se encontra mais à esquerda e as restantes são empilhadas (terceira cláusula de *execute*1). Por outro lado, se a cláusula for unitária ou se todas as metas de seu corpo já foram satisfeitas, procede-se a satisfazer a meta que se encontra no topo da pilha (segunda cláusula de *execute*1). Finalmente, quando não houverem mais metas a serem satisfeitas no corpo da cláusula nem na pilha de continuação, o algoritmo termina (primeira cláusula de *execute*1).

Devemos observar que, devido ao seu não-determinismo, a nossa nova especificação ainda não é completamente iterativa. Apesar de que o predicado *execute*1 possa ser visto como um ciclo iterativo do tipo **while**, mesmo contendo uma chamada recursiva na sua segunda cláusula, o predicado *tente*1 é não-determinístico, e por isso precisa preservar sua natureza recursiva.

A questão do não-determinismo é um pouco mais complexa já que ela depende diretamente da estratégia utilizada para a alocação de variáveis: durante a execução de um programa, são geradas

ocorrências para as diversas variáveis como resultado de unificações bem sucedidas; no entanto, se alguma tentativa de unificação fracassar, deve ser iniciado um processo de "falha", o qual deve encontrar o último predicado para o qual ainda existem cláusulas candidatas e re-estabelecer o estado das variáveis àquele em que se encontravam quando este predicado foi chamado. Isto significa que quaisquer unificações que tiverem sido feitas desde a chamada até o instante da falha, e somente essas unificações, devem ser desfeitas, e as variáveis envolvidas devem voltar ao estado de "livres".

Para evitar um grau excessivo de detalhamento, vamos deixar a análise destas operações de mais baixo nivel para mais adiante quando falarmos de compilação. Por ora vamos isolar o não determinismo em um só procedimento e prosseguir com o estudo do problema da unificação. Teremos então:

```
chame1(Meta, Cont) :-
        clausulas(Meta, Clausulas), !,
        tente1((Cabeca :- Corpo), Clausulas),
        unifique(Meta, Cabeca),
        execute1(Corpo, Cont).

tente1(X, [X|_]).
tente1(X, [_|Y]) :-
        tente1(X, Y).
```

A grande virtude desta pequena reorganização do nosso programa é que o não-determinismo ficou completamente circunscrito pelo procedimento *tente1*. Dada uma lista qualquer, inicialmente é retornado o elemento que se encontra na sua cabeça; em caso de retrocesso, as unificações são desfeitas (pelo Prolog subjacente ao nosso interpretador) e o controle é passado para a segunda cláusula de *tente1*, a qual realiza uma chamada recursiva descartando a cabeça da lista.

Até agora, conseguimos realizar algumas transformações sobre

o nosso interpretador Prolog que, além de torná-lo mais eficiente, nos permitiram compreender melhor seu funcionamento interno. De fato, tais mudanças não requereram de nossa parte quaisquer considerações sobre a representação interna dos programas que estão sendo interpretados e dos termos por eles manipulados. Isso graças ao fato de estarmos implementando o nosso interpretador tambem em Prolog, o que nos permite delegar as considerações de nivel inferior ao sistema subjacente. Convem agora considerar algumas modificações sobre as cláusulas que compõem nossos programas de tal forma que possamos tornar mais explícito o funcionamento do interpretador.

Inicialmente, vamos considerar a representação das variáveis. Em vez de nos referirmos a elas simplesmente pelos seus nomes (começados por letras maiúsculas), vamos representá-las por estruturas do tipo $v(N)[2], onde N é um número inteiro, maior do que 0, atribuido a cada variável diferente da cláusula. Assim sendo, vamos supor tambem que o primeiro termo presente no corpo de cada cláusula é um número natural NVars que corresponde à quantidade total de variáveis daquela cláusula. A seguir mostramos um programa para encontrar os descendentes de uma pessoa e sua correspondente representação:

```
des(X,Y) :-                    |  des($v(1), $v(2)) :-
     des0(X,Y).                |           2, des0($v(1), $v(2)).
des(X,Y) :-                    |  des($v(1),  $v(2))  :-
     des0(X,Z),                |           3, des0($v(1), $v(3)),
     des(Z,Y).                 |           des($v(3), $v(2)).
                               |
des0(g2,g1).                   |  des0(g2, g1) :- 0, true.
des0(g3,g2).                   |  des0(g3, g2) :- 0, true.
```

Nota :

[2] Estamos supondo, sem perda de generalidade, que o programa original não contem estruturas deste mesmo tipo.

Da simples observação de qualquer programa Prolog, sabemos que o escopo de qualquer variável é a cláusula. Se fizéssemos uma analogia com linguagens de programação convencionais, poderiamos dizer que toda variável é local à cláusula que a contem. Não existem variáveis globais em Prolog. Existe uma diferença, porém, em relação às outras linguagens: o escopo da variável Prolog não é o procedimento completo, mas tão somente a cláusula (isto é vital para garantir o funcionamento do não determinismo conforme foi dito anteriormente).

O escopo de uma variável é importante porque nos permite definir, em tempo de execução, quais são as variáveis às quais se tem acesso e quais não. Como Prolog é uma linguagem que permite recursão, devemos dizer com mais precisão que o escopo das variáveis é na verdade a ativação da cláusula que as contem, já que num programa recursivo, cada cláusula pode estar ativa mais de uma vez e as variáveis correspondentes a uma ativação são completamente independentes das variáveis de uma outra ativação.

Finalmente, devemos observar que o único mecanismo de atribuição de valores às variáveis é a unificação que, por sua vez, funciona como mecanismo de passagem de parâmetros de uma cláusula para outra. Nesse contexto, a unificação engloba a passagem de parâmetros por valor e a passagem por referência, dependendo do estado da variável (livre ou com ocorrência) no momento da chamada. Nas implementações mais puras de Prolog as variáveis não são tipadas, ou seja, elas podem ser substituidas por ocorrências de qualquer constante, função, ou até mesmo outra variável.

Levando tudo isso em conta, podemos alterar nosso interpretador como segue:

```
chame2(Meta, Cont) :-
        clausulas(Meta, Clausulas), !,
        tente((Cabeca :- NVars, Corpo), Clausulas),
        functor(Vars, vars, NVars),
        unifique2(Meta, Cabeca, Vars),
        execute2(Corpo, Cont, Vars).
```

Uma vez que foi obtida uma cláusula candidata de acordo com o novo formato, *chame2* aloca espaço em memória para as variáveis presentes na cláusula através da primitiva *functor*. Com esse ambiente criado, *unifique2* se encarrega de fazer a passagem de parâmetros entre *Meta* e *Cabeca* gerando as primeiras ocorrências de *Vars*. Se houver sucesso, *execute2* é chamado para executar o *Corpo* da cláusula utilizando o ambiente ativo de variáveis. O procedimento *execute2* é definido assim:

```
execute2(true, [], _) :- !.
execute2(true, [amb(C, V)|Cont], _) :- !,
        execute2(C, Cont, V).
execute2(Meta, Cont, Vars) :-
        separe(Meta, Esquerda, Direita),
        unifique2(NEsquerda, Esquerda, Vars),
        chame2(NEsquerda, [amb(Direita, Vars)|Cont]).
```

O terceiro argumento de *execute2* é justamente o ambiente de variáveis criado por *chame2* para a cláusula que está em execução. Depois de isolar a meta mais à esquerda, chamamos novamente o procedimento *unifique2* com o propósito de montar a nova chamada, isto é, substituir as eventuais estruturas $v(N) presentes na meta, pelas suas ocorrências atuais. Em seguida, *chame2* é chamada com esta nova meta, empilhando, além das metas restantes, o ambiente de variáveis que está sendo utilizado.

O procedimento *unifique2* tem uma finalidade dupla conforme dissemos anteriormente: gerar ocorrências para as variáveis de uma cláusula, a partir de argumentos da meta que a ativou, e extrair tais ocorrências do ambiente para gerar uma nova meta. Vejamos a

sua definição:

```
unifique2(Meta, Cabeca, Vars) :-
        functor(Cabeca, N, A),
        functor(Meta, N, A),
        Cabeca =.. [N|CArgs],
        Meta =.. [N|MArgs],
        unifique_args(MArgs, CArgs, Vars, []).

unifique_args([Arg|MArgs], ['$v'(N)|CArgs], Vars, Aux) :- !,
        arg(N, Vars, Arg),
        unifique_args(MArgs, CArgs, Aux).
unifique_args([ConsM|MArgs], [ConsC|CArgs], Vars, Aux) :-
        atomic(ConsC), !, ConsM = ConsC,
        unifique_args(MArgs, CArgs, Vars, Aux).
unifique_args([FunM|MArgs], [FunC|CArgs], Vars, Aux) :- !,
        functor(FunC, N, A),
        functor(FunM, N, A),
        FunC =.. [N|CCArgs],
        FunM =.. [N|MMArgs],
        unifique_args(MMArgs, CCArgs, Vars, [t(MArgs,CArgs)|Aux])
unifique_args([], [], Vars, [t(MArgs, CArgs)|Aux]) :- !,
        unifique_args(MArgs, CArgs, Vars, Aux).
unifique_args([], [], _, []).
```

Depois de verificar que seus dois primeiros argumentos são estruturas cujos nomes e aridades são iguais, *unifique2* coloca os argumentos de cada um deles em listas e chama o procedimento *unifique_args* passando a lista de argumentos da meta, a lista de argumentos da cabeça, o ambiente de variáveis e uma lista vazia, que servirá como pilha auxiliar para a unificação de estruturas.

As três primeiras cláusulas de *unifique_args* realizam a unificação propriamente dita sendo que a primeira trata das variáveis, a segunda das constantes e a terceira das estruturas. Nesta última, é feita a verificação de nome e aridade e a unificação prossegue com os argumentos da estrutura, empilhando os

argumentos restantes da meta e da cláusula para unificação posterior[3]. As duas cláusulas restantes tratam da continuação da unificação, caso hajam argumentos de estruturas ainda pendentes, e da parada, caso todos os argumentos já tenham sido unificados.

A versão atual do interpretador já é bastante completa e nos dá uma idéia clara do que está envolvido numa implementação real. Porém, antes de continuar com o seu detalhamento interno, vamos introduzir duas alterações que vão melhorar drásticamente seu desempenho. A primeira delas está relacionada com o que foi dito anteriormente a respeito de chamadas recursivas cujo comportamento é o de um laço iterativo. Isto acontece quando a chamada recursiva é a última meta de uma cláusula correspondente a uma meta determinística, isto é, não há mais cláusulas candidatas e todas as metas anteriores à chamada recursiva foram satisfeitas determinísticamente. Ou seja:

rec :- . . . . .
rec :- . . . . .
rec :- m_1, m_2, ..., m_n, rec.

A chamada recursiva rec está na última cláusula do procedimento e m_1, m_2, ..., m_n foram satisfeitas determinísticamente. Este efeito tambem ocorre quando a chamada recursiva é precedida por um corte, mesmo que ela não esteja na última cláusula:

rec :- . . . . .
rec :- . . . . . , !, rec.
rec :- . . . . .

Nota :
[3] Poderiamos ter optado por empilhar os argumentos internos das estruturas e continuar a unificação com os argumentos posteriores. Embora o resultado seja o mesmo, escolhemos esse caminho por ser mais conveniente para a compilação.

Nestes casos podemos alocar as variáveis de tal forma que elas possam ser descartadas quando for feita a chamada recursiva, assim como os argumentos passados para o procedimento. Este tipo de recursão é chamado de **recursão de cauda** e é na verdade o resultado que viabiliza o uso de provadores automáticos de teoremas como linguagens de programação. A recursividade de cauda é um caso específico da combinação de dois casos mais gerais: a **otimização de última chamada**, que nos permite liberar o espaço alocado para as variáveis e evitar o empilhamento inútil de continuações quando é chamada a última meta de uma cláusula qualquer e a **otimização de chamadas determinísticas** que permite evitar empilhamento de argumentos quando o procedimento contém somente uma cláusula ou estamos prestes a utilizar a última cláusula de um procedimento não determinístico.

Podemos obter o efeito de otimização de última chamada acrescentando uma cláusula ao procedimento *execute2* e eliminando o procedimento *separe*:

```
execute2(true, [], _) :- !.
execute2(true, [amb(C, V)|Cont], _) :- !,
     execute2(C, Cont, V).
execute2((Esquerda, Direita), Cont, Vars) :- !,
     unifique2(NEsquerda, Esquerda, Vars),
     chame2(NEsquerda, [amb(Direita,Vars)|Cont]).
execute2(Meta, Cont, Vars) :-
     unifique2(NMeta, Meta, Vars),
     chame2(NMeta, Cont).
```

A otimização de chamadas determinísticas é obtida acrescentando uma cláusula ao procedimento *tente*:

```
tente(X, [Y]) :- !, X = Y.
tente(X, [X|_]).
tente(X, [_|Y]) :-
     tente(X, Y).
```

A segunda alteração surge a partir de uma observação dos procedimentos *execute2* e *chame2*: logo depois de criar o ambiente para as variáveis e unificar a meta com a cabeça da cláusula, é chamado *execute2* que, se a cláusula não for unitária, chamará novamente *unifique2* para construir a primeira meta a ser chamada. Podemos reorganizar o programa de tal forma que as chamadas a *unifique2* sejam otimizadas. A seguir apresentamos o código completo dos procedimentos *execute3* e *chame3*:

```
execute3(Meta, Cabeca, true, Vars, []) :- !,
        unifique2(Meta, Cabeca, Vars).
execute3(Meta, Cabeca, true, Vars, Cont) :- !,
        unifique2(Meta, Cabeca, Vars),
        continue(Cont).
execute3(Meta, Cabeca, (Esq, Dir), Vars, Cont) :- !,
        unifique2((Meta, NEsq), (Cabeca, Esq), Vars),
        chame3(NEsq, [amb(Dir, Vars)|Cont]).
execute3(Meta, Cabeca, Corpo, Vars, Cont) :- !,
        unifique2((Meta, NMeta), (Cabeca, Corpo), Vars),
        chame3(NMeta, Cont).

continue([amb((Esq, Dir), Vars)|Cont]) :- !,
        unifique2(NEsq, Esq, Vars),
        chame3(NEsq, [amb(Dir,Vars)|Cont]).
continue([amb(Meta, Vars)|Cont]) :- !,
        unifique2(NMeta, Meta, Vars),
        chame3(NMeta, Cont).

chame3(Meta, Cont) :-
        clausulas(Meta, Clausulas), !,
        tente((Cabeca :- NVars, Corpo), Clausulas),
        functor(Vars, vars, NVars),
        execute3(Meta, Cabeca, Corpo, Vars, Cont).
```

Para concluir a discussão sobre interpretação de Prolog vamos extender a idéia de modificar a representação das variáveis para programas inteiros. Desta forma, será possível melhorar ainda

mais o desempenho do interpretador e o detalhamento de sua especificação. Em contrapartida, a transformação dos programas escritos em Prolog para esta nova representação irá requerer de auxílio de uma ferramenta. A partir daí iniciaremos a discussão sobre compilação de Prolog.

2.4 Compilação de Prolog

É fácil verificar que os procedimentos *unifique2* e *unifique_args* teriam um comportamento muito mais eficiente se seus argumentos já viessem "linearizados" de tal forma que eles não tivessem que se preocupar com a estrutura sintática de tais argumentos e se ocupassem sómente com a sua semântica. Em outras palavras, podemos adiantar uma grande parte do trabalho da unificação realizando os testes a respeito do tipo do termo que está sendo unificado com antecedência. Dessa forma, as seguintes transformações, propostas em [Bowen, 1983], nos serão uteis:

variáveis: $v(N) ——> [var, N]

constantes: C ——> [const, C]

estruturas: $f(t_1, \ldots, t_n)$
——> [functor, f/n, t_1, ..., t_n, pop]

Com isto os procedimentos *unifique2* e *unifique_args* podem ser substituidos por um novo procedimento *unifique3*:

```
unifique3([Arg|MArgs], [var, N|CArgs], Vars, Aux) :- !,
        arg(N, Vars, Arg),
        unifique3(MArgs, CArgs, Vars, Aux).
unifique3([C|MArgs], [const, C|CArgs], Vars, Aux) :- !,
        unifique3(MArgs, CArgs, Vars, Aux).
unifique3([Arg|MArgs], [functor, F/A|CArgs], Vars, Aux) :- !,
        functor(Arg, F, A),
        Arg =.. [F|FArgs],
        unifique3(FArgs, CArgs, Vars, [MArgs|Aux]).
```

```
unifique3([], [pop|CArgs], Vars, [MArgs|Aux]) :- !,
     unifique3(MArgs, CArgs, Vars, Aux).
unifique3([], [], _, []).
```

Aplicando a mesma idéia às cláusulas e procedimentos teremos as seguintes transformações:

metas: $m(t_1, \ldots, t_n) \longrightarrow [t_1, \ldots, t_n, call, m/n]$

cláusulas: $c(c_1, \ldots, c_n) :- m_1, \ldots, m_m.$

$\longrightarrow$ clause(NVars, $[c_1, \ldots, c_n$, enter, ..., call, m_1, ..., call, m_m, exit])

procedimentos:

$c_1.$
:
$c_n.$

$\longrightarrow$ procedure(Chave, $[c_1, \ldots, c_n]$).

onde Chave é uma estrutura nome/aridade correspondente às cláusulas que compõem o procedimento.

O programa do nosso exemplo ficaria assim:

```
procedure(des/2,
          [clause(2, [var, 1, var, 2, enter,
                      var, 1, var, 2, call, des0/2,
                      exit]),
           clause(3, [var, 1, var, 2, enter,
                      var, 1, var, 3, call, des0/2,
                      var, 3, var, 2, call, des/2,
                      exit])]).
procedure(des0/2,
          [clause(0, [const, g2, const, g1, exit]),
           clause(0, [const, g3, const, g2, exit])]).
```

Os procedimentos *execute3* e *unifique3* podem agora ser combinados:

```
execute4([Arg|MArgs], [var, N|PC], Vars, Cont, Aux) :- !,
    arg(N, Vars, Arg),
    execute4(MArgs, PC, Vars, Cont, Aux).
execute4([C|MArgs], [const, C|PC], Vars, Cont, Aux) :- !,
    execute4(MArgs, PC, Vars, Cont, Aux).
execute4([Arg|MArgs], [functor, F/A|PC], Vars, Cont, Aux)  :- !,
    functor(Arg, F, A),
    Arg =.. [F|FArgs],
    execute4(FArgs, PC, Vars, Cont, [MArgs|Aux]).
execute4([], [pop|PC], Vars, Cont, [MArgs|Aux]) :- !,
    execute4(MArgs, PC, Vars, Cont, Aux).
execute4([], [enter|PC], Vars, Cont, []) :- !,
    execute4(Args, PC, Vars, Cont, Args).
execute4([], [call, Proc|PC], Vars, Cont, Args) :- !,
    chame4(Proc, Args, [amb(PC, Vars)|Cont]).
execute4([], [exit], _, [amb(PC, Vars)|Cont], []) :- !,
    execute4(Args, PC, Vars, Cont, Args).
execute4([], [exit], _, [], []).
```

A otimização de última chamada tambem pode ser introduzida observando que a instrução *call*, correspondente à última meta, empilha informação inútil que será descartada pela instrução *exit* quando o controle retornar. Isto pode ser fácilmente sanado se a última meta for executada através de uma nova instrução que não empilha tal informação e não é seguida de um *exit*:

```
execute4([], [depart,Proc], Vars, Cont, Args) :- !,
    chame4(Proc, Args, Cont).
```

O mesmo raciocínio se aplica quando se executa uma instrução *functor* que corresponde ao último argumento da cabeça da cláusula ou de um outro termo qualquer. Ela empilha uma lista vazia de argumentos que será posteriormente desempilhada por *pop* sem nenhuma utilidade. Novamente, uma instrução extra resolve o nosso problema:

```
execute4([Arg], [lastfunctor, F/A|PC], Vars, Cont, Aux) :- !,
        functor(Arg, F, A),
        Arg =.. [F|FArgs],
        execute4(FArgs, PC, Vars, Cont, Aux).
```

Desta forma continuamos tendo recursividade de cauda já que o interpretador garante a otimização de chamada determinística.

O procedimento *chame4*, que é o ponto de entrada do nosso interpretador, fica assim:

```
chame4(Proc, Args, Cont) :-
        procedure(Proc, Clausulas), !,
        tente(clause(NVars, PC), Clausulas),
        functor(Vars, vars, NVars),
        execute4(Args, PC, Vars, Cont, []).
```

O programa para traduzir programas escritos em Prolog para o código aceito por este interpretador é o seguinte:

```
compile(File) :- open(H,File,r),
        read(H,Term),
        process(Term,nil,Clauses,Clauses,H),
        close(H).

process(end_of_file,Key,[],Clauses,_) :- !,
            assert(procedure(Key,Clauses)).
process(Term,OKey,ClausesO,Clauses,H) :-
            translate(Term,Clause,Key),
            store(OKey,Key,Clause,ClausesO,Clauses,H).

store(Key,Key,Clause,[Clause|ClausesO],Clauses,H) :- !,
            read(H,Term),
            process(Term,Key,ClausesO,Clauses,H).
store(OKey,NKey,Clause,[],OClauses,H) :-
            assert(procedure(OKey,OClauses)),
            read(H,Term),
            process(Term,NKey,Clauses,[Clause|Clauses],H).
```

```
translate((Head :- Body),clause(NVars,Args),N/A) :- !,
      functor(Head,N,A),
      head(Head,Args,Vars0,Tail0),
      body(Body,Goals,Vars0,Vars,[exit]),
      length(Vars,NVars),
      optimise([enter|Goals],Tail0).

translate(Head,clause(NVars,Args),N/A) :- !,
      functor(Head,N,A),
      head(Head,Args,Vars,[exit]),
      length(Vars,NVars).

head(Head,LArgs,Vars,Tail) :-
      Head =.. [N|Args],
      arguments(Args,[],LArgs,[],Vars,Tail).


body(true,Tail,Vars,Vars,Tail) :- !.
body(Body,Goal,Vars0,Vars,Tail) :-
      split(Body,First,Rest),
      tcall(Rest,Call,Tail,NTail),
      functor(First,F,A),
      First =.. [F|Args],
      arguments(Args,[],Goal,Vars0,Vars2,[Call,F/A|Tail1]),
      body(Rest,Tail1,Vars2,Vars,NTail).

split((First,Rest),First,Rest) :- !.
split(One,One,true).

tcall(true,depart,_,[]) :- !.
tcall(_,call,T,T).

optimise([enter,exit],[exit]) :- !.
optimise(Body,Body).
```

```
arguments([Arg|Arest],Astack,[var,Pos|PC],Vars0,Vars,Tail) :-
        var(Arg), !,
        tbl_member(Arg,Vars0,Vars1,1,Pos),
        arguments(Arest,Astack,PC,Vars1,Vars,Tail).
arguments([Arg|Arest],Astack,[const,Arg|PC],Vars0,Vars,Tail) :-
        atomic(Arg), !,
        arguments(Arest,Astack,PC,Vars0,Vars,Tail).
arguments([Arg|Arest],Astack,[TFuntor,F/A|PC],Vars0,Vars,Tail) :-
        functor(Arg,F,A), !,
        tfuntor(Arest,Astack,TFuntor,Fstack),
        Arg =.. [F|Fargs],
        arguments(Fargs,Fstack,PC,Vars0,Vars,Tail).
arguments([],[Arest|Astack],[pop|PC],Vars0,Vars,Tail) :- !,
        arguments(Arest,Astack,PC,Vars0,Vars,Tail).
arguments([],[],Tail,Vars,Vars,Tail).

tbl_member(Arg,[],[Arg],N,N) :- !.
tbl_member(Arg,[Arg1|Args],[Arg1|Args],N,N) :- Arg == Arg1, !.
tbl_member(Arg,[Arg1|Args],[Arg1|NArgs],N,M) :- NN is N + 1,
        tbl_member(Arg,Args,NArgs,NN,M).

tfuntor([],Astack,lastfunctor,Astack) :- !.
tfuntor(Arest,Astack,functor,[Arest|Astack]).
```

É interessante observar que o conjunto de instruções obtido deriva-se "naturalmente" da sintaxe dos programas Prolog uma vez que se compreende o seu funcionamento. Como de costume, para reforçar a idéia de que a leitura procedimental de programas lógicos os aproxima de programas convencionais, a unidade de compilação é o procedimento[4]. Daqui para a frente, vamos

---

Nota :

[4] Alguns consideram a cláusula como unidade de compilação de Prolog para efeitos de compilação incremental. Embora esta abordagem seja valiosa para áreas como a Engenharia do Conhecimento, ela não beneficia a compreensão da compilação.

considerar mais alguns refinamentos do nosso conjunto de instruções e, consequentemente, do compilador e do sistema de execução. Uma vez que estamos agora tratando de um sistema sequencial convencional, o uso de uma linguagem de programação de mais baixo nivel se torna mais conveniente. Portanto, no que segue usaremos a linguagem C para codificar nossas rotinas.

Já temos uma boa idéia de como seria o procedimento de execução de programas Prolog e das suas necessidades de gerenciamento de memória. Sabemos ser necessário alocar uma região de memória para armazenar o código correspondente aos procedimentos e uma outra região para instalar a pilha de variáveis e continuações. Assim, ao receber uma chamada, o interpretador:

1. Localiza o procedimento.
2. Tenta executar cada uma das cláusulas anotando se existem ou não mais cláusulas candidatas.
   2.1 Aloca espaço para as variáveis da cláusula.
   2.2 Executa cada uma das instruções da cláusula.
      2.2.1 Em caso de falha durante alguma operação de unificação:
         2.2.1.1 Desfaz as unificações feitas até o momento.
         2.2.1.2 "Retrocede" ao passo 2 para tentar a próxima cláusula.
      2.2.2 Ao chamar um novo procedimento empilha (se for necessário) o ponto de continuação e as variáveis e prossegue com o passo 1.
      2.2.3 Ao terminar a execução de uma cláusula (instrução *exit*) desempilha as variáveis e ponto de continuação que eventualmente existirem e continua (passo 2.2 de alguma cláusula préviamente ativada). Caso a pilha esteja vazia, termina a execução com sucesso.
3. Caso não hajam mais cláusulas candidatas termina a execução com fracasso.

Seria interessante que o passo 2 deste algoritmo fosse tambem linear, isto é, que o gerenciamento do não determinismo tambem fosse controlado por instruções no programa e não mais pelo interpretador. Para isto, basta introduzir "marcas" no inicio de cada cláusula de um procedimento não deterministico indicando o ponto de continuação em caso de falha na unificação:

```
c1.                          proc: try_me_else C2
c2.          ------->                 <código para c1>
:                              C2: retry_me_else C3
cn.                                  <código para c2>
                                      :
                               Cn: trust_me_else fail
                                     <código para cn>
```

Estas instruções ajudam o interpretador no gerenciamento do não determinismo como segue:

1. A instrução *try_me_else* indica que o interpretador vai iniciar a execução de um procedimento não determinístico. Com essa informação ele providencia para que os argumentos que estão sendo passados sejam preservados para uso da próxima cláusula em caso de falha; inicializa uma pilha auxiliar (chamada de trilha) onde serão guardados os endereços de quaisquer variáveis para as quais forem geradas ocorrências a partir deste momento para que tais ocorrências sejam desfeitas em caso de falha; e obtem a posição de inicio da próxima cláusula na área de código para reiniciar a execução em caso de falha. Estas três informações, consideradas de procedimento, podem ser armazenadas também na pilha num registro chamado de **ponto de escolha**.

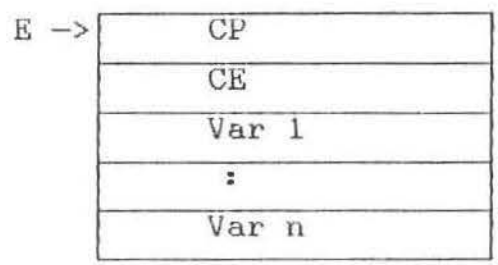
2. A instrução *retry_me_else* marca o inicio de uma cláusula intermediária de um procedimento não deterministico. Portanto, já existe um ponto de escolha correspondente a este procedimento e a única operação que o interpretador deve realizar é atualizar a posição de inicio da próxima cláusula.

3. A instrução *trust_me_else* indica o inicio da última cláusula de um procedimento não deterministico. O ponto de escolha atual pode ser descartado, uma vez que, se houver falha, a

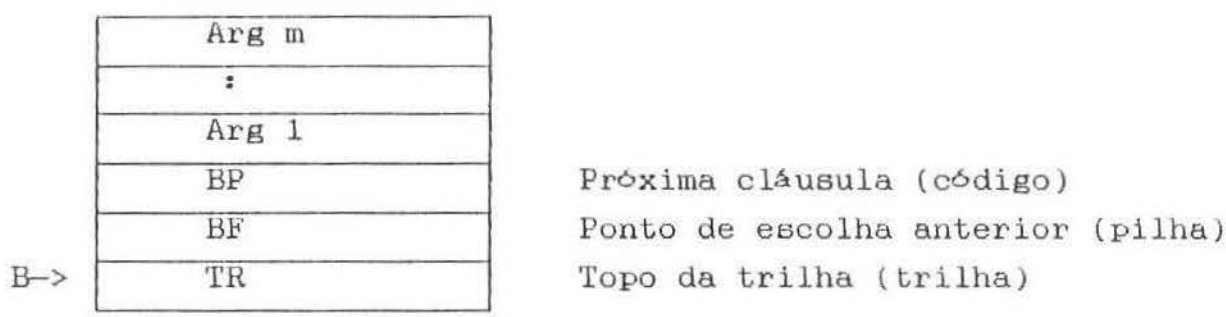
execução deve recomeçar em algum outro procedimento não deterministico chamado antes, para o qual deve existir um ponto de escolha em algum lugar mais profundo da pilha. Este ponto de escolha se torna então o ponto de escolha corrente.

Os registros criados na pilha são distribuidos da seguinte maneira:

Ponto de continuação/ambiente:



Ponto de escolha:



Os registradores *E* e *B* apontam ao ponto de continuação e ao ponto de escolha correntes respectivamente.

Nosso programa exemplo ficaria assim:

```
des/2:    try_me_else L1
          allocate 2
          var 1
          var 2
          enter
          var 1
```

```
                var 2
                deallocate
                depart des0/2
L1:             trust_me_else fail
                allocate 3
                var 1
                var 2
                enter
                var 1
                var 3
                call des0/2
                var 3
                var 2
                deallocate
                depart des/2

des0/2:         try_me_else L2
                const g2
                const g1
                exit
L2:             trust_me_else fail
                const g3
                const g2
                exit
```

As instruções *allocate* e *deallocate* se encarregam agora de reservar espaço na pilha para as variáveis.

As instruções de unificação (*const*, *functor*, *var*), como vimos anteriormente, são ambivalentes e sua função é determinada pelo lugar da cláusula onde elas estão inseridas: antes do *enter*, isto é, na cabeça da cláusula, elas "desmontam" uma meta e servem para analisar seu conteúdo; depois do *enter*, isto é, no corpo da cláusula, elas "montam" os argumentos da próxima meta que será executada. Se criarmos uma instrução para cada tipo de operação teremos um novo interpretador mais eficiente com as seguintes instruções:

get_const	*put_const*
get_struct	*put_struct*
get_var	*put_var*

Além disto, se levarmos em conta que a primeira aparição de uma instrução de unificação de variável sempre corresponderá a uma variável livre e as subsequentes aparições corresponderão à variável já unificada com algum outro valor, podemos usar as instruções *get_var* e *put_var* para as primeiras aparições e definir as seguintes instruções para as subsequentes:

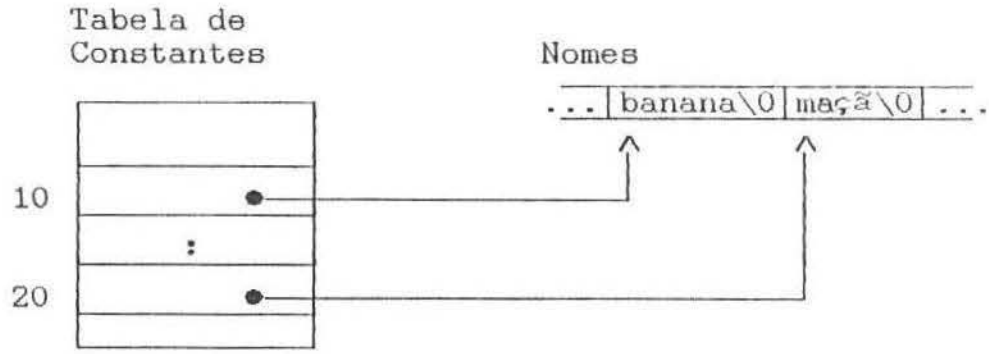
get_value	*put_value*

Algumas considerações sobre a representação interna dos termos se tornam agora convenientes. A distribuição dos registros nos sugere um tratamento uniforme dos termos do Prolog, no sentido de que se utilize uma "célula" do registro para acessar um termo qualquer. O uso de uma célula (um dado estruturado) é necessário porque não estamos considerando verificação de tipos como parte da nossa linguagem e, por isso, ao gerar uma ocorrência para uma variável precisamos especificar, além do seu valor, o seu tipo (constante, variável ou estrutura). Assim, cada termo do Prolog será representado internamente por um par

<tipo, valor>

denominado célula.

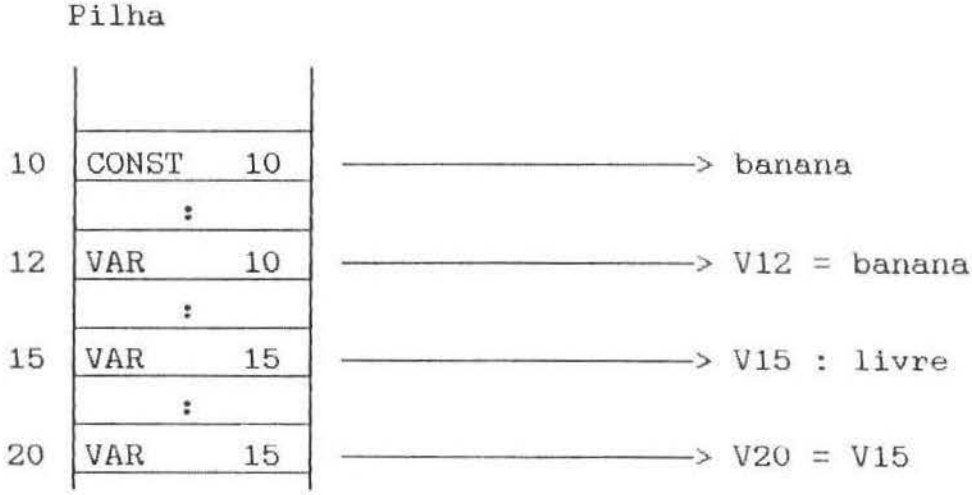
No caso das constantes, podemos imaginar que existe uma tabela onde são armazenadas todas as constantes e que o valor presente na célula corresponde à posição da constante em questão dentro da tabela. Além de uma economia de espaço, a unificação de constantes se transforma numa simples comparação/atribuição de inteiros.



<CONST, 10> = banana

<CONST, 20> = maçã

Variáveis podem ser representadas por células cujo tipo seja VAR e cujo valor seja a posição da variável na pilha:



As estruturas merecem um cuidado especial, uma vez que requerem mais do que uma célula e a escolha de um método para sua representação terá grande impacto no desempenho do interpretador/sistema de execução tanto em termos de consumo de memória como de tempo de execução. Na primeira implementação bem sucedida de Prolog [Warren, 1977], foi utilizada a técnica conhecida como "compartilhamento de estruturas" [Boyer, 1972], para a representação de estruturas. Aqui, cada estrutura é representada por um par

<esqueleto,ambiente>

onde o primeiro elemento indica o local aonde está descrita a parte fixa da estrutura, e o segundo indica o local aonde estão armazenadas as ocorrências geradas para as variáveis presentes em tais estruturas. A idéia por trás desta representação é a de minimizar o consumo de memória, mantendo somente uma cópia dos componentes não variáveis da estrutura (funtor e constantes) e alocando espaço novo somente para a representação das variáveis contidas na estrutura. Mais tarde [Clocksin, 1980; Bruynooghe, 1980], viu-se que uma representação mais direta, na qual cada nova ocorrência de uma estrutura fosse representada por uma cópia completa, proporcionava um ganho substancial de velocidade de execução com perdas pouco significativas (medidas empiricamente) de eficiência de consumo de memória. Este segundo método de representação, chamado "cópia de estruturas" é utilizado em todas as implementações mais recentes de Prolog [Bowen, 1983; Bowen, K. 1985], inclusive no "Novo Engenho de Prolog" de Warren [Warren, 1983].

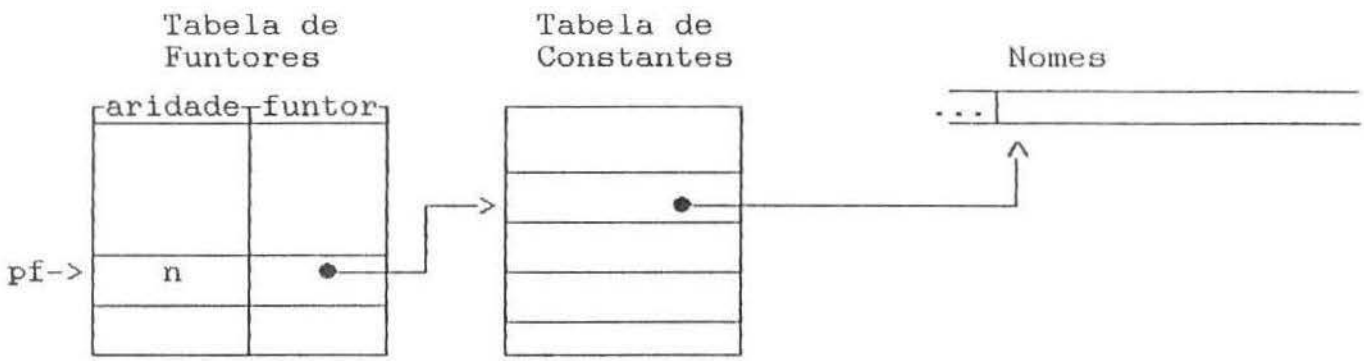
Poderiamos representar uma estrutura por uma sequência de células na qual a primeira nos indicaria o funtor e a aridade da estrutura e as seguintes representariam os seus argumentos internos. Este esquema, porém, é problemático: ao serem passadas como argumentos para outros procedimentos, as variáveis podem ser unificadas com estruturas nos procedimentos chamados. Em muitos casos, tais ocorrências precisam ser mantidas até o final da execução do programa, o que impediria que utilizássemos a técnica de otimização de última chamada descrita anteriormente. Para sanar este problema, foi sugerida a utilização de uma nova região de memória, chamada de "heap" ou "pilha global" para armazenar as estruturas. Desta forma, uma estrutura será representada pela célula

<ESTR,pos_heap>

onde pos_heap é a posição de uma sequência de células na pilha global da seguinte forma:

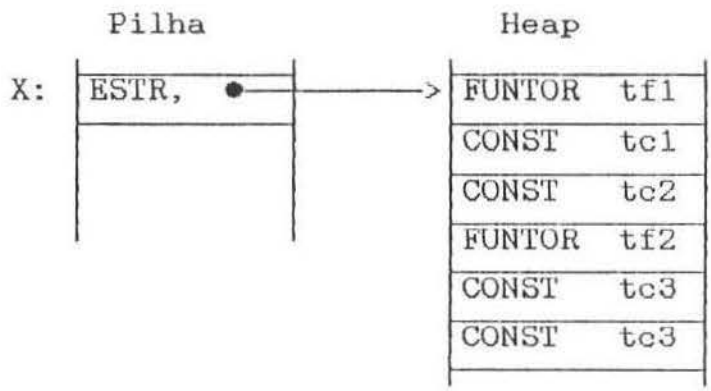
<FUNTOR,pos_funtor>
célula arg 1
:
célula arg n

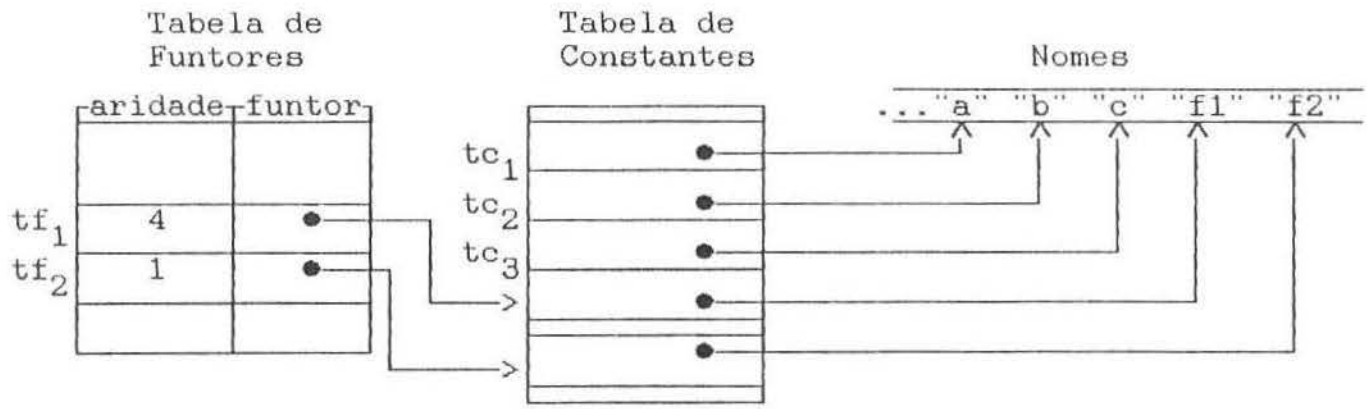
e pos_funtor é a posição correspondente ao funtor numa tabela de funtores:



Assim, quando uma variável presente numa estrutura tiver ocorrência gerada, ela será "globalizada", podendo portanto ser eliminada da pilha (local) sem riscos.

Suponhamos que em um certo momento, o interpretador encontra-se no seguinte estado:





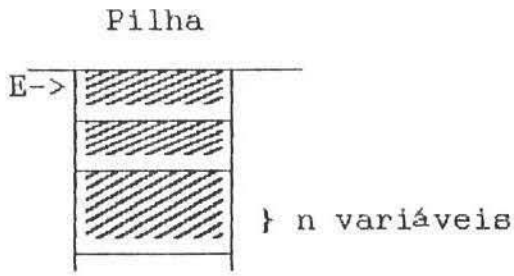
O termo representado pela posição assinalada como X na pilha é f1(a,b,f2(c),c). Para construí-lo poderiamos usar, por exemplo, a seguinte sequência de intruções:

```
:
:
put_struct      tf1
put_const       tc1
put_const       tc2
put_struct      tf2
put_const       tc3
pop
put_const       tc3
pop
:
:
```

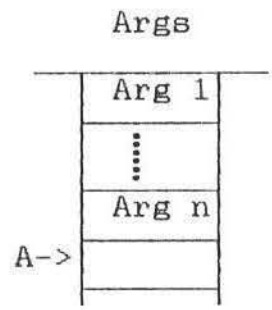
sendo que a instrução *put_struct* deve criar a célula <ESTR,pos> na pilha, somente para as estruturas externas. Para as internas, sua função será sómente criar a célula <FUNTOR,pos> no heap. Isto pode ser implementado mediante marcas auxiliares que indicam o contexto atual de execução do interpretador.

Podemos agora detalhar um pouco melhor o fluxo de execução de um procedimento Prolog:

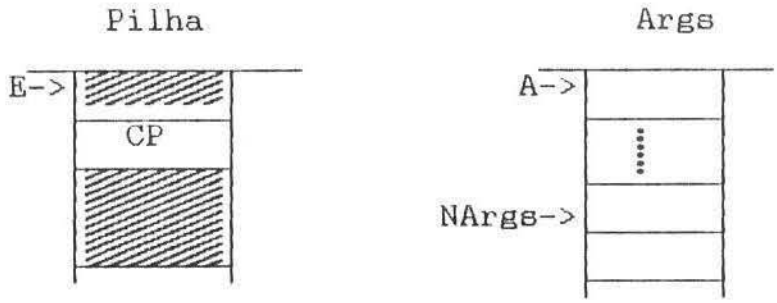
1. Aloca-se espaço para as variáveis e a continuação da meta inicial na pilha local através da instrução *allocate*.



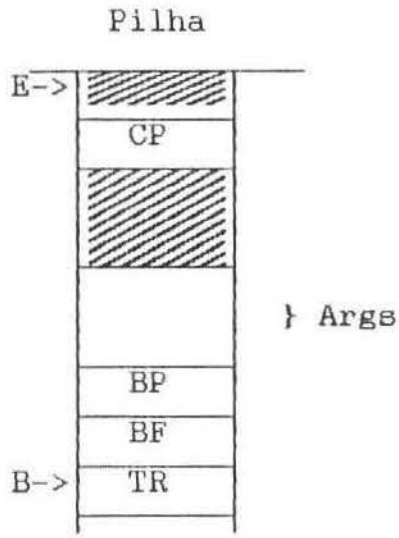
2. Os argumentos da meta inicial são armazenados numa pilha de argumentos através de instruções tipo *put*.



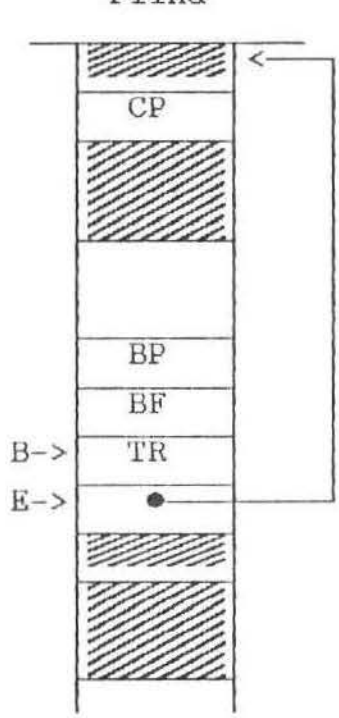
3. Através de um *call* é atualizado o ponto de continuação, localizado o inicio do novo procedimento a ser executado, e a pilha de argumentos é preparada para a unificação.



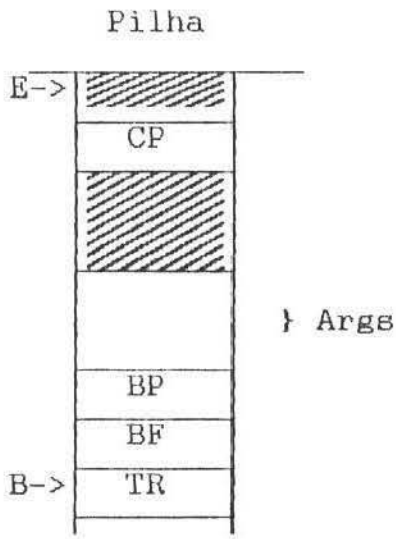
4. Se o procedimento for não determinístico, a primeira instrução será *try_me_else*, que criará um novo ponto de escolha na pilha copiando os argumentos e inicializando os outros parâmetros de controle para o caso de falha.



5. Em seguida será executada uma instrução *allocate* que criará um novo ponto de continuação na pilha local e reservará espaço para o ambiente de variáveis.

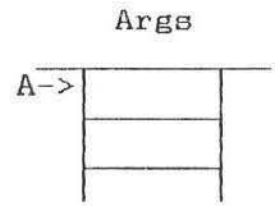


6. Os argumentos passados serão unificados através de instruções tipo *get*. Se ocorrer uma falha o sistema procurará o ponto de escolha mais recente (através do registrador B), restaurará os argumentos, variáveis e registradores ao estado em que se encontravam ao iniciar o procedimento.

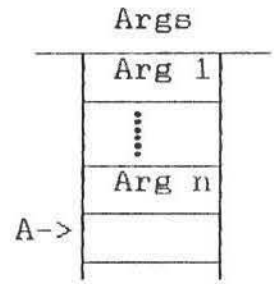


7. Se a cláusula em execução for unitária, será executada uma instrução *deallocate* que fará com que o registrador E seja restabelecido para o ambiente da cláusula chamadora e, em seguida será executada uma instrução *exit* que obterá o ponto de continuação apartir do ambiente restabelecido.

8. Se a cláusula não for unitária, será executada a instrução *enter* que reinicializará a pilha de argumentos preparando-a para uma nova meta.



9. Os argumentos para a nova meta são empilhados com instruções *put*.



10. Se a nova meta for a última da cláusula, será executada uma instrução *deallocate* que libera o ambiente de variáveis e continuação desta cláusula. Em seguida é executada a instrução *depart* que é semelhante a *call* exceto que o ponto de continuação não é atualizado.

É interessante analisar o efeito da interação dos valores dos registradores E e B (ambiente corrente e ponto de escolha corrente, respectivamente): quando o ambiente for mais recente que o ponto de escolha (ou seja, E aponta para uma posição mais próxima ao topo da pilha do que B), significa que a cláusula que está sendo executada atualmente é determinística. A consequência importante disso é que, ao executar a instrução *deallocate*, o ambiente será efetivamente descartado possibilitando a reutilização do espaço na pilha (otimização de última chamada).

Por outro lado, se o ponto de escolha for mais recente estamos numa situação em que alguma meta de alguma cláusula já foi satisfeita, porém não deterministicamente, e a presença do ponto de escolha no topo da pilha impede que o ambiente da cláusula seja descartado ao executar um *deallocate*. Isto é necessário já que, mesmo depois de terminada, uma cláusula pode ser "ressucitada" em caso de falha para tentar satisfazer novamente as metas para as quais ainda existirem alternativas.

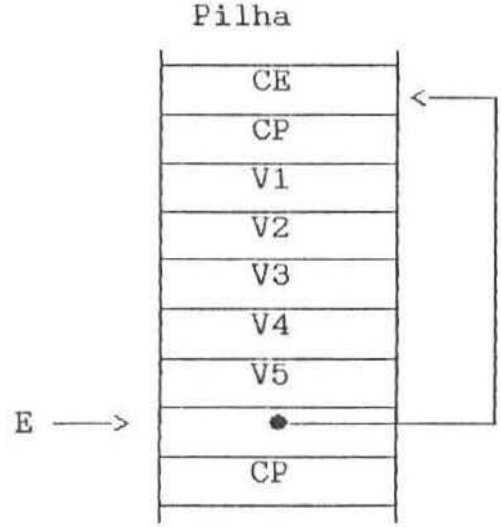
Seja como for, a criação de ambientes e pontos de escolha na pilha deve levar em conta qual o tipo de registro que está no topo para que não seja destruída informação.

Três observações são pertinentes no sentido de melhorar o desempenho deste sistema de execução. A primeira delas é que toda cláusula tem no seu começo uma instrução *allocate*, mesmo que ela seja unitária ou tenha somente uma meta no seu corpo. Nestes casos, estaremos criando inutilmente um ponto de continuação e um ambiente que será rápidamente descartado. Em vez disso poderiamos suprimir as instruções *allocate* e *deallocate* destas cláusulas e

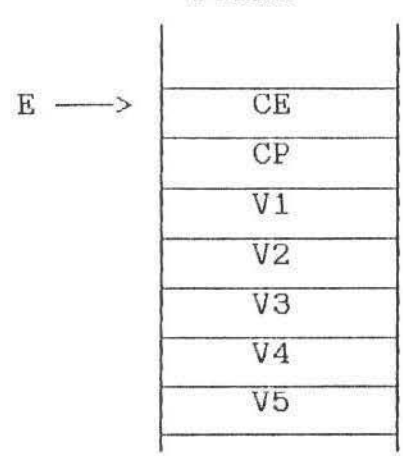
utilizar, por exemplo, a pilha de argumentos para armazenar temporáriamente as variáveis nela presentes. Esta idéia pode também ser extendida para cláusulas não unitárias com mais de uma meta no seu corpo. Apesar de ser necessária a inclusão das instruções *allocate* e *deallocate* nessas cláusulas, as variáveis podem ser classificadas em **temporárias** e **permanentes**, sendo que sómente as últimas ficarão armazenadas na pilha. Além disso, podemos numerar as variáveis permanentes de tal forma que elas possam ser descartadas aos poucos, mesmo antes de terminar a execução da cláusula. Warren [Warren, 1983] chamou este método de liberação de variáveis de "poda de ambientes". Para implementá-la, o comportamento das instruções *call*, *allocate* e *deallocate* deverá ser ligeiramente alterado: em lugar de utilizar um parâmetro com o número de variáveis na instrução *allocate*, ele é colocado na instrução *call* da cláusula chamadora, indicando quantas variáveis devem ser preservadas; *allocate* acessará este valor para criar o novo ponto de continuação e *deallocate* simplesmente restabelecerá o ponto de continuação anterior:

```
            call xpto/4, 5
            :
xpto/4:     allocate
            :
            deallocate
```

Ao iniciar a execução de *xpto/4*, a instrução *allocate* obterá o número de variáveis da cláusula chamadora (neste caso 5) e criará o novo ponto de continuação:



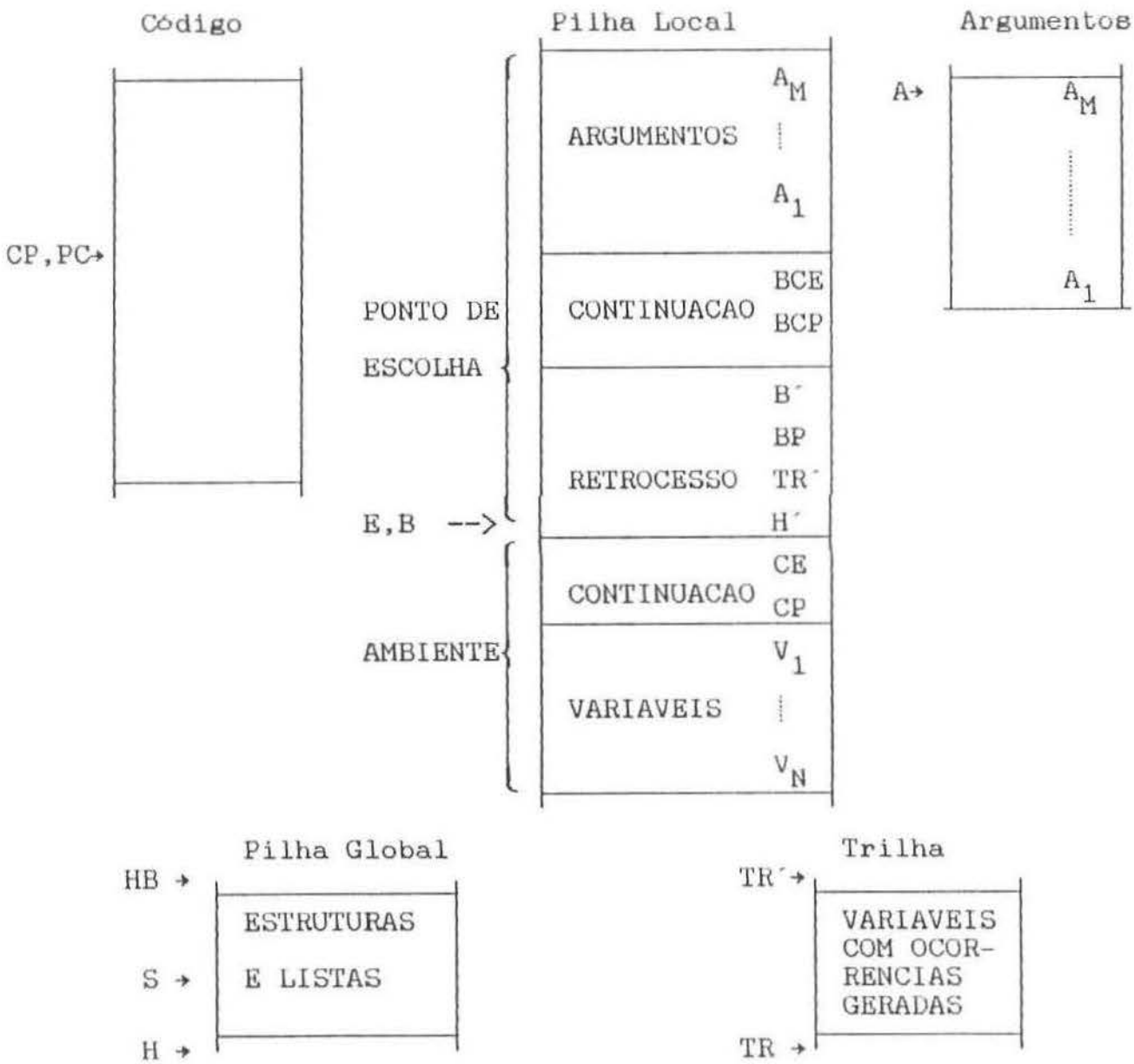
Por sua vez, a única função de *deallocate* será restabelecer o ambiente anterior:



A segunda observação se refere ao gerenciamento de pontos de escolha dentro da pilha. Ao criá-los, a instrução *try_me_else* copia todos os argumentos da chamada para a pilha e as instruções *retry_me_else* e *trust_me_else fail* restauram tais argumentos a partir da pilha em caso de falha. O que propomos é retirar os pontos de escolha da pilha local e transformar a pilha de argumentos numa pilha de escolha, ou seja, os argumentos são empilhados pelas instruções *put* e, se for executada uma instrução *try_me_else*, tais argumentos são considerados parte integrante do novo ponto de escolha que está sendo criado. Isto evita uma grande quantidade de cópia de argumentos. A unificação procede normalmente com as instruções *get* retirando os argumentos do próprio ponto de escolha. As posteriores instruções *put* empilharão os novos argumentos fora do ponto de escolha uma vez que a instrução *try_me_else* pode estabelecer o novo inicio da pilha de argumentos/pontos de escolha.

Warren propõe que não se meçam esforços no sentido de minimizar os deslocamentos de argumentos. Com este novo esquema, tais esforços podem ser dispensados já que em procedimentos não determinísticos sempre haverá deslocamentos, embora em caso de falha a recuperação dos argumentos é instantânea: ganha-se portanto uma simplificação no compilador e uma melhora na execução.

MÁQUINA ABSTRATA DE WARREN



P.

Instruções *get*
proceed

P :- Q.

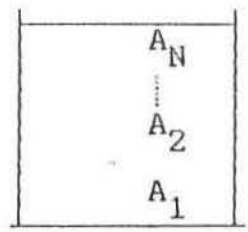
Instruções *get*
Instruções *put*
execute Q

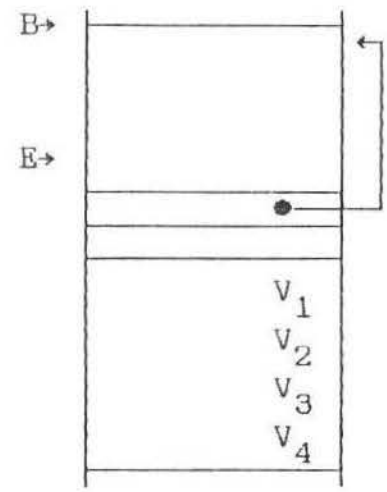
P :- Q, R.

allocate
Instruções *get*
Instruções *put*
call Q, N
Instruções *put*
deallocate
execute R

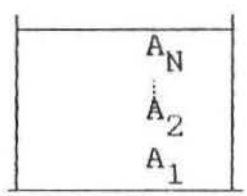
CHAMADA DE UM PROCEDIMENTO NÃO-DETERMINÍSTICO (WAM)

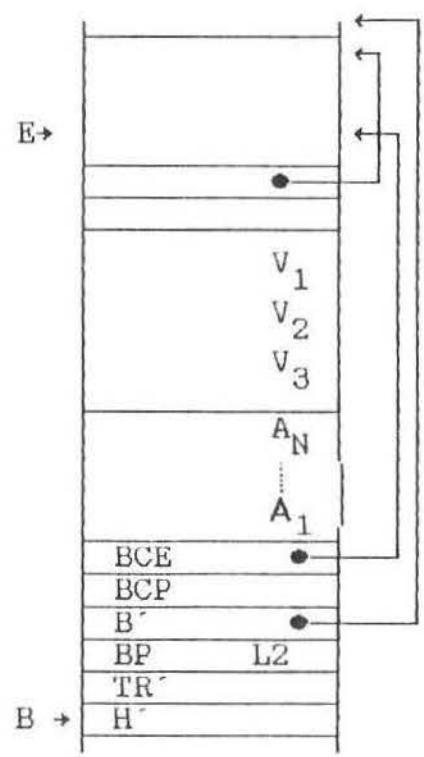
```
        PUT   A1
        PUT   A2
        ⋮
        PUT   AN
```



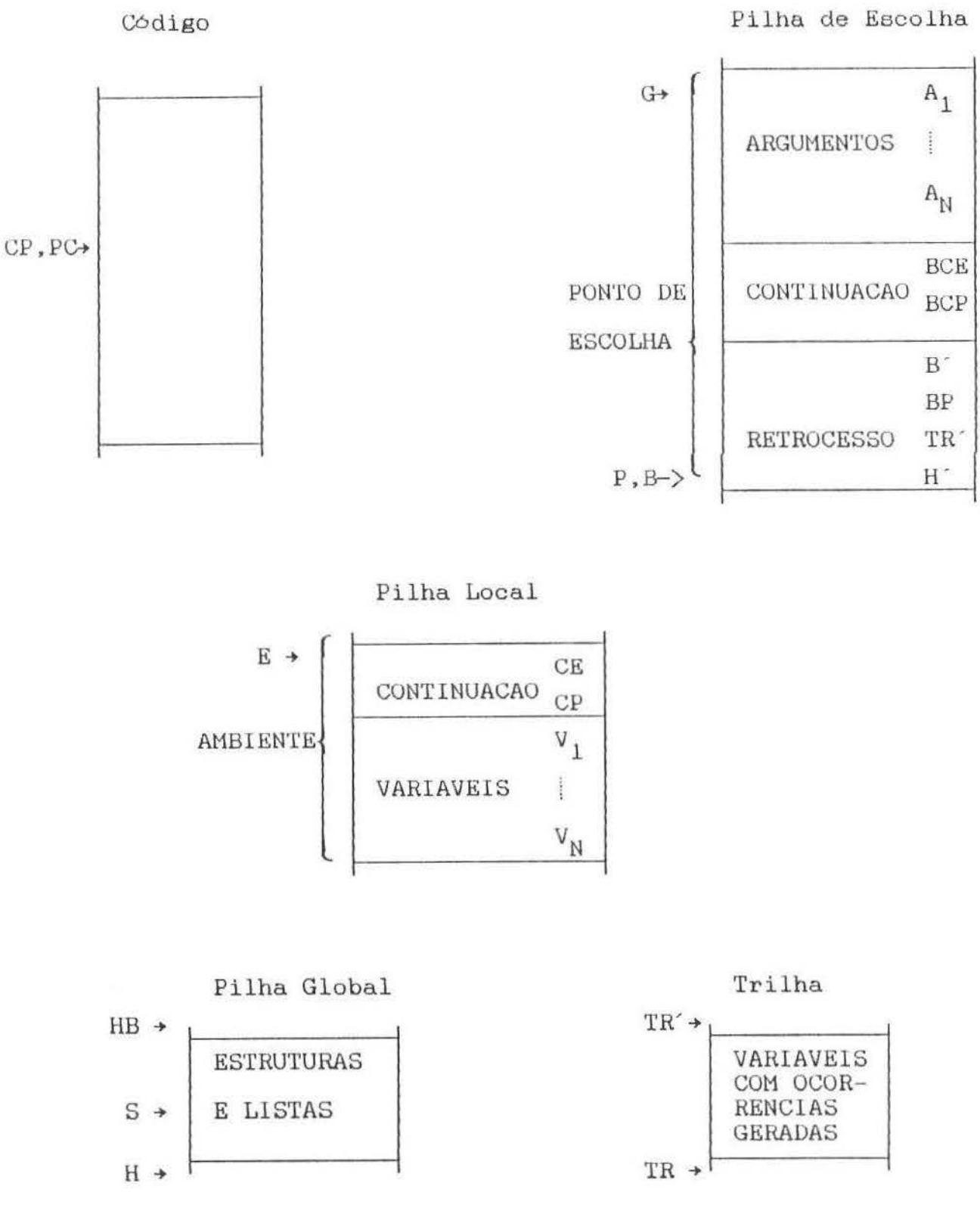



```
        CALL PROC, 3        CP ← L1
L1:     PUT A1
        ⋮
PROC:   TRY_ME_ELSE L2
```

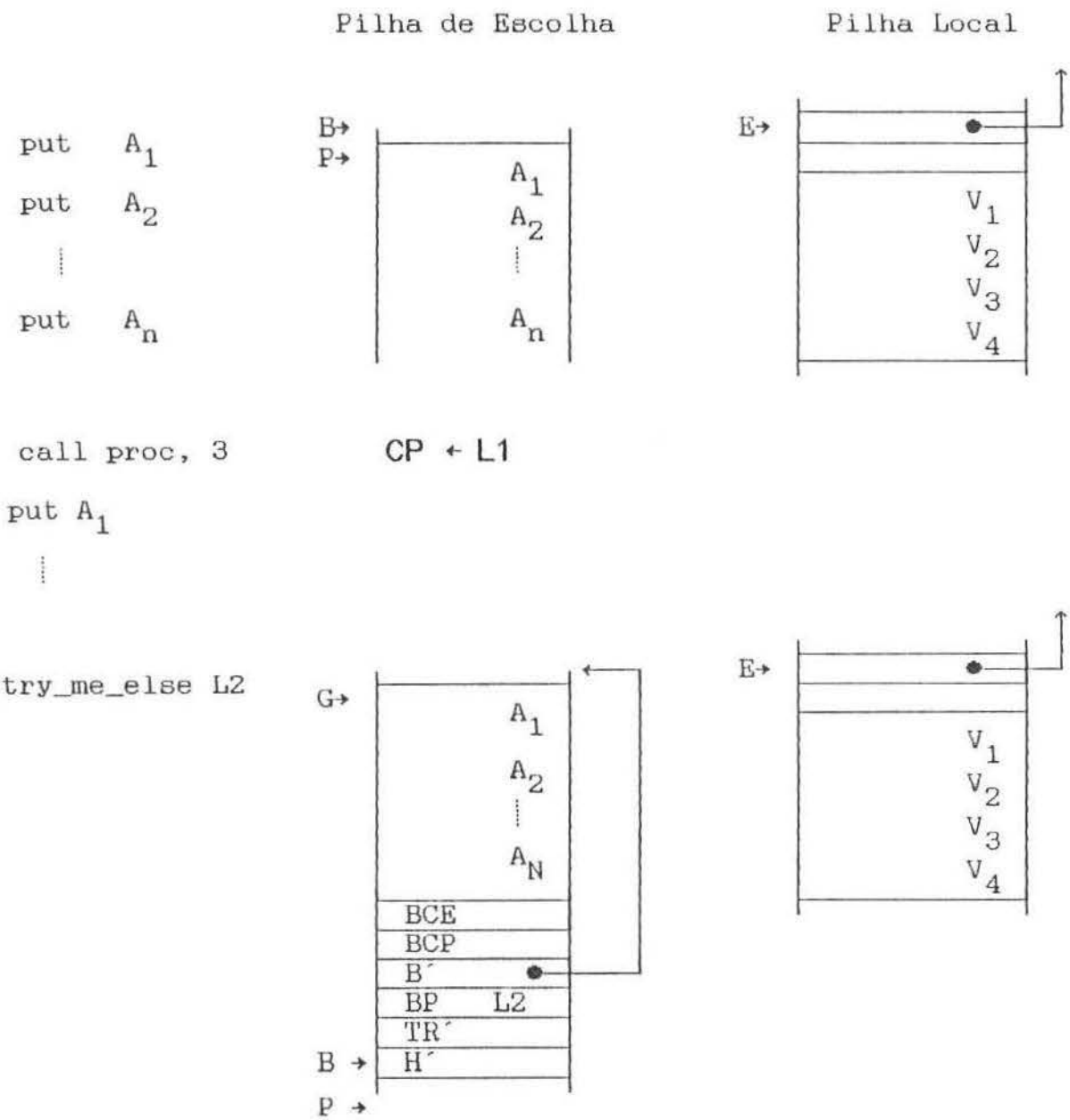




NOVA MAQUINA PROLOG



CHAMADA DE UM PROCEDIMENTO NÃO-DETERMINÍSTICO



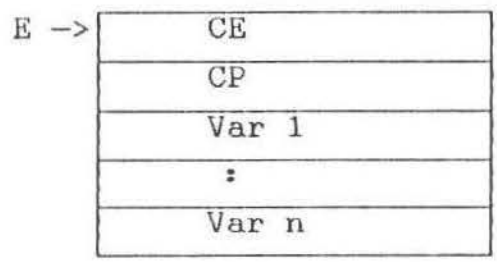
Em terceiro lugar, mais alguns comentários sobre o gerenciamento de estruturas. Como dissemos anteriormente, a instrução *put_struct* (e consequentemente, *put_laststruct*, *get_struct* e *get_laststruct*) tem um comportamento sensível ao contexto de execução do interpretador. Além disso, para obter a otimização de última chamada, foi necessária a criação das versões "*last*" que evitam a geração de instruções *pop* subsequentes. Isto é resultado da opção que fizemos de empilhar argumentos externos e prosseguir a unificação com os argumentos internos das estruturas. Na Máquina Abstrata de Warren (WAM) [Warren, 1983], a opção foi diferente: ao unificar os argumentos da cabeça da cláusula, as estruturas externas são tratadas imediatamente, porém através de instruções de unificação específicas que trabalham com o heap em vez da pilha; se houverem estruturas internas elas são "salvas" em variáveis temporárias e tratadas posteriormente. No corpo da cláusula, no entanto, as estruturas externas são criadas pela instrução *put_struct* como antes e seus argumentos pelas mesmas instruções de unificação específicas; para as estruturas internas também são usadas variáveis temporárias que podem estar unificadas ou livres para posterior unificação com tais estruturas internas. A vantagem desta abordagem é que embora tenhamos um conjunto maior de instruções, o comportamento de cada uma delas é mais homogêneo assim como a representação interna das estruturas. O conjunto de instruções usado para o tratamento de estruturas é o seguinte:

get_structure F
put_structure F
unify_constant C
unify_variable V
unify_value V

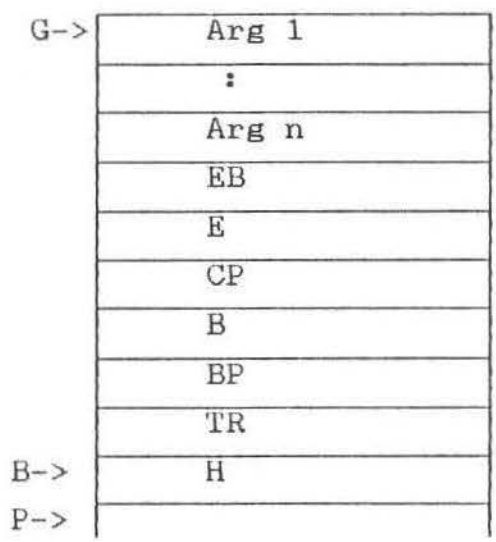
Em resumo, temos no nosso sistema de execução os seguintes registradores:

E : inicio do ambiente atual na pilha local.
B : final do ponto de escolha atual na pilha de escolha.
H : última posição ocupada na pilha global.
TR : última posição ocupada na trilha.
EB : último ambiente determinístico na pilha local.
PC : próxima instrução a ser executada na área de código.
G : primeiro argumento da chamada atual na pilha de escolha. Usado pelas instruções *get*.
P : primeiro argumento da próxima chamada na pilha de escolha. Usado pelas instruções *put*.

A distribuição dos ambientes é a seguinte:



e a distribuição do pontos de escolha é a seguinte :



2.5 Exemplo de Execução

Para fixar idéias, vamos discutir passo a passo a execução de um exemplo. Em seguida, vamos mostrar um emulador, em C, de uma máquina capaz de executar o exemplo. Seja o programa abaixo.

```
avo(X,Y) :- pai(X,Z), pai(Z,Y).

pai(a,b).
pai(c,d).
pai(b,c).
```

O programa acima é compilado da seguinte maneira:

```
11   avo/2:    allocate
12             get_var    V1, A1
13             get_var    V2, A2
14             put_value  V1, A1
15             put_var    V3, A2
16             call       pai/2, 3
17             put_value  V3, A1
18             put_value  V2, A2
19             deallocate
20             depart pai/2

21   pai/2:    try_me_else    L1
22             get_const a, A1
23             get_const b, A2
24             exit
25   L1:       retry_me_else  L2
26             get_const c, A1
27             get_const d, A2
28             exit
29   L2:       trust_me_else  fail
30             get_const b, A1
31             get_const c, A2
32             exit
```

Suponhamos agora que se faça a seguinte consulta :

```
?- avo(X,Y).
```

que, traduzida, ficaria assim:

```
1        allocate
2        put_var   V1, A1
3        put_var   V2, A2
4        call      avo/2
5        stop
```

Os passos da execução podem ser resumidos da seguinte forma:

1 *allocate*

Coloca o registrador E=0, fazendo-o apontar para a primeira posição livre da pilha local.

Local : []
Escolha : []
Trilha : []
E=0,G=-1,P=-1,PC=2,B=-1,EB=-1,TR=-1

---

2 *put_var V1, A1*

Local : [V2,_,_] — O prefixo V indica que a célula está ocupada por variável. O índice de V indica a atribuição da variável. V2 foi atribuída a ela mesma o que indica variável livre. O índice 2 de V2 é obtido somando-se 1+E ao índice V1 da instrução.

Escolha : [V2] — A posição a ser usada pelo argumento na pilha de escolha é obtida somando P ao índice A1 da instrução.

Trilha : []
E=0,G=-1,P=-1,PC=3,B=-1,EB=-1,TR=-1

---

3 *put_var V2, A2*

Local : [V3,V2,_,_]
Escolha : [V3,V2]
Trilha : []
E=0,G=-1,P=-1,PC=4,B=-1,EB=-1,TR=-1

---

4 *call avo/2,2*

Local : [V3,V2,5,_] 5 é o ponto de retorno. Foi obtido através de PC.

Escolha : [V3,V2]
Trilha : []
E=0,G=-1,P=-1,PC=11,B=-1,EB=-1,TR=-1

---

5 *allocate*

Local : [0,V3,V2,5,_] 0 é o apontador ao inicio do ambiente (registro de ativação) anterior. Foi obtido do registrador E. O novo E é calculado somando-se o E antigo com 2 e com o número de variáveis a serem preservadas.

Escolha : [V3,V2]
Trilha : []
E=4,G=-1,P=-1,PC=12,B=-1,EB=-1,TR=-1

---

6 *get_var V1, A1*

Local : [V2,_,0,V3,V2,5,_] A primeira variável do ambiente atual foi unificada com a primeira variável do ambiente anterior. Não é necessário empilhar esta unificação na trilha já que ela é mais recente que o último ponto de escolha e em caso de retrocesso ela desaparecerá

automáticamente.

Escolha : [V3,V2]

Trilha : []

E=4,G=-1,P=-1,PC=13,B=-1,EB=-1,TR=-1

---

7 *get_var V2, A2*

Local : [V3,V2,_,0,V3,V2,5,_]

Escolha : [V3,V2]

Trilha : []

E=4,G=-1,P=-1,PC=14,B=-1,EB=-1,TR=-1

---

8 *put_value V1, A1*

Local : [V3,V2,_,0,V3,V2,5,_]

Escolha : [V3,V2]

Os argumentos da chamada anterior podem ser destruídos uma vez que este procedimento é determinístico.

Trilha : []

E=4,G=-1,P=-1,PC=15,B=-1,EB=-1,TR=-1

---

9 *put_var V3, A2*

Local : [V8,V3,V2,_,0,V3,V2,5,_]

Escolha : [V8,V2]

Trilha : []

E=4,G=-1,P=-1,PC=16,B=-1,EB=-1,TR=-1

---

10 *call pai/2, 3*

Local : [V8,V3,V2,17,0,V3,V2,5,_]

Escolha : [V8,V2]

Trilha : []

E=4,G=-1,P=-1,PC=21,B=-1,EB=-1,TR=-1

---

11 *try_me_else L1*

Local : [V8,V3,V2,17,0,V3,V2,5,_]

Escolha : [-1,-1,25,-1,17,4,-1,V8,V2]

Trilha : []

E=4,G=-1,P=9,PC=22,B=8,EB=9,TR=-1

EB é calculado

somando o valor atual de E+2 e o número de variáveis no ambiente.
B é calculado somando B+7 ao número de argumentos (aridade) do procedimento.
P é calculado somando 1 ao novo B.

---

12 *get_const a, A1*

Local : [V8,V3,V2,17,0,V3,a,5,_]
Escolha : [-1,-1,25,-1,17,4,-1,V8,V2]
Trilha : [V2] V2 é empilhada na trilha porque corresponde a um ambiente mais antigo que o ponto de escolha atual.
E=4,G=-1,P=9,PC=23,B=8,EB=9,TR=0

---

13 *get_const b, A2*

Local : [b,V3,V2,17,0,V3,a,5,_]
Escolha : [-1,-1,25,-1,17,4,-1,V8,V2]
Trilha : [V8,V2]
E=4,G=-1,P=9, PC=24,B=8,EB=9,TR=1

---

14 *exit*

Local : [b,V3,V2,17,0,V3,a,5,_]
Escolha : [-1,-1,25,-1,17,4,-1,V8,V2]
Trilha : [V8,V2]
E=4,G=-1,P=9, PC=17,B=8,EB=9,TR=1 O endereço de retorno é obtido da pilha local.

---

15 *put_value V3, A1*

Local : [b,V3,V2,17,0,V3,a,5,_]
Escolha : [b,-1,-1,25,-1,17,4,-1,V8,V2]
Trilha : [V8,V2]
E=4,G=-1,P=9, PC=18,B=8,EB=9,TR=1

---

16 *put_value V2, A2*

Local : [b,V3,V2,17,0,V3,a,5,_]
Escolha : [V3,b,-1,-1,25,-1,17,4,-1,V8,V2]

Trilha : [V8,V2]
E=4,G=-1,P=9, PC=19,B=8,EB=9,TR=1

---

17 *deallocate*

Local : [b,V3,V2,17,0,V3,a,5,_]
Escolha : [V3,b,-1,-1,25,-1,17,4,-1,V8,V2]
Trilha : [V8,V2]
E=0,G=-1,P=9, PC=20,B=8,EB=9,TR=1

O ambiente atual não é descartado devido à presença do ponto de escolha. O ambiente anterior é reativado.

---

18 *depart pai/2*

Local : [b,V3,V2,17,0,V3,a,5,_]
Escolha : [V3,b,-1,-1,25,-1,17,4,-1,V8,V2]
Trilha : [V8,V2]
E=0,G=9,P=9, PC=21,B=8,EB=9,TR=1

---

19 *try_me_else L1*

Local : [b,V3,V2,17,0,V3,a,5,_]
Escolha : [-1,1,25,8,17,0,9,V3,b,-1,-1,25,-1,17,4,-1,V8,V2]
Trilha : [V8,V2]
E=0,G=9,P=18, PC=22,B=17,EB=9,TR=1

---

20 *get_const a, A1* FALHA !

Local : [b,V3,V2,17,0,V3,a,5,_]
Escolha : [-1,1,25,8,17,0,9,V3,b,-1,-1,25,-1,17,4,-1,V8,V2]
Trilha : [V8,V2]
E=0,G=9,P=18, PC=25,B=17,EB=9,TR=1

PC obtém o endereço da próxima cláusula a ser tentada. TR é igual ao valor salvo no ponto de escolha, portanto não há unificações que precisem ser desfeitas.

---

21 *retry_me_else L2*

Local : [b,V3,V2,17,0,V3,a,5,_]
Escolha : [-1,1,29,8,17,0,9,V3,b,-1,-1,25,-1,17,4,-1,V8,V2]

```
        Trilha  : [V8,V2]
        E=0,G=9,P=18,   PC=26,B=17,EB=9,TR=1
```

---

22 *get_const c, A1* FALHA !

```
        Local   :  [b,V3,V2,17,0,V3,a,5,_]
        Escolha :  [-1,1,29,8,17,0,9,V3,b,-1,-1,25,-1,17,4,-1,V8,V2]
        Trilha  : [V8,V2]
        E=0,G=9,P=18,   PC=29,B=17,EB=9,TR=1
```

---

23 *trust_me_else fail*

```
        Local   :  [b,V3,V2,17,0,V3,a,5,_]
        Escolha :  [V3,b,-1,-1,25,-1,17,4,-1,V8,V2]
        Trilha  : [V8,V2]
        E=0,G=9,P=9,   PC=30,B=8,EB=9,TR=1
```

---

24 *get_const b, A1*

```
        Local   :  [b,V3,V2,17,0,V3,a,5,_]
        Escolha :  [V3,b,-1,-1,25,-1,17,4,-1,V8,V2]
        Trilha  : [V8,V2]
        E=0,G=9,P=9,   PC=31,B=8,EB=9,TR=1
```

---

25 *get_const c, A2*

```
        Local   :  [b,V3,V2,17,0,c,a,5,_]
        Escolha :  [V3,b,-1,-1,25,-1,17,4,-1,V8,V2]
        Trilha  : [V3,V8,V2]
        E=0,G=9,P=9,   PC=32,B=8,EB=9,TR=1
```

---

26 *exit*

```
        Local   :  [b,V3,V2,17,0,c,a,5,_]
        Escolha :  [V3,b,-1,-1,25,-1,17,4,-1,V8,V2]
        Trilha  : [V3,V8,V2]
        E=0,G=9,P=9,   PC=5,B=8,EB=9,TR=1
```

---

27 *stop*

Termina com sucesso! Encontrou *avo(a,c)*.

---

28 Forçamos uma falha para procurar a próxima solução:

```
Local    :  [V8,V3,V2,17,0,V3,V2,5,_]
Escolha  :  [V3,b,-1,-1,25,-1,17,4,-1,V8,V2]
Trilha   : []
E=4,G=-1,P=9,   PC=25,B=8,EB=9,TR=1
```

---

29 *retry_me_else L2*

```
Local    :  [V8,V3,V2,17,0,V3,V2,5,_]
Escolha  :  [V3,b,-1,-1,29,-1,17,4,-1,V8,V2]
Trilha   : []
E=4,G=-1,P=9,   PC=26,B=8,EB=9,TR=-1
```

---

30 *get_const c, A1*

```
Local    :  [V8,V3,V2,17,0,V3,c,5,_]
Escolha  :  [V3,b,-1,-1,29,-1,17,4,-1,V8,V2]
Trilha   : [V2]
E=4,G=-1,P=9,   PC=27,B=8,EB=9,TR=0
```

---

31 *get_const d, A2*

```
Local    :  [d,V3,V2,17,0,V3,c,5,_]
Escolha  :  [V3,b,-1,-1,29,-1,17,4,-1,V8,V2]
Trilha   : [V8,V2]
E=4,G=-1,P=9,   PC=28,B=8,EB=9,TR=1
```

---

32 *exit*

```
Local    :  [d,V3,V2,17,0,V3,c,5,_]
Escolha  :  [V3,b,-1,-1,29,-1,17,4,-1,V8,V2]
Trilha   : [V8,V2]
E=4,G=-1,P=9,   PC=17,B=8,EB=9,TR=1
```

---

33 *put_value V3, A1*

```
Local    :  [d,V3,V2,17,0,V3,c,5,_]
Escolha  :  [V3,d,-1,-1,29,-1,17,4,-1,V8,V2]
Trilha   : [V8,V2]
E=4,G=-1,P=9,   PC=18,B=8,EB=9,TR=1
```

---

34 *put_value V2, A2*

Local : [d,V3,V2,17,0,V3,c,5,_]
Escolha : [V3,d,-1,-1,29,-1,17,4,-1,V8,V2]
Trilha : [V8,V2]
E=4,G=-1,P=9, PC=19,B=8,EB=9,TR=1

---

35 *deallocate*

Local : [d,V3,V2,17,0,V3,c,5,_]
Escolha : [V3,d,-1,-1,29,-1,17,4,-1,V8,V2]
Trilha : [V8,V2]
E=0,G=-1,P=9, PC=20,B=8,EB=9,TR=1

---

36 *depart pai/2*

Local : [d,V3,V2,17,0,V3,c,5,_]
Escolha : [V3,d,-1,-1,29,-1,17,4,-1,V8,V2]
Trilha : [V8,V2]
E=0,G=-1,P=9, PC=21,B=8,EB=9,TR=1

---

37 *try_me_else L1*

Local : [d,V3,V2,17,0,V3,c,5,_]
Escolha : [-1,1,25,8,17,0,9,V3,d,-1,-1,29,-1,17,4,-1,V8,V2
Trilha : [V8,V2]
E=0,G=-1,P=18, PC=22,B=17,EB=9,TR=1

---

38 *get_const a, A1* FALHA !

Local : [d,V3,V2,17,0,V3,c,5,_]
Escolha : [-1,1,25,8,17,0,9,V3,d,-1,-1,29,-1,17,4,-1,V8,V2
Trilha : [V8,V2]
E=0,G=-1,P=18, PC=25,B=17,EB=9,TR=1

---

39 *retry_me_else L2*

Local : [d,V3,V2,17,0,V3,c,5,_]
Escolha : [-1,1,29,8,17,0,9,V3,d,-1,-1,29,-1,17,4,-1,V8,V2
Trilha : [V8,V2]
E=0,G=-1,P=18, PC=26,B=17,EB=9,TR=1

---

40 *get_const c, A1* FALHA !

```
Local    :  [d,V3,V2,17,0,V3,c,5,_]
Escolha  :  [-1,1,29,8,17,0,9,V3,d,-1,-1,29,-1,17,4,-1,V8,V2]
Trilha   : [V8,V2]
E=0,G=-1,P=18,   PC=29,B=17,EB=9,TR=1
```

---

41 *trust_me_else fail*

```
Local    :  [d,V3,V2,17,0,V3,c,5,_]
Escolha  :  [V3,d,-1,-1,29,-1,17,4,-1,V8,V2]
Trilha   : [V8,V2]
E=0,G=-1,P=9,   PC=30,B=8,EB=9,TR=1
```

---

42 *get_const b, A1* FALHA !

```
Local    :  [V8,V3,V2,17,0,V3,V2,5,_]
Escolha  :  [V3,d,-1,-1,29,-1,17,4,-1,V8,V2]
Trilha   : []
E=0,G=-1,P=9,   PC=29,B=17,EB=9,TR=1
```

---

43 *trust_me_else fail*

```
Local    :  [V8,V3,V2,17,0,V3,V2,5,_]
Escolha  :  [V8,V2]
Trilha   : []
E=4,G=-1,P=-1,   PC=30,B=-1,EB=-1,TR=-1
```

---

44 *get_const b, A1*

```
Local    :  [V8,V3,V2,17,0,V3,b,5,_]
Escolha  :  [V8,V2]
Trilha   : []
E=4,G=-1,P=-1,   PC=31,B=-1,EB=-1,TR=-1
```

---

45 *get_const c, A2*

```
Local    :  [c,V3,V2,17,0,V3,b,5,_]
Escolha  :  [V8,V2]
Trilha   : []
E=4,G=-1,P=-1,   PC=32,B=-1,EB=-1,TR=-1
```

---

46 *exit*

Local : [c,V3,V2,17,0,V3,b,5,_]
Escolha : [V8,V2]
Trilha : []
E=4,G=-1,P=-1, PC=17,B=-1,EB=-1,TR=-1

---

47 *put_value V3, A1*

Local : [c,V3,V2,17,0,V3,b,5,_]
Escolha : [V8,c]
Trilha : []
E=4,G=-1,P=-1, PC=18,B=-1,EB=-1,TR=-1

---

48 *put_value V2, A2*

Local : [c,V3,V2,17,0,V3,b,5,_]
Escolha : [V3,c]
Trilha : []
E=4,G=-1,P=-1, PC=19,B=-1,EB=-1,TR=-1

---

49 *deallocate*

Local : [V3,b,5,_]

Desta vez o ambiente é efetivamente descartado já que a meta está sendo resolvida determinísticamente.

Escolha : [V3,c]
Trilha : []
E=0,G=-1,P=-1, PC=20,B=-1,EB=-1,TR=-1

---

50 *depart pai/2*

Local : [V3,b,5,_]
Escolha : [V3,c]
Trilha : []
E=0,G=-1,P=-1, PC=21,B=-1,EB=-1,TR=-1

---

51 *try_me_else L1*

Local : [V3,b,5,_]
Escolha : [-1,-1,25,-1,17,0,-1,V3,c]
Trilha : []
E=0,G=-1,P=9, PC=22,B=8,EB=4,TR=-1

---

52 *get_const a, A1* FALHA !

Local : [V3,b,5,_]
Escolha : [-1,-1,25,-1,17,0,-1,V3,c]
Trilha : []
E=0,G=-1,P=9, PC=25,B=8,EB=4,TR=-1

---

53 *retry_me_else L2*

Local : [V3,b,5,_]
Escolha : [-1,-1,29,-1,17,0,-1,V3,c]
Trilha : []
E=0,G=-1,P=9, PC=26,B=8,EB=4,TR=-1

---

54 *get_const c, A1*

Local : [V3,b,5,_]
Escolha : [-1,-1,29,-1,17,0,-1,V3,c]
Trilha : []
E=0,G=-1,P=9, PC=27,B=8,EB=4,TR=-1

---

55 *get_const d, A2*

Local : [d,b,5,_]
Escolha : [-1,-1,29,-1,17,0,-1,V3,c]
Trilha : [V3]
E=0,G=-1,P=9, PC=28,B=8,EB=4,TR=1

---

56 *exit*

Local : [d,b,5,_]
Escolha : [-1,-1,29,-1,17,0,-1,V3,c]
Trilha : [V3]
E=0,G=-1,P=9, PC=5,B=8,EB=4,TR=1

---

57 *stop*

Encontrou a segunda solução : *avo(b,d)*

---

58 Ainda existe um registro na pilha de escolha. Portanto é possível forçar novamente uma falha:

Local : [V3,b,5,_]

Escolha : [-1,-1,29,-1,17,0,-1,V3,c]

Trilha : []

E=0,G=-1,P=9, PC=29,B=8,EB=4,TR=-1

---

59 *trust_me_else fail*

Local : [V3,b,5,_]

Escolha : [V3,c]

Trilha : []

E=0,G=-1,P=-1, PC=30,B=-1,EB=-1,TR=-1

---

60 *get_const b, A1* FALHA !

Não há mais pontos de escolha (B=-1). Portanto, o emulador deve indicar que não há mais soluções possíveis e reinicializar-se para uma próxima consulta.

no.

?-

O laço principal do emulador pode ser implementado como segue:

```
void executa( pc )
int pc ;
{
  int cod_op;

  Get = Put = -1 ;
  ExCod = &Codigo[ pc ] ;
  while( 1 )
  {
    cod_op = ( int ) pega_byte() ;
    if( cod_op == STOP ) break ;
    switch( cod_op )
    {
      case TRY_ME_ELSE :

      case RETRY_ME_ELSE :

      case TRUST_ME_ELSE :

      case ALLOCATE :

      case DEALLOCATE :

      case CALL :

      case EXECUTE :

      case PROCEED :

      case PUT_VAR :

      case PUT_VAL :

      case PUT_ATOM :

      case GET_VAR :

      case GET_VAL :

      case GET_ATOM :

    }
  }
}
```

A seguir apresentaremos a implementação de cada uma das instruções, iniciando pelas mais simples :

```
case PUT_VAR :
      Vn = E + 1 + pega_indice_var() ;
      Ai = pega_indice_arg() ;
      Celula.tag = LOCAL ;
      Celula.valor = Vn ;                          /* 1 Ver nota */
      Choicel P + Ai ] = Stackl Vn ] = Celula ;
      /*  Refira-se à seção 2 do exemplo de avô.
          pega_indice_var() fornece o índice 1 de V1.
          Vn é a posição da variável na pilha local.
          Stack é a pilha local.
          Choice é a pilha de escolhas e argumentos.
          Célula.tag indica o tipo de dado na pilha:
               variável, constante, controle.
          1. Note que a variável livre aponta para ela  mesma
      */
      break ;

case GET_VAR :
     Vn = pega_indice_var() ;
     Ai = pega_indice_arg() ;
     Stackl E + 1 + Vn ] = Choicel G + Ai ] ;
     break ;

  case CALL :
     PC = endereco_do_inicio_da_primeira_clausula() ;
     NVars = numero_de_variaveis_preservadas() ;
     /* No código call g/2,5,  NVars = 5.
        Na seção 4 do exemplo, PC = 11 */
     NArgs = quantidade_de_argumentos() ;
     CP = endereco_de_retorno() ;
     G = P ;
     if( PC < 0 )
       falha() ;
     break ;

  case ALLOCATE :
     CE = E ;
     E = CE < EB ? EB : CE + NVars + 2 ;
     Stackl E ].valor = CE ;
     Stackl E + 1 ].valor = CP ;
     Stackl E ].tag = Stackl E + 1 ].tag = CONTROLE ;
     break ;

case PUT_VALUE :
      Vn = pega_byte() ;
      i = E + 1 + Vn ;
      Ai = pega_byte() ;
      Celula = *deref( &Stackl i ] ) ;
      Choicel P + Ai ] = Celula ;
      break ;
```

```
case GET_CONST :
     i = pega_inteiro() ;
     get_const( pega_byte(), ATOMO, i ) ;
     break ;

 case TRY_ME_ELSE :
     Ender1 = &Choice[ B + NArgs ] ;
     Ender1 -> tag = CONTROLE ;
     Ender1++ -> valor = EB ;
     Ender1 -> tag = CONTROLE ;
     Ender1++ -> valor = E ;
     Ender1 -> tag = CONTROLE ;
     Ender1++ -> valor = CP ;
     Ender1 -> tag = CONTROLE ;
     Ender1++ -> valor = B ;
     Ender1 -> tag = CONTROLE ;
     Ender1++ -> valor = pega_inteiro() ;
     Ender1 -> tag = CONTROLE ;
     Ender1++ -> valor = TR ;
     Ender1 -> tag = CONTROLE ;
     HB = Ender1++ -> valor = H ;

     B += NArgs + 7 ;
     P = B - 1 ;
     CE = E + NVars + 2 ;
     EB = CE > EB ? CE : EB ;
     break ;

 case RETRY_ME_ELSE :
     Choice[ B - 3 ].valor = pega_inteiro() ;
     G = Choice[ B - 4 ].valor - 1 ;
     P = B - 1 ;
     CE = E + NVars + 2 ;
     EB = CE > EB ? CE : EB ;
     break;

 case TRUST_ME_ELSE :
     HB = Choice[ B - 1 ].valor ;
     EB = Choice[ B - 7 ].valor ;
     B = Choice[ B - 4 ].valor ;
     ( void ) pega_inteiro() ;
     G = P = B - 1 ;
     break ;

 case DEALLOCATE :
     CE = E ;
     CP = Stack[ E + 1 ].valor ;
     E = Stack[ E ].valor ;
     NVars = Codigo[ CP - 1 ] ;
     break;

 case EXIT :
     PC = CP ;
     ExCod = &Codigo[ PC ] ;
     break;
```

```
      case DEPART :
         PC = endereco_do_inicio_da_primeira_clausula() ;
         NArgs = quantidade_de_argumentos() ;
         G = P ;
         if( PC < 0 )
           falha() ;



void falha()
{
  int ii, tr_linha;

  if( B == 0 )
  {
    PC = 0 ;
    ExCod = Codigo ;
    return ;
  }
  tr_linha = ( int ) Choicel[ B - 2 ].valor ;
  for( ii = tr_linha; ii < TR ; ii++ )
  {
    if( Traill[ ii ].tag == LOCAL )
      Ender1 = &Stackl[ ( int ) Traill[ ii ].valor ] ;
    else
      Ender1 = &Heapl[ ( int ) Traill[ ii ].valor ] ;
    *Ender1 = Traill[ ii ] ;
  }
  CE = E ;
  E = Choicel[ B - 6 ].valor ;
  H = Choicel[ B - 1 ].valor ;
  CP = Choicel[ B - 5 ].valor ;
  NVars = Codigol[ CP - 1 ] ; /* (CE == E ? NVars : CE - E - 2) ; */
  TR = tr_linha ;
  PC = ( int ) Choicel[ B - 3 ].valor ;
  ExCod = &Codigol[ PC ] ;
  return ;
}

void trail( tipo, posicao )
int tipo, posicao ;
{
    Traill[ TR ].tag = tipo ;
    Traill[ TR++ ].valor = posicao ;
}
```

Podemos agora descrever a implementação do corte a partir das estruturas de dados que foram descritas até agora. Antes, porém, convém lembrar como é o seu funcionamento. Se uma das cláusulas que compõem um procedimento não determinístico contém um corte, ao executá-lo, tal procedimento se tornará determinístico, isto é, as eventuais cláusulas que ainda não foram tentadas, não o serão mais. Além disso, quaisquer metas do corpo da cláusula que contém o corte que tenham sido resolvidas não determinísticamente também passarão a se comportar como se tivessem sido resolvidas determinísticamente. Em outras palavras, se depois do corte houver uma falha de unificação, o retrocesso será feito para o último procedimento resolvido não determinísticamente no momento da chamada do procedimento atual.

Em termos de nosso sistema de execução, o significado deste comportamento é que, ao executar-se um corte, todos os pontos de escolha que tiverem sido criados desde a chamada do procedimento que está em execução, serão descartados. A partir daí podemos deduzir a estratégia para a sua implementação: ao chamar um procedimento qualquer (via *call* ou *depart*), salvamos num registrador auxiliar, digamos *CutPt*, o valor atual de B para que ele possa ser restaurado durante a execução da instrução correspondente ao corte. Se uma determinada cláusula contém um corte, será gerada, durante compilação uma instrução específica de acordo com o tipo de cláusula: se tiver sido criado um ambiente para ela, *CutPt* será salvo numa variável auxiliar e o corte será traduzido por *cute Vn*. Por outro lado, se não for gerado um ambiente para a cláusula, o corte será simplesmente traduzido pela instrução *cutne* que restaura B a partir do valor de *CutPt*. É importante notar que o registrador EB também tem que ser ajustado de tal forma que os ambientes que estavam sendo mantidos para a eventual reativação de uma cláusula também sejam descartados. Este valor pode ser obtido do ponto de escolha que se tornar corrente depois da execução do corte. Esta estratégia é uma variação dos esquemas sugeridos em [Debray, 1986; Barklund, 1986].

Um outro aspecto importante do sistema de execução diz respeito à implementação dos predicados intrínsecos. Nesta implementação, tais predicados, assim como todo o sistema, foram desenvolvidos na linguagem C, observando as convenções adotadas quanto à passagem de parâmetros, continuação e retrocesso em caso de falha. As instruções *call* e *depart* foram ligeiramente alteradas para que ao procurar o procedimento, detectem se na realidade se trata de um predicado intrínseco. Neste caso, o sistema chama a função C correspondente e ela se encarregará de analisar os argumentos, gerar ocorrências para as variáveis da chamada e ativar o retrocesso conforme for necessário.

Finalmente uma palavra sobre o analisador léxico-sintático para o compilador da linguagem. Devido à simplicidade estrutural da linguagem, a sua compilação pode ser feita por um analisador descendente recursivo de dois passos: no primeiro a análise propriamente dita é realizada e o termo Prolog é traduzido para a sua representação interna. No segundo passo é feito um exame das variáveis contidas na cláusula para que sejam geradas instruções que permitam a poda de ambientes em tempo de execução. É conveniente notar que o primeiro passo do compilador nada mais é do que uma implementação do predicado intrínseco *read* que gera a ocorrência de seu argumento a partir de um termo lido de um dispositivo de entrada. Além disso, a sintaxe usada nesta implementação, chamada de sintaxe de Edinburgo ou DEC-10 [Pereira, 1980], permite o uso e criação de operadores para melhorar a legibilidade das cláusulas. Portanto, além da descida recursiva, o analisador sintático deverá conter um componente de análise de expressões por precedência de operadores [Aho, 1980].

Capítulo III. Lógica de Restrições

3.1 Introdução.

O ser humano, mesmo quando usa lógica em seu raciocínio, frequentemente se vê obrigado a impor restrições entre as variáveis que aparecem nos literais. Estas restrições têm origem em conhecimentos que se tem sobre relações existentes no domínio das variáveis presentes nos literais. Estas relações estão frequentemente associadas a procedimentos e, por isso, não são bem expressas na forma de sentenças lógicas. Por exemplo, alguém que estivesse trabalhando com economia, poderia ter entre suas sentenças lógicas várias equações e inequações. Em vez de tentar propor mais sentenças lógicas para resolvê-las ou procurar escrever um procedimento sob a forma de sentenças lógicas (e.g. um procedimento em Prolog), o economista poderia colocar as equações em uma pilha a medida em que fossem sendo encontradas no processo de prova aplicado às sentenças lógicas. Quando elas fossem suficientes para isto, seria aplicado um procedimento (e.g. método simplex) que as resolveria e atribuiria valores às variáveis. Tais valores poderiam orientar a prova. O procedimento também poderia descobrir que as equações e inequações são insatisfatíveis e, desta forma, disparar um retrocesso na prova.

Podemos também encarar restrições como predicados bem definidos sobre algum dominio. Assim, uma linguagem de programação por restrições é também uma linguagem lógica que privilegia um (ou vários) domínio(s) dando um tratamento mais homogêneo do ponto de vista lógico aos procedimentos usados em tais domínios, isto é, os predicados são agrupados em **interpretados** (restrições) e **não interpretados** (definidos pelo programador) [Nilsson, 1990]. Desta forma pode-se dizer que Prolog é uma linguagem de restrições sobre um domínio de árvores [Colmerauer, 1990], tendo como único procedimento embutido o de unificação de termos. Já linguagens como CLP(R) [Lassez, 1987] e Prolog III [Colmerauer, 1990] definem procedimentos sobre domínios como os números reais, árvores infinitas e os números inteiros.

A origem deste tipo de linguagens, como dissemos antes, veio da necessidade de se dar um tratamento aos predicados que melhor se expressam como procedimentos que fosse consistente com a programação em lógica, isto é, fazer com que tais procedimentos fossem capazes de manipular variáveis cujos valores ainda não fossem conhecidos como se faz nas primitivas do Prolog. Assim, a tarefa de um provador de teoremas para este tipo de linguagens não seria mais a de obter os valores de variáveis que tornam um conjunto de sentenças verdadeiro, mas sim possiveis conjuntos de valores que satisfazem tais sentenças. Quando se tratam domínios numéricos (naturais, inteiros ou reais) a idéia é análoga à dos diversos tipos de programação matemática (programação linear, programação inteira) que tratam, por exemplo, da satisfação de conjuntos de desigualdades.

O uso de restrições como uma representação de relações não é completamente novo. Abelson [1985] nos dá um breve histórico da evolução destas idéias que começa em 1963 num sistema de interface gráfica, passando por um sistema de propagação de restrições baseado na linguagem Smalltalk até um sistema comercial de análise de circuitos elétricos de 1983.

3.2 Exemplos de Programação por Restrições.

Fixando-nos no domínio dos números reais aproximados, vamos apresentar algumas aplicações da lógica de restrições encontradas na bibliografia [Lassez, 1987; Colmerauer, 1990; Cohen, 1990].

3.2.1 Malha resistiva.

Consideremos o seguinte programa CLP(R) [Lassez, 1987] :

```
lei_de_ohm(V,I,R) :- V = I * R.
lei_de_kirchoff(L) :- soma(L,0).
```

```
soma([],0).
soma([L|R],N) :- L + M = N,
      soma(R,M).

resistencia(10).
resistencia(14).
resistencia(27).
resistencia(60).
resistencia(100).

bateria(10).
bateria(20).
```

O objetivo é construir um circuito de duas resistencias (R1 e R2) conectadas em série com uma bateria (V) de tal forma que a voltagem em R2 fique entre 14,5 e 16,5 Volts. A seguinte meta pede os valores possiveis para os componentes do circuito:

```
?- 14.5 < V2, V2 < 16.5,
      resistencia(R1), resistencia(R2),
      bateria(V),
      lei_de_ohm(V1,I1,R1),
      lei_de_ohm(V2,I2,R2),
      lei_de_kirchoff([I1,-I2]),
      lei_de_kirchoff([-V,V1,V2]).
```

As duas primeiras desigualdades são restrições que, por conterem variáveis para as quais não foram geradas ocorrências, são empilhadas na pilha de restrições. Em seguida, são chamadas tres metas lógicas (definidas pelo programador e não consideradas como restrições) que "sugerem" valores possíveis para as variáveis *R1*, *R2* e *V*, os quais são passados para as metas seguintes. Estas, por sua vez estão definidas em termos de restrições que geram o seguinte sistema de equações:

```
V1/a - V2/b = 0
V1 + V2 = c
```

onde *a*, *b*, e *c* são as possíveis ocorrências para *R1*, *R2* e *V* respectivamente. Do conjunto resultante de quatro restrições, CLP(R) calcula os tres conjuntos de soluções:

```
V=20, R1=10, R2=27
V=20, R1=14, R2=60
V=20, R1=27, R2=100
```

Neste exemplo podemos ver claramente a diferença entre linguagens como CLP(R) e Prolog: as chamadas ao predicado *lei_de_ohm* são feitas com valor conhecido em apenas um de seus argumentos, o que impossibilitaria ao Prolog a avaliação da expressão aritmética contida no corpo da cláusula e ocorreria portanto um retrocesso; já em CLP(R) a restrição seria empilhada uma vez que não há evidencias suficientes de que ela não poderá ser satisfeita posteriormente.

3.2.2 Cálculo de Prestações.

O seguinte exemplo, apresentado em [Colmerauer, 1990], mostra os recursos de Prolog III para manipulação de números reais.

O problema é o de calcular uma série de pagamentos a serem feitos para amortizar um certo capital emprestado a uma taxa de juros de dez por cento por período.

```
pagamentos([], 0).
pagamentos([I|X],C) :-
        pagamentos(X,(110/100)*C-I).
```

Uma pergunta interessante que pode ser apresentada a este programa é qual a taxa *i* necessária para pagar $1000 através da sequência [*i*, 2*i, 3*i] de pagamentos:

```
?- pagamentos([I,2*I,3*I],1000).
```

para obter a resposta

I = 207 + 413/641.

Uma análise passo a passo de como é feita a execução do programa mostra como esse resultado é obtido. Inicialmente temos:

*pagamentos([1,2*I,3*I],1000)*

e aplicando a regra

*pagamentos([I|X],C) :- pagamentos(X,(110/100)*C-I)*

obtemos

*pagamentos([2*I,3*I],1100-I).*

Aplicando a mesma regra mais duas vezes, obtemos:

*pagamentos([3*I],1210-(31/10)*I)*

e

*pagamentos([],1331-(641/100)*I).*

Aplicando agora a regra

pagamentos([],0)

obtemos finalmente

*1331-(641/100)*I = 0*

que ao ser simplificada dá

I = 207 +413/641.

Neste exemplo fica claro que em linguagens como Prolog III, a unificação deixa de ser uma mera atribuição sintática de estruturas para se tornar um poderoso mecanismo que é capaz de avaliar tais estruturas de acordo com as operações dependentes de domínio nelas presentes.

3.2.3 Analisador Livre de Contexto.

Neste exemplo, também extraído de [Colmerauer, 1990], é mostrada uma forma natural de associar as regras de uma gramática livre de contexto com as regras de Prolog III. Considere a seguinte gramática:

$$\{S \rightarrow AX,\ A \rightarrow \Lambda,\ A \rightarrow aA,\ X \rightarrow \Lambda,\ X \rightarrow aXb\}$$

que define a linguagem que consiste das sequencias de simbolos da forma $a^m b^n$ para $m \geq n$. O seguinte programa corresponde a esta gramática:

```
formaS(U) :-
        formaA(V),
        formaX(W),
        {U = V • W}.

formaA(U) :-
        {U = []}.
formaA(U) :-
        formaA(V),
        {U = "a" • V}.

formaX(U) :-
        {U = []}.
formaX(U) :-
        formaX(V),
        {U = "a" • V • "b"}.
```

Em Prolog III a diferença entre predicados lógicos (definidos pelo programador) e restrições é enfatizada sintáticamente isolando todas as restrições no final da cláusula delimitadas por chaves. Na verdade tal diferenciação não é estritamente necessária como pode observar-se no exemplo de CLP(R) e poderia dizer-se até que esta separação explícita representa uma nova forma de obrigar o programador a introduzir controle nos programas.

3.3 Propagação local.

Para implementar sistemas de satisfação de restrições é necessário lançar mão de algoritmos, às vezes muito complexos, que nos permitam obter o efeito desejado de lidar com variáveis lógicas que participam de restrições mesmo antes de terem seus valores definidos. Certamente, o grau de complexidade dos algoritmos envolvidos irá depender dos domínios e das operações representadas na linguagem de satisfação de restrições.

Um exemplo simples de algoritmos desta natureza é o de propagação local, também conhecido como propagação de restrições [Abelson, 1985]. A idéia por trás deste algoritmo é que, enquanto que os programas são tradicionalmente construídos em termos de computações em uma só direção, frequentemente modelamos sistemas em termos de relações entre grandezas. Por exemplo [Abelson, 1985], o modelo matemático de uma estrutura mecânica poderia incluir a informação de que a deflexão d de uma barra metálica é proporcional à força F sobre a barra, seu comprimento C, sua área transversal A e o coeficiente de elasticidade E, através da equação

$$d \cdot A \cdot E = F \cdot C.$$

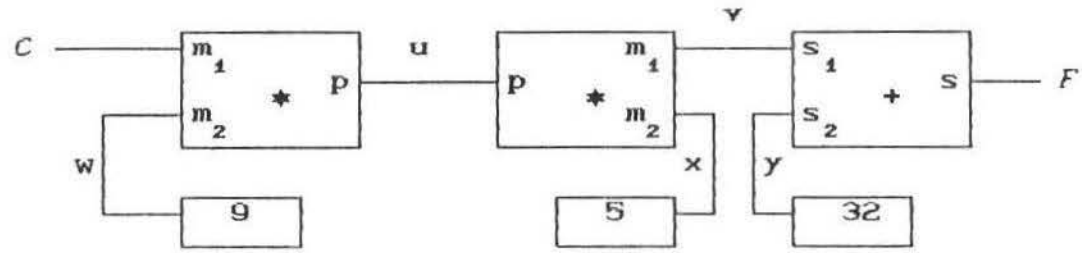
Tal equação não é unidirecional. Dados quaisquer quatro dos valores envolvidos, podemos usá-la para calcular o quinto.

A propagação local proporciona um meio de combinar as restrições primitivas da linguagem para expressar relações mais complexas. Tais combinações são construidas através de **redes de restrições**, nas quais as restrições são unidas por **conectores**.

Outro exemplo clássico é a relação entre temperaturas expressas em graus Centrigrados e graus Fahrenheit

$$9C = 5(F - 32)$$

Tal restrição pode ser vista como uma rede que combina as primitivas **soma**, **multiplicação**, e **constante** como é mostrado na seguinte figura:



Numa rede deste tipo, cada vez que um conector (C, F, u, v, w, x ou y) recebe um valor (do usuário ou da restrição à qual ele está ligado), ele a transmite a todas as outras restrições que estiverem conectadas a ele. Estas, por sua vez, verificam se já contam com informação suficiente para calcular o valor de algum outro conector e, se este for o caso, o transmitem às demais restrições. No exemplo, ao atribuir ao conector C um valor de, digamos, 25, o multiplicador da esquerda será ativado e atribuirá o valor de 225 (= 9 * 25) ao conector u. Por sua vez, o conector u ativará o segundo multiplicador que atribuirá o valor de 45 a v, o qual ativará o somador que dará a F o valor de 77.

3.4 Execução e compilação de programas com restrições.

Nesta seção apresentaremos um interpretador de uma linguagem lógica com restrições semelhante ao mostrado no capítulo II. Como dissémos anteriormente, restrições são na verdade predicados privilegiados de algum domínio específico. O compilador de uma linguagem de restrições deverá então distinguir esses predicados daqueles definidos pelo programador e gerar um código especial para eles. Durante execução, tal instrução empilhará a nova restrição. Além disso, periódicamente o sistema de execução poderá tentar simplificar o sistema de restrições gerado usando os valores encontrados até o momento para as variáveis. Esta simplificação pode ser vista como uma coleta de lixo na pilha de restrições e pode ser realizada em tres ocasiões:

1. quando a pilha de restrições estiver práticamente cheia e houver a necessidade de empilhar uma nova restrição,
2. durante a execução de uma instrução *exit* correspondente à

conclusão de uma cláusula unitária, e

3. quando o programador assim o solicitar através de uma primitiva especial (e.g. *gc*).

3.4.1 Um interpretador básico de linguagens de restrições.

De maneira análoga à usada no Capítulo II, podemos descrever um pequeno interpretador de linguagens de restrições através do seguinte programa escrito em Prolog:

```
execute(true,C,C) :- !.
execute(Meta,C0,C) :-
        separe(Meta, Esquerda, Direita),
        chame(Esquerda, C0, C1),
        execute(Direita, C1, C).

separe((Esq, Dir), Esq, Dir) :- !.
separe(Meta, Meta, true).

chame(Meta, C0, C) :-
        clausulas(Meta, Clausulas),
        tente(Meta, Clausulas, C0, C).

clausulas(Meta, Clausulas) :-
        findall((Meta :- Corpo, Restr),
                clause(Meta, Corpo, Restr), Clausulas).

tente(Meta, [(Cabeca :- Corpo, Restr)|Clausulas], C0, C) :-
        unifique(Meta, Cabeca),
        simplifique(C0, Restr, C1),
        execute(Corpo, C1, C) ;
        tente(Meta, Clausulas, C0, C).

unifique(X,X).
```

As novas variáveis, *C0* e *C*, introduzidas nos procedimentos *execute*, *chame* e *tente* representam, respectivamente o estado

inicial e final da pilha de restrições. Assim, o procedimento *execute*, resolve a primeira meta através do procedimento *chame* o qual produz um estado intermediário da pilha de restrições que, por sua vez é utilizado como ponto de partida para solução das metas restantes.

O procedimento *chame*, como anteriormente, obtém uma lista de cláusulas candidatas a resolver a meta apresentada, sendo que estas cláusulas agora são apresentadas separando as metas lógicas (definidas pelo usuário) das restrições.

Finalmente, no procedimento *tente*, após a unificação convencional dos argumentos da chamada com os da cabeça da cláusula, é chamado um novo procedimento *simplifique*, que efetivamente se encarregará de atualizar pilha de restrições em face dos novos valores das variáveis, eventualmente obtidos da unificação e das restrições presentes no corpo da cláusula que está sendo usada.

3.4.2 Um conjunto de instruções para a execução de restrições.

Seguindo os mesmos passos que foram utilizados no Capítulo II, poderemos obter uma versão mais refinada do nosso interpretador de restrições no qual ficam explícitas as operações necessárias para o tratamento da recursividade de cauda e do não determinismo. No entanto, podemos notar que a estrutura do novo interpretador é a mesma que a anterior, sendo que foram acrescentadas novas variáveis para representar a pilha de restrições e novas cláusulas para o empilhamento e simplificação de tais restrições.

```
execute([Arg|MArgs], [var,N|PC], Vars, Cont, Aux, GO, Col ) :- !,
        arg(N, Vars, Arg),
        execute(MArgs, PC, Vars, Cont, Aux, GO, Col).
execute([C|MArgs], [const,C|PC], Vars, Cont, Aux, GO, Col) :- !,
        execute(MArgs, PC, Vars, Cont, Aux, GO, Col).
```

```
execute([], [enter|PC], Vars, Cont, [], GO, Col) :-
    execute(Args, PC, Vars, Cont, Args, GO, Col).
execute(Args, [constr,Op|PC], Vars, Cont, [], GO, Col) :- !,
    execute(Args1, PC, Vars, Cont, Args1, [c(Op,Args)|GO], Col).
execute(Args, [gc|PC], Vars, Cont, Aux, GO, Col) :-
    simplifique(GO, Gs),
    execute(Args, PC, Vars, Cont, Aux, Gs, Col).
execute([], [exit], _, [], [], G, Gs) :-
    simplifique(G, Gs).
```

Supondo uma linguagem que trata de restrições sobre números inteiros e reconhece especificamente os predicados > e =, e as operações + e *, o programa de cálculo de fatorial apresentado anteriormente seria traduzido como segue:

```
fat:    const   0
        const   1
        exit

fat:    var     1
        var     2
        enter
        var     1
        const   0
        constr  >
        var     1
        var     3
        const   1
        constr  +
        var     3
        var     4
        call    fat
        var     2
        var     1
        var     4
        constr  *
        exit
```

sendo que *constr* + é usada tanto para a soma como para a subtração e empilhará uma restrição da forma *c(+,[Op1, Op2, Op3])* que deve ser interpretada como *Op1 = Op2 + Op3*. Assim, restrição *A = B - C* é primeiro transformada na restrição equivalente *B = A + C* para depois ser traduzida como mostramos.

3.5 Conclusões

No capítulo II falamos sobre compilação de Prolog e fizemos uma proposta para melhorar a eficiência do processo de geração de código e a velocidade de execução, a saber, utilizamos uma mesma pilha para os argumentos e para armazenamento de pontos de escolha. Com isto, evitamos que os argumentos tivessem de ser copiados nas estruturas que representam os pontos de escolha. Apesar deste resultado interessante que obtivemos em nossa pesquisa, é bom levar em conta que compiladores Prolog já existem há pelo menos 10 anos. Consideramos, portanto, necessário incluir neste trabalho um estudo sobre compilação de Lógica de Restrições. Como este assunto é recente e pouquíssimos resultados foram publicados até agora, esperamos ter feito alguma contribuição mais próxima da última palavra (estado da arte).

Neste capítulo vimos que a diferença básica de um compilador para linguagens de restrições com relação ao Prolog é que o primeiro deverá diferenciar predicados interpretados dos não interpretados e gerar instruções específicas em cada caso, de tal forma que cada nova restrição que for encontrada durante a execução do programa possa ser utilizada como entrada para um algoritmo de simplificação de um sistema de restrições que é armazenado numa pilha separada. Por sua vez, os predicados não interpretados continuarão sendo tratados como em Prolog, isto é, as ocorrências das variáveis contidas nestes predicados serão obtidas a partir do algoritmo de unificação.

Na verdade esta seria uma primeira abordagem à questão da compilação de linguagens de restrições já que o único trabalho que o compilador tem é o de reconhecer as restrições e gerar um código

diferente. Na medida em que os domínios que são tratados numa determinada linguagem são definidos, as operações poderiam ser eventualmente otimizadas de tal forma que o sistema de execução tivesse um desempenho melhor. No entanto, é claro que não existe uma linguagem de restrições universal que possa oferecer desempenho ótimo em todos os domínios. Talvez, uma possivel abordagem que permita uma melhor compreensão do problema da compilação deste tipo de linguagens seja a de construir compiladores de compiladores de linguagens de restrições.

BIBLIOGRAFIA.

[Abelson, 1985] Abelson, H., Sussman, G.J. **Structure and Interpretation of Computer Programs.** MIT Press. Cambridge, EUA. 1985.

[Aho, 1986] Aho, A.V., Sethi, R., Ullman, J.D. **Compilers, Principles, Techniques and Tools.** Addison Wesley Publishing Company, EUA, 1986.

[Barklund, 1986] Barklund, J., Millroth, H. **Garbage Cut for Garbage Collection of Iterative Prolog Programs**, in *Proceedings 1986 Symposium on Logic Programming*, IEEE Computer Society Press, pp 276-290, 1986.

[Bowen D, 1983] Bowen, D.L., Byrd, L.M., Clocksin, W.F. **A Portable Prolog Compiler.** Logic Programming Workshop '83. Universidade Nova de Lisboa, Jun. 1983, pp 74-83.

[Bowen K, 1985] Bowen, K.A., Buettner, K.A., Ciceklis, I., Turk, A. **The Design and Implementation of a High-Speed Incremental Portable Prolog Compiler.** Syracuse University, Technical Report CIS-85-6. Nov. 1985.

[Boyer, 1972] Boyer, R.S., Moore, J.S. **The Sharing of Structure in Theorem-Proving Programs**, in *Machine Intelligence*. V 7, pp 101-116, 1972.

[Bruynooghe, 1982] Bruynooghe, M. **The Memory Management of Prolog Implementations**, in *Logic Programming*, K.L. Clark, S.A. Taernlund (eds) Academic Press, New York, EUA. 1982.

[Chang, 1973] Chang, C.-L., Lee, R.C.-T. **Symbolic Logic and Mechanical Theorem Proving.** Academic Press, New York, EUA, 1973.

[Clocksin, 1981] Clocksin, W.F., Mellish, C.S. **Programming in Prolog.** Springer-Verlag, 1981.

[Cohen, 1990] Cohen, J. **Constraint Logic Programming Languages**, in *Communications of the Association for Computing Machinery*, V 33, N 7, pp 52-68, Jul. 1990.

[Cohn, 1965] Cohn, P.M. **Universal Algebra.** Harper & Row, Publishers. EUA, 1965.

[Colmerauer, 1990] Colmerauer, A. **An Introduction to Prolog III**, in *Communications of the Association for Computing Machinery*, V 33, N 7, pp 69-91, Jul. 1990.

[Davis, 1960] Davis, M., Putnam, H. **A computing procedure for quantification theory.** JACM V 7, N 3, pp 201-215, 1960.

[de Barros, 1991] de Barros, M.P. **Introdução Automática de Controle em Programação Lógica.** Tese de Mestrado, Departamento de Engenharia Elétrica, Universidade Federal de Uberlândia, Março, 1991.

[Debray, 1986] Debray, S.K. **Register Allocation in a Prolog Machine**, in *Proceedings 1986 Symposium on Logic Programming*, IEEE Computer Society Press pp 267-275. 1986

[Epstein, 1989] Epstein, R.L., Carnielli, W.A. **Computability: Computable Functions, Logic and the Foundations of Mathematics.** Wadsworth & Brooks/Cole. EUA, 1989.

[Gries, 1981] Gries, D. **The Science of Programming.** Springer-Verlag. New York, 1981.

[Horn, 1951] Horn, A. **On Sentences which are true of direct unions of algebras**, in *Journal of Symbolic Logic*, V 16, N 1, pp 14-21. Mar. 1951

[Kowalski, 1971] Kowalski, R., Kuehner, D. **Linear Resolution with Selection Function**, in *Artificial Intelligence*, V 2, N 3/4, pp 227-260.

[Kowalski, 1974] Kowalski, R. **Predicate Logic as a Programming Language**, in *Proc. IFIP '74 World Congress.* North-Holland Publishing Company. Amsterdam, pp 689-754, 1974.

[Kuehner, 1972] Kuehner, D. **Some Special Purpose Resolution Systems**, in *Machine Intelligence*, V 7, pp 117-128.

[Lassez, 1987] Lassez, C. **Constraint Logic Programming**, in *BYTE*, V 12, N 9, Ago. 1987.

[Lucena, 1990] Lucena, H. Comunicação particular com o Prof. Antonio E. Costa Pereira. 1990.

[Mellish, 1982] Mellish, C.S. **An Alternative to Structure Sharing in the Implementation of a Prolog Interpreter**, in *Logic Programming*, K.L. Clark, S.A. Taernlund (eds) Academic Press, New York, EUA. 1982.

[Nilsson, 1990] Nilsson, U., Małuszyński, J. **Logic, Programming and Prolog.** John Wiley & Sons, Inglaterra. 1990.

[Robinson, 1963] Robinson, J.A. **Theorem-Proving on the Computer**, in *Journal of the Association for Computing Machinery*. V 10, N 12, pp 163-174, 1963.

[Robinson, 1965] Robinson, J.A. **A Machine Oriented Logic Based on the Resolution Principle**, in *Journal of the Association for Computing Machinery*. V 12, N 1, pp 23-41, 1965.

[van Emden, 1984] van Emden, M.H. **An Interpreting Algorithm for Prolog Programs**, in *Implementations of Prolog*, Campbell, J.A. (ed.). John Wiley, pp 93-110. 1984.

[Warren, 1977] Warren, D.H.D. **Applied Logic: Its use and implementation as a programming tool.** Tese de Doutoramento. University of Edimburgh, 1977.

[Warren DHD, 1983] Warren, D.H.D. **An Abstract Prolog Instruction Set.** SRI International, Technical Note 309. Out. 1983.

[Warren DS, 1983] Warren, D.S. **The Runtime Environment for a Prolog Compiler.** State University of New York at Stony Brook, Technical Report #83/52. Ago. 1983.

[Wos, 1984] Wos, L., Overbeek, R., Lusk, E., Boyle, J. **Automated Reasoning. Introduction and Applications.** Prentice-Hall, 1984.

[Yamaki, 1990] Yamaki, C.K., **Combinando Avaliação Parcial e Introdução Automática de Controle para Aumentar a Eficiência de Sistemas Especialistas.** Tese de Mestrado. Instituto de Matemática, Estatística e Ciência da Computação, Universidade Estadual de Campinas. Março, 1990.