048

PROGRAMAÇÃO EM CLÁUSULAS GENÉRICAS UTILIZANDO O MÉTODO DE ELIMINAÇÃO DE MODELOS

Ramiro Guerreiro[1], Andrea Silva[1,2] e Marco A. Casanova[1]

[1]Centro Científico Rio - IBM Brasil
Caixa Postal 4624
22.071, Rio de Janeiro, RJ

[2]Departamento de Informática - PUC/RJ
R. Marquês de S.Vicente, 209
22.453, Rio de Janeiro, RJ

SUMÁRIO

Uma linguagem para programação em cláusulas genéricas cuja semântica operacional baseia-se em uma variação do método de eliminação de modelos é apresentada. Aspectos sintáticos e semânticos são abordados, acompanhados de uma análise comparativa com a linguagem Prolog.

1. INTRODUÇÃO

Este trabalho apresenta uma linguagem para programação em lógica, chamada MEL ("Model Elimination Language"). Programas nesta linguagem são conjuntos finitos de cláusulas com um número arbitrário de literais positivos e negativos, sendo que a negação é interpretada no sentido clássico. A linguagem possui ainda o acervo usual de comandos extra-lógicos, incluindo um versão do "cut" e a negação por falha finita. A linguagem generaliza Prolog na medida em que qualquer programa em cláusulas definidas (ou seja, cláusulas com apenas um literal positivo) possui a mesma semântica em MEL e em Prolog. Da mesma forma, a versão do "cut" em MEL também é uma extensão do "cut" em Prolog.

A semântica operacional da linguagem MEL baseia-se em um método de refutação, chamado eliminação de modelos fraca (EMF) (Loveland [1968, 1969, 1978]). Este método é uma alternativa bastante interessante para a construção de sistemas para Programação em Lógica pois aceita cláusulas genéricas, é linear de entrada, não utiliza fatoração e, apesar destas características, mantém a completude. O espaço de busca pode ainda ser reduzido com a incorporação de filtros e restrições na aplicação das regras de derivação.

Além disto, Resolução-SLD, o método de refutação em que Prolog se baseia (Lloyd [1984]), pode ser visto como uma variação de eliminação de modelos restrita a cláusulas definidas. Procedimentos de refutação baseados em eliminação de modelos fraca podem também ser implementados de forma muito semelhante aos interpretadores Prolog (veja Casanova e Walter [1986]).

O protótipo de um interpretador para MEL, codificado em Prolog, encontra-se operacional. A implementação partiu da idéia usual de construir

interpretadores Prolog em Prolog, adaptando-a para as características próprias de eliminação de modelos. Em seguida, o código Prolog sofreu sucessivas otimizações até atingir um comportamento adequado em termos de tempo e memória. Desta forma, aproveitou-se ainda todos os extra-lógicos de Prolog para gerência da área de trabalho, entrada e saída, e outros. O código final possui cerca de 550 comandos Prolog.

Este trabalho está dividido da seguinte forma. A seção 2 apresenta a sintaxe da linguagem MEL básica, enquanto a seção 3 define a semântica operacional da linguagem. Finalmente, a seção 4 descreve as facilidades extra-lógicas, estabelecendo um paralelo com Prolog.

2. SINTAXE DA LINGUAGEM MEL BÁSICA

O *alfabeto básico MEL* compreende os seguintes símbolos:

pontuação:	(,), . , ', "
conectivos:	¬, ∧, |
letras:	a, b, c, ...
digitos:	0, 1, 2, ...
caracteres especiais:	*, _, +, -, /, ...

As definições de *átomo*, *constante*, *variável*, *termo* e *fórmula atômica* são as mesmas da linguagem Prolog básica e podem ser encontradas em Casanova et alii [1987,Cap.10]. Em particular, um átomo pode representar simultaneamente o papel de um ou mais símbolos funcionais ou predicativos de aridades diferentes. Este uso múltiplo do mesmo átomo não causa ambigüidades pois o contexto de um programa indicará sempre o papel que representa. Semelhantemente, uma mesma expressão pode ser um termo ou uma fórmula atômica, dependendo do contexto.

Seguem-se exemplos destes conceitos:

Átomos:	b, maria, caFE, "a b","Joao", "<-"
Constantes:	caFE, "a b", 15.5, 'a b', '←'
Variáveis:	X, Y, Mae, Z523x, RESPOSTA, A11, *
Termos MEL:	x, a, "a b", fortran, 'Joao Silva', 15.5, livro(Autor,Editora,Ano)
Fórmulas Atômicas MEL:	x(a,"a b",fortran,'Joao Silva',15.5,X) livro(Autor,Editora,Ano)

Cláusulas MEL são listas de fórmulas atômicas, negadas ou não, separadas pelo símbolo "|".

Exemplos de cláusulas MEL são:

odeia(cesar,marcus)	("César odeia Marcus")
¬pompeu(X) | romano(X)	("X não é de Pompéia ou é romano")

Um *programa MEL* é uma sequência de cláusulas MEL, sendo cada uma finalizada por um ponto (.). Permitiremos ainda a inclusão de comentários em um programa MEL entre os delimitadores "/*" e "*/".

Exemplo 1: Exemplo de um programa MEL.

```
          /*              Programa ROMANO            */

1. homem(marcus).
2. pompeu(marcus).
3. imperador(cesar).
4. tentamatar(marcus,cesar).
5. odeia(marcus,otavio).
6. ¬pompeu(X) | romano(X).
7. ¬romano(Y) | leal(Y,cesar) | odeia(Y, cesar).
8. leal(Z, f(Z)).
9. ¬homem(X) | ¬tentamatar(X,Y) | ¬imperador(Y) | ¬leal(X,Y)
```

Observe que do Cálculo Proposicional temos as seguintes equivalências:

$$A \rightarrow B \equiv \neg A \mid B \quad e \quad A \wedge B \equiv \neg(\neg A \mid \neg B)$$

Assim sendo, as interpretações pretendidas para as cláusulas c_7 e c_9, por exemplo, são "Se Y é romano então ou Y é leal a César ou Y odeia César" e "Se X é um homem e X tenta matar Y então, se Y é um imperador então X não é leal a Y".

O conjunto das *consultas admissíveis* MEL é definido como o menor conjunto que satisfaz às seguintes condições:

- se "A" é uma cláusula MEL então "A" é uma consulta admissível MEL;
- se "A" é uma consulta admissível MEL, então "(A)" é uma consulta admissível MEL.
- se "A" e "B" são consultas admissíveis MEL, então "A ∧ B" é uma consulta admissível MEL.
- se "A" e "B" são consultas admissíveis MEL, então "A | B" é uma consulta admissível MEL.

Uma *consulta MEL* é uma consulta admissível MEL finalizada por um ponto (.).

A prioridade entre os conectivos segue a ordem, do maior para o menor peso: ¬, ∧, |. O usuário pode fazer uso da parentização para alterar esta ordem, porém a negação só é admitida imediatamente antes de uma fórmula atômica. Assim, por exemplo, "¬(p(a) | p(b))" não é uma consulta MEL.

Se R é uma cláusula MEL, denote por $\overline{R}$ a conjunção dos literais complementares aos literais em R. Seja Q uma consulta MEL a um programa

MEL **P** e seja **C** a representação clausal de ¬Q. Uma *resposta* de Q a **P** é uma disjunção $\overline{R}$ da forma "$\overline{R}_1$ | ... | $\overline{R}_n$" onde $R_1,...,R_n$ são instâncias de cláusulas em **C**; se n=1, $\overline{R}$ é uma resposta *simples*, caso contrário, $\overline{R}$ é uma resposta *genérica*. Finalmente, $\overline{R}$ é uma resposta *correta* se e somente se **P** implica logicamente $\overline{R}$.

Exemplo 2: Consultas e Respostas.

Seja a seguinte consulta MEL ao programa Romano:

```
odeia(marcus,X)
```

Então teremos duas respostas corretas:

```
¬odeia(marcus,otavio)   e   ¬odeia(marcus,cesar)
```

3. SEMÂNTICA OPERACIONAL DA LINGUAGEM MEL BÁSICA

Esta seção introduz uma semântica operacional para a linguagem MEL através da definição da máquina abstrata MEL.

Inicialmente introduziremos informalmente o método de eliminação de modelos fraca. Este método trabalha com seqüências de literais, resolvidos ou não, chamados de *cadeias*. Literais resolvidos são denotados entre colchetes, "[" e "]". Uma cadeia é *elementar* se não contém literais resolvidos. Refutações por eliminação de modelos fraca sempre partem de um conjunto de cadeias elementares, são lineares de entrada e dispensam fatoração.

O método trabalha com duas regras de inferência, extensão plena e redução plena, definidas da seguinte forma.

Sejam A'' e A' cadeias e β uma renomeação para A'' em presença de A'. Seja L' o elemento mais à esquerda de A' e suponha que L' seja um literal. Uma cadeia A é uma *extensão* de A' *por* A'' se e somente se existe um literal L'' de A'' e uma substituição θ tais que L' e L''β são canceláveis por θ e A = B''B', onde B'' é a cadeia A''$\beta\theta$ com o literal L''$\beta\theta$ removido e B' é a cadeia A'θ com o literal L'θ transformado em um R-literal.

Seja A' uma cadeia. Seja L' o elemento mais à esquerda de A' e suponha que L' seja um literal. Uma cadeia A é uma *redução* de A' se e somente se existe um R-literal M' de A' e uma substituição θ tais que L' e M' são canceláveis por θ e A é A'θ com o literal L'θ removido.

Uma cadeia A é a *contração* de uma cadeia A' se e somente se A é obtida removendo-se repetidamente o elemento mais à esquerda de A' até que este seja um literal ou que A' se transforme na cadeia vazia.

Uma cadeia A é uma *extensão plena* de A' por A'' se somente se for a contração da extensão de A' por A''. Uma cadeia A é uma *redução plena* de uma cadeia A' se somente se for a contração da redução de A'.

A máquina MEL opera então da seguinte forma.

Dado um programa e uma consulta MEL, a máquina mapeia inicialmente as cláusulas MEL do programa diretamente em uma lista de cadeias elementares. Em seguida, a máquina acrescenta ao final desta lista o conjunto de cadeias oriundo da representação clausal da negação da consulta MEL (em uma ordem arbitrária). A máquina constrói então, em pré-ordem, o conjunto das árvores de refutação (por eliminação de modelos fraca) para estas cadeias cujas raízes são rotuladas com as cadeias oriundas da negação da consulta.

A máquina páara ou quando obtém um ramo de sucesso, retornando a resposta correta da consulta que corresponde ao ramo, ou quando todas as árvores foram construídas, retornando então falha. Porém, a máquina pode ainda não produzir nenhuma saída, caso comece a percorrer um ramo infinito. Com algumas alterações, a máquina pode também enumerar ramos de sucessos, produzindo respostas alternativas.

A construção de cada árvore obedece às seguintes regras:

- a regra de redução plena é exaustivamente aplicada antes da regra de extensão plena;
- nas aplicações de redução plena, os R-literais da cadeia em questão são selecionados da esquerda para a direita;
- nas aplicações de extensão plena:
  - as cadeias de entrada são selecionadas na ordem dada;
  - os literais de cada cadeia de entrada são selecionados da esquerda para a direita.

É possível provar que, de fato, as substituições efetuadas sobre as variáveis de cadeias originárias da negação da consulta em um ramo de sucesso induzem uma resposta correta (ver Casanova et alii [1988]). Porém, é necessário utilizar literais de resposta (Luckham e Nilsson [1971]) para capturar tais substituições pois a representação clausal da negação da consulta pode dar origem à várias cadeias que, por sua vez, podem ser utilizadas diversas vezes no ramo de sucesso.

Esta técnica exige substituir cada cadeia A originária do programa pelo par (A,$\emptyset$) e cada cadeia C originária da negação da consulta pelo par (C,{R($\overline{x}$)}), onde $\overline{x}$ é uma lista das variáveis que ocorrem em C e R é um novo símbolo predicativo de aridade igual ao comprimento de $\overline{x}$. O literal R($\overline{x}$) passa então a ser o *literal de resposta* associado a C. Pares da forma (C,**R**), onde C é uma cadeia e **R** é um conjunto de literais, são chamados de *cadeias ativadas*. As regras de inferência são então modificadas para trabalhar com cadeias ativadas, aplicando todas as substituições também aos literais de resposta (ver Casanova et alii [1988]). O próximo exemplo ilustra estes conceitos.

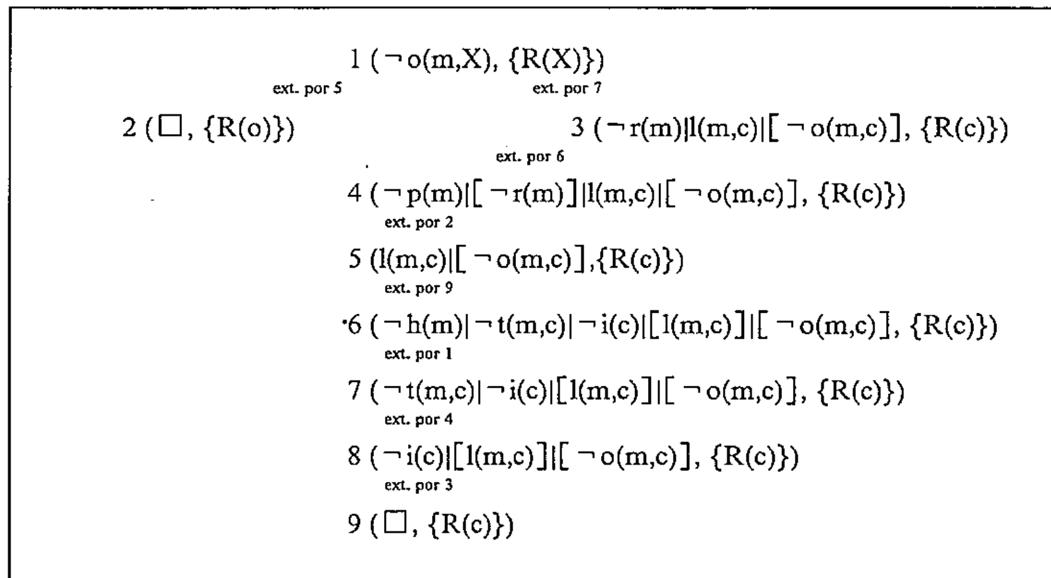
Seja C a seguinte consulta ao programa Romano:

```
odeia(marcus,X)
```

A representação clausal da negação de C gera apenas uma cadeia ativada, onde `R(X)` é o literal de resposta associado:

`(¬odeia(marcus,X), {R(X)})`

A máquina MEL gerará então apenas uma árvore de refutação (os átomos são simbolizados por suas letras iniciais e os nós numerados na ordem em que a máquina os gera). Note que a árvore possui dois ramos de sucesso. Portanto, a máquina retornará uma das respostas ao detectar o primeiro, no caso, "`odeia(marcus,otavio)`". Se requerido, a máquina retornará em seguida a resposta correta alternativa, que é "`odeia(marcus,cesar)`".



4. A LINGUAGEM MEL ESTENDIDA

As facilidades extra-lógicas estendem a linguagem MEL básica permitindo estilos de programação mais próximos aos tradicionais. Discutimos aqui a sintaxe e semântica dos comandos que implementam estas facilidades e estabelecemos um paralelo entre estes e os comandos Prolog correspondentes.

Vários comandos existentes em dialetos Prolog podem ser diretamente herdados por MEL, como comandos de aritmética e comparação, comandos de comunicação, comandos de processamento de cláusulas, comandos de depuração, entre outros. Apresentamos em detalhe aqui apenas os considerados mais relevantes.

4.1 Comando "cut"

O extra-lógico *"cut"*, denotado por "/", é o principal comando de controle do procedimento de retrocesso. Ele se comporta como um literal que é sempre satisfeito em uma cláusula MEL mas, em retrocesso, causa poda no nó imediatamente superior da árvore de refutação. O ramo corrente falha assim como todos os outros ramos seguintes que passam por este nó.

Exemplo 3:

Seja o programa Prolog abaixo:

```
1. a <- / & b.
2. a.
```

O programa MEL correspondente é:

```
1. a | / | ¬b.
2. a.
```

Seja a consulta "a" , que gera a cadeia:

3. ¬ a.

Como o comando *"cut"* em (1) impede um retrocesso sobre "a", a cláusula em (2) não é avaliada e a máquina MEL retorna falha.

O uso do *"cut"* em MEL exige mais cuidados do que em Prolog pois, para construir os descendentes de um nó *N*, a máquina MEL tenta inicialmente todas as possíveis reduções da cadeia P que rotula *N*, selecionando os literais resolvidos de P da esquerda para a direita, e em seguida todas as possíveis extensões com cada cadeia de entrada C, selecionando os literais de C da esquerda para a direita. Portanto, a poda realizada pelo *"cut* é muito mais complexa na máquina MEL.

Exemplo 4:

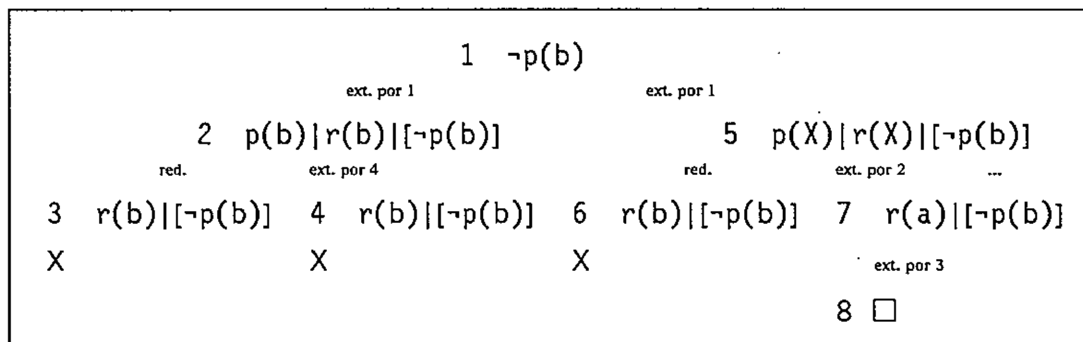
Este exemplo ilustra como o *"cut"* pode podar ramos após uma redução. Seja o programa abaixo:

```
1. p(X) | p(b) | r(X).
2. ¬p(a).
3. ¬r(a).
```

e a consulta "p(b)", que gera a cadeia:

4. ¬ p(b)

Seja a árvore de refutação, iniciando-se em (4):



Se alterarmos o programa anterior acrescentando à cláusula (1) o comando *"cut"*, obtemos:

1. p(X) | / | p(b) | r(X).

O ramo de fracasso que passa pelo nó (4) e os ramos seguintes que passam pelo nó (7), inclusive o de sucesso finalizado em (8), não são analisados devido à presença do *"cut"*. A máquina MEL retorna então falha.

Exemplo 5:

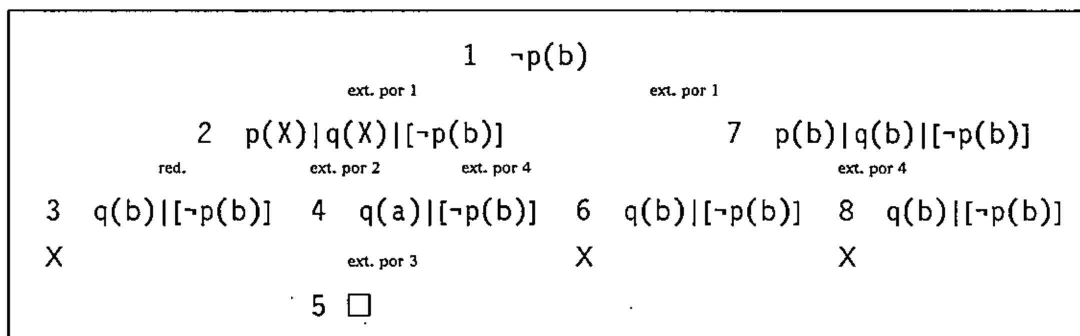
Este exemplo ilustra novamente o uso do *"cut"* influenciando a computação de respostas pela poda de possíveis ramos. Seja o programa:

1. p(b) | p(X) | / | q(X).
2. ¬p(a).
3. ¬q(a).

Seja a consulta "p(b)", que gera a cadeia:

4. ¬p(b).

Seja a árvore de refutação, desconsiderando o *"cut"*:



O ramo de sucesso terminado no nó 5 e os ramos de fracasso seguintes não são avaliados devido à presença do *"cut"* na cláusula (1). Percebemos então que uma possível resposta correta deixa de ser computada e a máquina MEL retorna falha.

4.2 Comando "¬%(*)"

Observamos que a linguagem MEL implementa a negação de um literal de acordo com o seu real sentido lógico. Se o usuário desejar, pode utilizar o comando "¬%(*)", que implementa a negação por falha finita.

4.3 Comando "mref"

O comando *mref* verifica se as cláusulas na área de trabalho formam um programa inconsistente, quando obtém sucesso, ou não, falhando. O procedimento adotado é tentar refutar todas as cláusulas, uma a uma.

4.4. Comandos "call(*)" e "macro(*)"

O comando *macro* possui a forma "macro(E)", onde E é uma disjunção de variáveis ou literais, e o comando *call* é uma expressão da forma "call(E)", onde E é uma conjunção de variáveis ou literais.

Para acomodar estes extra-lógicos, a máquina MEL é modificada da seguinte forma. Se, durante a construção de uma árvore, o rótulo de um nó *N* for da forma

$$\mathrm{macro}(C) \mid L_1 \mid \ldots \mid L_m$$

a máquina produzirá apenas um descendente para *N*, rotulado com

$$C \mid L_1 \mid \ldots \mid L_m$$

se C for uma cláusula MEL. Se C não for uma cláusula, o nó não tem descendentes.

Já se o rótulo de um nó *N* for da forma

$$\mathrm{call}(D) \mid L_1 \mid \ldots \mid L_m$$

onde D é uma conjunção de literais, a máquina chamará a si própria recursivamente, tendo como entrada o próprio programa inicial e a consulta D. Como D é uma conjunção de literais, cada resposta da chamada recursiva terá a forma genérica $D\beta_1 \mid \ldots \mid D\beta_n$. Seja $\overline{D}$ a cadeia representando a negação de D. Para cada resposta da chamada recursiva da forma $D\beta_1 \mid \ldots \mid D\beta_n$, a máquina gerará um novo descendente de *N* rotulado com

$$\overline{D}\beta_1 \mid L_1\beta_1 \mid \ldots \mid L_m\beta_1 \mid \ldots \mid \overline{D}\beta_n \mid L_1\beta_n \mid \ldots \mid L_m\beta_n$$

O resto desta seção discute brevemente a diferença entre estes comandos e o "*call*" do Prolog apresentando uma justificativa para a semântica e sintaxe adotada.

Por exemplo, suponha a seguinte cláusula objetivo em Prolog:

```
<-p & call(q & r) & s.
```

Repare que a negação dos literais q e r está implícita devido às suas posições após o "<-" na cláusula. Este fato não ocorre em cláusulas MEL. Este mesmo exemplo poderia ser escrito de duas formas em MEL:

$\neg p \mid \mathrm{call}(q \,\&\, r) \mid \neg s.$ ou $\neg p \mid \mathrm{macro}(\neg q \mid \neg r) \mid \neg s.$

Podemos observar que o comando *call* captura a idéia de uma subconsulta, negando implicitamente seu argumento. Já o comando *macro* acompanha a idéia de uma macro expansão, admitindo seu argumento sem modificações.

Uma diferença relevante entre Prolog e MEL consiste no primeiro computar apenas respostas simples. De fato, o tratamento de respostas genéricas de modo uniforme e coerente com a definição teórica apresentada direcionou a implementação dos comandos *call* e *macro*.

Por exemplo, considere o seguinte programa:

```
1. q(a).
2. q(b).
3. p(a) | p(b).
```

e a consulta:

```
4.  call (p(X)) & q(X).
```

Note que o argumento do *"call"* possui uma resposta genérica, qual seja "p(a) | p(b)". Logo, não podemos expressar a resposta de "p(X)" como argumento do *"call"* pois seríamos obrigados a exibir a resposta a (4) conforme sugerido pela expressão:

```
call(p(a) | p(b)) & q(??).
```

A máquina MEL exibe então a resposta a (4) da seguinte forma:

```
call(p(a)) & q(a) | call(p(b)) & q(b).
```

Existem ainda outros problemas que justificam restringir o argumento do *"call"* a conjunções de literais. Por exemplo, seja o programa:

```
1. p(a).
2. p(b).
```

e a consulta:

```
3.  call (p(a) | p(b) | p(c))
```

A questão aqui envolve não só uma resposta genérica como também uma subconsulta com disjunção de literais. Uma solução do tipo:

```
call(p(a) | p(b))
```

contraria o conceito de resposta pois implica em uma "cirurgia" no único literal da consulta em (3) tendo em vista que "call(p(a) | p(b))" não é uma instância desta. Por este motivo, o argumento do comando *call* é restrito a conjunções de literais.

Por analogia, o argumento do comando *macro* é restrito a uma disjunção de literais.

4.5 Comando "@(*)"

Este extra-lógico só tem sentido em linguagens de programação em cláusulas genéricas, que trabalham com respostas genéricas. O comando "@(*)" tem semântica semelhante à do comando *call*: seu argumento deve ser uma conjunção de literais, tem função de uma subconsulta e há uma negação implícita. Mas, ao contrário do *call*, o comando "@(*)" só retorna respostas definidas da subconsulta. Caso esta não possua respostas definidas, ele falha.

Exemplo 6:

Seja o programa:

```
1. p(a) | p(b).
2. q(a) | q(b).
3. q(c).
```

e a consulta:

```
4.  @(p(X)).
```

Como a resposta à subconsulta "p(X)." é genérica, a máquina MEL retorna falha. Suponha agora a consulta:

```
4.  @(q(X)).
```

A subconsulta "q(X)." possui a resposta simples "q(c)" e a máquina MEL retorna sucesso com "@(q(c))."

4.6 Comando "\(*,*)"

O comando "\(*,*)" também possui semântica semelhante à do comando *call*, sendo seu primeiro argumento uma subconsulta, conjunção de literais, cuja resposta genérica é retornada em seu segundo argumento, na forma de uma lista de disjuntos.

Exemplo 7:

Seja o programa anterior:

```
1. p(a) | p(b).
2. q(a) | q(b).
3. q(c).
```

e a consulta:

```
4.  \(p(X),Y).
```

A máquina MEL retorna \(p(X),<p(a),p(b)>).

5. CONCLUSÕES

A linguagem MEL pode ser vista como uma extensão coerente de Prolog para cláusulas genéricas, preservando o significado clássico da negação. Porém, se o programador assim o desejar, poderá invocar a negação por falha finita explicitamente através do extra-lógico "¬%". MEL oferece ainda uma expansão do conceito de "cut" para as árvores de refutação por eliminação de modelos, bem como outros extra-lógicos interessantes, como "call" e "macro".

Por fim, observamos que o protótipo de um interpretador para MEL encontra-se totalmente operacional, permitindo o uso experimental da linguagem.

REFERÊNCIAS BIBLIOGRÁFICAS

Casanova, M.A., F.A.C. Giorno e A.L. Furtado [1987]. *Programação em Lógica e a Linguagem Prolog*, Ed. Blucher.

Casanova, M.A.,Guerreiro, R.A.T. e A. Silva [1988]. "Computação de respostas em sistemas baseados em eliminação de modelos e defaults" (em preparação).

Casanova, M.A. e Walter, M.E.M.T. [1986]. "Uma implementação do Método de Eliminação de Modelos", anais do 3° Simpósio Brasileiro de Inteligência Artificial, Rio de Janeiro, 311-322.

Lloyd, J.W. [1984]. *Foundations of Logic Programming*, Springer-Verlag.

Loveland, D.W. [1968]. "Mechanical theorem-proving by model elimination", *Journal of the ACM* **15:2**, 236-251.

Loveland, D.W. [1969]. "A simplified format for the model elimination theorem-proving procedure", *Journal of the ACM* **16:3**, 349-363.

Loveland, D.W. [1978]. *Automated Theorem Proving: a Logical Basis*, North-Holland Publishing Company, Amsterdam.

Luckham, D.W. e N.J. Nilsson [1971]. "Extracting Information from Resolution Trees", *Artificial Intelligence* **2**, 27-54.