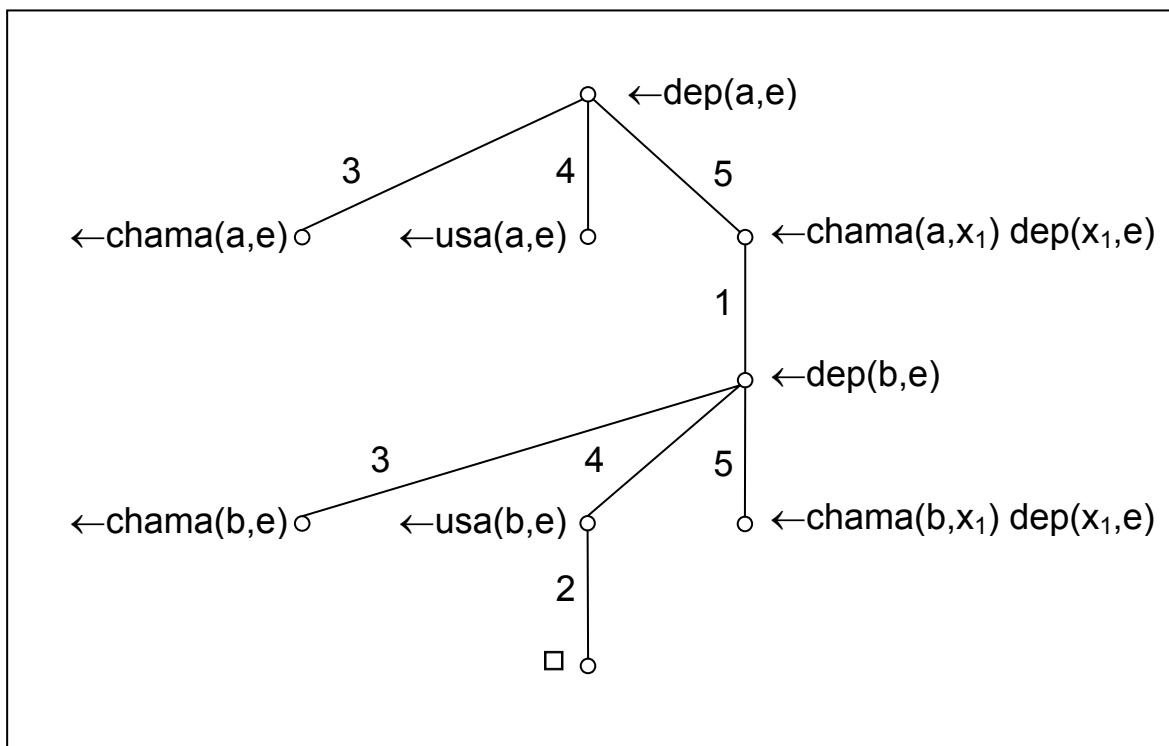


CASANOVA - GIORNO - FURTADO

PROGRAMAÇÃO EM LÓGICA E A LINGUAGEM PROLOG



PROGRAMAÇÃO EM LÓGICA E A LINGUAGEM PROLOG

**MARCO A. CASANOVA
FERNANDO A.C. GIORDO
ANTONIO L. FURTADO**

PROGRAMAÇÃO EM LÓGICA E A LINGUAGEM PROLOG

© 2006 Marco A. Casanova
 Fernando A.C. Giorno
 Antonio L. Furtado

*É permitida a reprodução deste texto para
uso pessoal.*

PREFÁCIO

ORGANIZAÇÃO DO TEXTO

Este texto divide-se em quatro partes, sendo que as três primeiras cobrem os fundamentos de Programação em Lógica enquanto que a última oferece uma descrição precisa da linguagem Prolog (PROgramming in LOGic), a versão do paradigma de Programação em Lógica mais difundida no momento.

Em detalhe, a primeira parte comprehende os capítulos 1 a 3 e apresenta conceitos de Lógica Matemática. O capítulo 1 cobre o essencial de Lógica Sentencial. O capítulo 2 aborda Lógica de Primeira Ordem, descrevendo a sintaxe e a semântica de linguagens de primeira ordem e incluindo uma breve discussão sobre sistemas formais e problemas de decisão. O capítulo 3 introduz a noção de cláusula e apresenta duas provas do Teorema de Herbrand. Esta parte inicial do texto é auto-contida, procurando tornar acessíveis os (poucos) conceitos mais complexos através de vários exemplos. Porém, ela é especialmente importante pois oferece os conceitos necessários a uma definição precisa do que significa programar em lógica e, através da discussão sobre decidibilidade, impõe limitações no que se pode esperar de Programação em Lógica.

A segunda parte, consistindo dos capítulos 4 a 7, enuncia certos resultados fundamentais em Prova Automática de Teoremas que facilitam o entendimento do funcionamento dos sistemas para Programação em Lógica existentes. O capítulo 4 discute em detalhe o sistema formal da resolução. O capítulo 5 apresenta o método da resolução linear e alguns outros métodos simples de resolução. O capítulo 6 cobre métodos baseados

na técnica de eliminação de modelos. O capítulo 7 apresenta o método da resolução com função de seleção, incluindo ainda uma discussão importante sobre um tratamento alternativo da negação, primordial para o entendimento da linguagem Prolog. Esta segunda parte separa de forma clara as noções de sistema formal, método de dedução e procedimento de refutação, um ponto que se não for levado em consideração dificulta o entendimento das inúmeras variações de resolução e de eliminação de modelos.

A terceira parte consiste dos capítulos 8 e 9 e aborda os conceitos de programa em lógica, consulta e resposta. O capítulo 8 investiga estes conceitos quando os programas são expressos por conjuntos finitos de cláusulas genéricas, enquanto que o capítulo 9 indica que simplificações são possíveis quando os programas consistem apenas de cláusulas definidas. Esta parte enfatiza principalmente o problema de dar uma semântica adequada para programas em lógica e o problema de computar respostas corretas através de procedimentos de refutação. A investigação destes problemas é fundamental pois distingue Programação em Lógica das áreas tradicionais de Lógica Matemática e de Prova Automática de Teoremas.

A quarta e última parte abrange os capítulos 10, 11 e 12 e os apêndices. O capítulo 10 apresenta a sintaxe e a semântica da linguagem Prolog básica. O capítulo 11 completa a descrição de Prolog introduzindo as facilidades extra-lógicas mais comuns aos diversos dialetos Prolog existentes. O capítulo 12 contém exemplos do uso de Prolog e uma comparação de Prolog com outras linguagens. Os apêndices apresentam um resumo dos comandos disponíveis nos diversos dialetos da linguagem Prolog, enunciados de exercícios e resolução de exercícios selecionados. Quando comparada com outros textos sobre Prolog, esta parte distingue-se por apresentar a linguagem básica de forma clara e precisa, beneficiando-se dos conceitos de Lógica e Prova Automática de Teoremas adquiridos nos capítulos anteriores.

UTILIZAÇÃO DO TEXTO

A estrutura do texto permite utilizá-lo tanto para autoestudo quanto para cursos formais, exigindo do leitor prerequisitos distintos. Algumas sugestões, ordenadas em grau crescente de dificuldade, são:

- Para os leitores interessados em uma rápida introdução à Programação em Lógica, ou para cursos rápidos de 12 a 20 horas, sugere-se cobrir apenas as seções indicadas como introdutórias no começo de cada capítulo. A maturidade matemática exigida é aquela esperada de um profissional de Informática.
- Para os leitores interessados apenas na linguagem Prolog, sugere-se uma leitura rápida das seções dos capítulos 2 a 9 marcadas como introdutórias e um estudo cuidadoso dos capítulos 10, 11 e 12. O apêndice II contém os exercícios necessários a este tipo de leitura. Não é necessário nenhum conhecimento prévio de Lógica ou de programação, exceto para a seção 12.3.
- Para um curso de 1 semestre a nível de graduação, sugere-se cobrir as seções marcadas como intermediárias. A maturidade matemática exigida é neste caso equivalente àquela de um bom curso de Cálculo.
- Para um curso sobre Prova Automática de Teoremas de 1 semestre, sugere-se cobrir os capítulos de 1 a 7. Esta parte do texto não tem como pré-requisito conhecimento prévio de Lógica Matemática e exige do leitor maturidade matemática apenas para acompanhar as demonstrações contidas nas seções avançadas.
- Finalmente, o texto completo contém material suficiente para um curso sobre Programação em Lógica de 1 semestre a nível de Mestrado ou Doutorado. O texto é praticamente autocontido e novamente exige alguma maturidade matemática.

AGRADECIMENTOS

Este texto foi escrito como parte das atividades dos autores dentro do Centro Científico do Rio de Janeiro e enquanto o terceiro autor (ALF) encontrava-se em licença da Pontifícia Universidade Católica do Rio de Janeiro. Por apoiar integralmente o nosso empreendimento, os autores gostariam de agradecer inicialmente à IBM Brasil e, especialmente, ao Sr. José Roberto Giannini de Freitas, gerente do Centro Científico na época em que o projeto começou e atualmente gerente do Centro de Tecnologia de Software da IBM, ao Sr. Fernando Borges Fortes, gerente do Centro Científico, e ao Dr. Almir Lopes de Almeida, gerente de pesquisa do Centro Científico a quem os autores se reportam.

Uma primeira versão deste texto foi preparada para a Quinta Escola de Computação. Pela confiança depositada no trabalho, os nossos agradecimentos estendem-se à comissão organizadora deste evento e, principalmente, ao seu coordenador, Prof. Alberto Henrique Frade Laender.

Os nossos colegas Arnaldo Vieira Moura, Luiz Tucherman, Raul Cesar Baptista Martins, Fernando Antonio Soeiro, Carlos Roberto de Souza e Ramiro Affonso de Tadeu Guerreiro, da IBM, e Maria Emilia Machado Telles Walter e Leonardo Lazarte, da Universidade de Brasília, contribuíram para a clareza e correção do texto com inúmeras sugestões.

Este texto foi composto utilizando o "Document Composition Facility", programa produto IBM, e os originais gerados em uma impressora IBM 3800 modelo 3. A utilização desta impressora, que muito contribuiu para a qualidade gráfica do texto, só foi possível graças aos esforços do nosso colega Arnaldo Viegas de Lima.

Rio, junho de 1987

M.A.C
F.A.C.G
A.L.F

ÍNDICE

CAPÍTULO 0: INTRODUÇÃO	1
0.1 O QUE É PROGRAMAÇÃO EM LÓGICA	1
0.2 EXEMPLO DE UM PROGRAMA EM LÓGICA	3
0.2.1 Descrição Informal	3
0.2.2 Descrição Formal	4
PARTE I - FUNDAMENTOS DE LÓGICA	11
CAPÍTULO 1: LÓGICA SENTENCIAL	13
1.1 INTRODUÇÃO	13
1.2 SINTAXE DAS LINGUAGENS PROPOSICIONAIS	14
1.3 SEMÂNTICA DAS LINGUAGENS PROPOSICIONAIS	19
1.4 O MÉTODO DA TABELA-VERDADE	24
CAPÍTULO 2: LÓGICA DE PRIMEIRA ORDEM	27
2.1 INTRODUÇÃO	27
2.2 SINTAXE DAS LINGUAGENS DE PRIMEIRA ORDEM	29
2.3 SEMÂNTICA DAS LINGUAGENS DE PRIMEIRA ORDEM	36
2.4 TEORIAS DE PRIMEIRA ORDEM	44
2.5 EXEMPLOS DE TEORIAS	46
2.5.1 Um Dicionário de Programas e Arquivos	46
2.5.2 Uma Teoria para Listas	50
2.6 UM SISTEMA AXIOMÁTICO	55
2.7 PROBLEMAS DE DECISÃO	59
CAPÍTULO 3: NOTAÇÃO CLAUSAL E O TEOREMA DE HERBRAND	65
3.1 INTRODUÇÃO	65
3.2 CLÁUSULAS	66
3.3 REPRESENTAÇÃO CLAUSAL DE FÓRMULAS E TEORIAS	69
3.3.1 Representação Clausal de Fórmulas	69
3.3.2 Representação Clausal de Teorias	77
3.4 PRIMEIRO ENUNCIADO DO TEOREMA DE HERBRAND	83
3.4.1 Estruturas de Herbrand	83
3.4.2 Uma Primeira Prova do Teorema de Herbrand	85

3.5 SEGUNDO ENUNCIADO DO TEOREMA DE HERBRAND	88
3.5.1 H-Interpretações	88
3.5.2 Árvores Semânticas	91
3.5.3 Uma Segunda Prova do Teorema de Herbrand	95
3.6 PROCEDIMENTOS DE REFUTAÇÃO BASEADOS NO TEOREMA DE HERBRAND	96
 PARTE II - PROVA AUTOMÁTICA DE TEOREMAS	103
 CAPÍTULO 4: RESOLUÇÃO	105
4.1 INTRODUÇÃO	105
4.2 O QUE É RESOLUÇÃO	106
4.3 UNIFICAÇÃO	112
4.4 O ALGORITMO DE UNIFICAÇÃO	116
4.5 CORREÇÃO DO ALGORITMO DE UNIFICAÇÃO	124
4.6 O SISTEMA FORMAL DA RESOLUÇÃO	129
4.7 CORREÇÃO E COMPLETITUDE DO SISTEMA FORMAL DA RESOLUÇÃO	134
 CAPÍTULO 5: MÉTODOS DE REFUTAÇÃO POR RESOLUÇÃO	143
5.1 INTRODUÇÃO	143
5.2 MÉTODOS E PROCEDIMENTOS DE DEDUÇÃO	144
5.3 MÉTODOS ELEMENTARES DE RESOLUÇÃO	146
5.3.1 Resolução por Saturação	146
5.3.2 Resolução por Saturação com Filtragem	147
5.3.3 Resolução com Conjunto de Suporte	152
5.4 O MÉTODO DA RESOLUÇÃO LINEAR	154
5.5 COMPLETITUDE DO MÉTODO DA RESOLUÇÃO LINEAR	158
5.6 PROCEDIMENTOS DE REFUTAÇÃO POR RESOLUÇÃO LINEAR	161
 CAPÍTULO 6: ELIMINAÇÃO DE MODELOS	169
6.1 INTRODUÇÃO	169
6.2 UM EXEMPLO INICIAL	170
6.3 O SISTEMA FORMAL DA ELIMINAÇÃO DE MODELOS	172
6.4 O MÉTODO DA ELIMINAÇÃO DE MODELOS FRACA	178
6.4.1 Definição do Método	178
6.4.2 Resumo das Principais Características do Método	184
6.5 PROCEDIMENTOS DE REFUTAÇÃO POR ELIMINAÇÃO DE MODELOS FRACA	186
 CAPÍTULO 7: RESOLUÇÃO-LSD E NEGAÇÃO POR FALHA FINITA	195
7.1 INTRODUÇÃO	195
7.2 O MÉTODO DA RESOLUÇÃO-LSD	196

7.3 CORREÇÃO E COMPLETITUDE DO MÉTODO DA RESOLUÇÃO-LSD	203
7.4 UMA INTERPRETAÇÃO INTUITIVA PARA RESOLUÇÃO-LSD	207
7.5 O MÉTODO DA RESOLUÇÃO-LSDNF	209
7.6 CORREÇÃO E COMPLETITUDE DO MÉTODO DA RESOLUÇÃO-LSDNF	216
7.6.1 Correção	216
7.6.2 Completude	222
PARTE III - PROGRAMAÇÃO EM LÓGICA	227
CAPÍTULO 8: PROGRAMAÇÃO EM CLÁUSULAS GENÉRICAS	229
8.1 INTRODUÇÃO	229
8.2 SINTAXE E SEMÂNTICA DECLARATIVA	231
8.3 SEMÂNTICA PROCEDIMENTAL INDUZIDA POR RESOLUÇÃO	236
8.3.1 Exemplos de Computação de Respostas	236
8.3.2 Correção e Completude da Computação de Respostas	242
8.4 SEMÂNTICA PROCEDIMENTAL INDUZIDA POR ELIMINAÇÃO DE MODELOS	247
8.5 BANCOS DE DADOS DEDUTIVOS	249
CAPÍTULO 9: PROGRAMAÇÃO EM CLÁUSULAS DEFINIDAS	257
9.1 INTRODUÇÃO	257
9.2 SINTAXE E SEMÂNTICA DECLARATIVA	259
9.3 SEMÂNTICA DO MODELO MÍNIMO	261
9.4 SEMÂNTICA PROCEDIMENTAL INDUZIDA POR RESOLUÇÃO-LSD	268
9.5 EXTENSÃO PARA CLÁUSULAS PSEUDO-DEFINIDAS	274
9.6 MODELAGEM DE ALGORITMOS	278
PARTE IV - A LINGUAGEM PROLOG	283
CAPÍTULO 10: A LINGUAGEM PROLOG BÁSICA	285
10.1 INTRODUÇÃO	285
10.2 SINTAXE DA LINGUAGEM PROLOG BÁSICA	288
10.2.1 Sintaxe das Cláusulas Prolog	289
10.2.2 Sintaxe dos Operadores Prolog	296
10.3 SEMÂNTICA DE UM PROGRAMA PROLOG BÁSICO	299
10.3.1 Semânticas Declarativa e Procedimental	299
10.3.2 Semântica Operacional	302
10.4 CODIFICAÇÃO DE CLÁUSULAS PROLOG	313
10.4.1 Área de Trabalho Prolog	313
10.4.2 Termos Prolog	314
10.4.3 Cláusula Unitária Básica	315

10.4.4 Cláusula Objetivo	318
10.4.5 Cláusula Não-Unitária	320
10.5 PROCESSAMENTO DE LISTAS	324
10.5.1 Sintaxe para Listas	324
10.5.2 Operações Básicas sobre Listas	327
10.5.3 Outras Operações sobre Listas	331
CAPÍTULO 11: A LINGUAGEM PROLOG ESTENDIDA	341
11.1 INTRODUÇÃO	341
11.2 FACILIDADES ARITMÉTICAS E DE COMPARAÇÃO	342
11.3 FACILIDADES EXTRALÓGICAS DE COMUNICAÇÃO	345
11.4 FACILIDADES EXTRALÓGICAS DE CONTROLE	349
11.4.1 Comando "cut"	349
11.4.2 Outros Comandos Extralógicos de Controle	354
11.5 FACILIDADES EXTRALÓGICAS DE PROCESSAMENTO DE CLÁUSULAS	360
11.6 FACILIDADES EXTRALÓGICAS DE DEPURAÇÃO	365
11.7 PROGRAMAÇÃO NA METALINGUAGEM	373
CAPÍTULO 12: EXEMPLOS DO USO DE PROLOG	389
12.1 GERAÇÃO DE PLANOS	389
12.2 DESENVOLVIMENTO DE APLICAÇÕES DE BANCOS DE DADOS	396
12.3 COMPARAÇÃO DE PROLOG COM OUTRAS LINGUAGENS ..	406
APÊNDICE I: COMANDOS EXTRA-LÓGICOS E OPERADORES PREDEFINIDOS DE PROLOG	417
APÊNDICE II: EXERCÍCIOS DE PROLOG	429
II.1 ENUNCIADO DOS EXERCÍCIOS	429
II.2 SOLUÇÃO DE EXERCÍCIOS SELECIONADOS	434
REFERÊNCIAS BIBLIOGRÁFICAS	447
Índice Remissivo	455

CAPÍTULO 0: INTRODUÇÃO

Este primeiro capítulo descreve brevemente os principais conceitos de Programação em Lógica e apresenta um exemplo simples de um programa em lógica. Em particular, a seção 0.2.2 ilustra, através de três formalizações distintas do exemplo, a seqüência de tópicos seguida pelo texto.

Este capítulo deverá ser coberto em qualquer nível de leitura, exceto possivelmente a seção 0.2.2.

0.1 O QUE É PROGRAMAÇÃO EM LÓGICA

Um programa em lógica é um modelo de um determinado problema ou situação expresso através de um conjunto finito de sentenças lógicas. Ao contrário de programas em FORTRAN ou Pascal, por exemplo, um programa em lógica não é, portanto, a descrição de um procedimento para obter soluções de um problema. De fato, o interpretador ou compilador utilizado para processar os programas em lógica fica inteiramente responsável pelo procedimento adotado para pesquisa de soluções. Um programa em lógica assemelha-se mais a um banco de dados, exceto que as afirmações em um banco de dados descrevem apenas observações como "João é gerente de Pedro", enquanto que as sentenças de um programa em lógica podem também ter um escopo mais genérico, como "o gerente de um funcionário é superior hierárquico do funcionário" e "o gerente de um

superior hierárquico de um funcionário é superior hierárquico do funcionário".

Programação em Lógica exemplifica assim um estilo mais fundamental, que pode ser chamado de Programação Declarativa (Assercional ou Não-Procedimental), em contraste com Programação Procedimental (ou Imperativa), típica das linguagens tradicionais. Programação Declarativa engloba também Programação Funcional, que tem em LISP o seu exemplo mais conhecido, e quase todas as linguagens mais recentes para consulta a bancos de dados, como SQL e QUEL. Lembrando que LISP data de 1960, Programação Funcional é então um estilo conhecido há bastante tempo ao contrário de Programação em Lógica, que só ganhou ímpeto depois de 1972 com o advento da linguagem Prolog (PROgramming in LOGic), abordada neste texto.

Os conceitos de chamada (ou consulta) e de resposta naturalmente também diferem das noções tradicionais. De fato, uma consulta a um programa em lógica é uma afirmação exprimindo as condições a serem satisfeitas por uma resposta correta em presença da informação descrita pelo programa.

Porém, o ponto fundamental de Programação em Lógica consiste em identificar a noção de computação com a noção de dedução. Mais precisamente, a maioria dos sistemas para Programação em Lógica reduzem a busca de respostas corretas à pesquisa de refutações a partir das sentenças do programa e da negação da consulta (uma refutação é uma dedução de uma contradição). Tais sistemas baseiam-se diretamente em procedimentos para pesquisa de refutações, estudados em Prova Automática de Teoremas, e em resultados de Programação em Lógica mostrando como extrair respostas corretas de refutações.

Assim, a resposta de uma consulta a um programa em lógica não se limita apenas a indicar que uma suposição acerca da informação contida no programa é falsa ou verdadeira. A resposta efetivamente exibe informação extraída do programa e pode vir acompanhada de uma explicação sobre como foi obtida, expressa em termos da refutação que a gerou.

0.2 EXEMPLO DE UM PROGRAMA EM LÓGICA

0.2.1 Descrição Informal

Considere o problema de descrever um catálogo de programas indicando a linguagem de programação em que cada programa é escrito. Suponha que, no momento, toda informação conhecida acerca dos programas seja descrita pelas seguintes afirmações:

- I₁. A é um programa em Fortran
- I₂. B é um programa em Prolog
- I₃. todo programa em Fortran é procedural
- I₄. todo programa em Pascal é procedural
- I₅. todo programa em Prolog é declarativo

Note que I₁ e I₂ representam informação factual e que I₃, I₄ e I₅ capturam informação genérica. O conjunto destas cinco afirmações, que chamaremos de CATÁLOGO, corresponde a um programa em lógica.

A seguinte afirmação:

- C. x é procedural

descreve uma consulta a CATÁLOGO.

Na maioria dos sistemas para Programação em Lógica, a execução do programa CATÁLOGO a partir de C corresponde à pesquisa de uma refutação a partir da negação de C e das afirmações em CATÁLOGO. Por exemplo, uma particular refutação é a seguinte seqüência de afirmações:

- R₁. A é um programa em Fortran
- R₂. todo programa em Fortran é procedural
- R₃. x não é procedural
- R₄. x não é um programa em Fortran
- R₅. contradição

Note que R₁ e R₂ são afirmações de CATALAGO, R₃ é a negação da consulta C, R₄ segue de R₂ e R₃ e, finalmente, R₅ segue de R₄ e R₁, substituindo-se x por A em R₄.

Na verdade, a substituição de x por A na refutação indica que A é um objeto que satisfaz às condições da consulta C em presença das afirmações no programa, ou seja, que A é uma resposta correta.

0.2.2 Descrição Formal

Esta seção apresenta formalizações do exemplo da seção 0.2.1 no contexto de:

- Programação em Linguagens de Primeira Ordem
- Programação em Cláusulas
- Programação em Cláusulas Definidas

O leitor poderá omitir esta seção em uma primeira leitura, embora ela seja útil também por indicar a seqüência de tópicos seguida pelo texto.

A formalização do exemplo começa com a escolha de uma linguagem (formal) apropriada. Utilizaremos neste texto, em primeira instância, *linguagens de primeira ordem*, definidas da seguinte forma. O alfabeto de uma linguagem de primeira ordem consiste de uma série de *símbolos lógicos*, incluindo *variáveis*, *conectivos lógicos* e *quantificadores*, e uma série de *símbolos não-lógicos*, incluindo *constantes* para denotar objetos específicos, *símbolos predicativos* para denotar relacionamentos entre objetos e *símbolos funcionais* para denotar funções sobre objetos. Apenas os símbolos não-lógicos variam de linguagem para linguagem, já que os símbolos lógicos são sempre parte do alfabeto por definição.

Na sua forma mais geral, explorada no Capítulo 2, as sentenças de uma linguagem de primeira ordem são os *termos* e as *fórmulas*. Um termo é uma variável, uma constante, ou uma expressão da forma $f(t_1, \dots, t_m)$, onde f é um símbolo funcional admitindo m argumentos e t_1, \dots, t_m são termos. Uma fórmula é ou uma *fórmula atômica* da forma $p(t_1, \dots, t_n)$, onde p é um símbolo predicativo admitindo n argumentos e t_1, \dots, t_n são termos, ou é uma expressão obtida compondo-se fórmulas através dos conectivos lógicos ou prefixando-se fórmulas com quantificadores.

Intuitivamente, os termos permitem denotar objetos do domínio do problema, diretamente através de seus nomes ou de variáveis, ou indiretamente através de funções. As fórmulas atômicas permitem expressar relações existentes entre objetos do domínio do problema. Finalmente, as fórmulas (não atômicas) permitem expressar propriedades das relações sobre o domínio do problema. Por exemplo, uma fórmula da

forma " $\forall x(P \rightarrow Q)$ " lê-se "para todo objeto x , se a propriedade P vale para x então a propriedade Q também vale para x ". Este será o único tipo de fórmula usado nos exemplos desta seção.

No caso específico do catálogo de programas, escolheremos um alfabeto de primeira ordem cujos símbolos não-lógicos incluem apenas constantes para denotar programas e linguagens de programação, um símbolo predicativo, *programa*, admitindo dois argumentos, e dois símbolos predicativos, *procedimental* e *declarativo*, admitindo apenas um argumento, de tal forma que:

- *programa(x,y)* lê-se " x é um programa escrito na linguagem y ";
- *procedimental(x)* lê-se " x é um programa procedimental";
- *declarativo(x)* lê-se " x é um programa declarativo".

Partindo das linguagens de primeira ordem, há várias possibilidades para o desenvolvimento de Programação em Lógica.

Em primeiro lugar, temos *Programação em Linguagens de Primeira Ordem* em que um programa é qualquer coleção finita de fórmulas de primeira ordem e uma consulta é qualquer fórmula de primeira ordem exprimindo as condições a serem satisfeitas por uma resposta correta.

Neste contexto, o seguinte conjunto de fórmulas define o programa apresentado na seção 0.2.1:

CATÁLOGO (Versão 1)

- $F_1. \quad programa(A,fortran)$
- $F_2. \quad programa(B,prolog)$
- $F_3. \quad \forall x(programa(x,fortran) \rightarrow procedural(x))$
- $F_4. \quad \forall x(programa(x,pascal) \rightarrow procedural(x))$
- $F_5. \quad \forall x(programa(x,prolog) \rightarrow declarativo(x))$

O leitor deve se convencer de que, usando a leitura intuitiva anteriormente indicada, as fórmulas acima possuem a mesma leitura que as afirmações da seção 0.2.1. Por exemplo, a fórmula F_5 tem exatamente o mesmo significado que a afirmação I_5 .

A fórmula abaixo por sua vez define a consulta apresentada na seção 0.2.1:

P. *procedimental(x)*

Embora conceitualmente importante, esta abordagem não será levada adiante neste texto pois a maioria dos sistemas para Programação em Lógica não a segue.

A segunda possibilidade para o desenvolvimento de Programação em Lógica, qualificada de *Programação em Cláusulas Genéricas* e abordada em detalhe no capítulo 8, assume que um programa é um conjunto finito de cláusulas e mantém a definição de consulta como qualquer fórmula de primeira ordem. Expressar programas apenas através de cláusulas foi um passo importante para a construção de sistemas práticos para Programação em Lógica, conforme ficará evidente ao longo dos capítulos 4 a 7.

A notação em cláusulas, embora muito mais simples, tem o mesmo poder de expressão que a notação das linguagens de primeira ordem. Porém, esta família de linguagens não deve ser ignorada pois, em certos casos, torna-se mais conveniente modelar primeiro o programa em lógica através de fórmulas, para depois mapeá-lo na notação de cláusulas.

Em detalhe, uma *cláusula* é uma lista $L_1 \dots L_n$ onde, para cada i no intervalo $[1, n]$, L_i é ou uma fórmula atômica ou a negação de uma fórmula atômica. A lista vazia é chamada de *cláusula vazia* e denotada por " \square ".

Uma cláusula da forma $L_1 \dots L_n$, cujas variáveis são x_1, \dots, x_k , deve ser lida como:

para todo x_1, \dots, x_k , L_1 ou ... ou L_n valem

Em particular, a cláusula vazia é sempre falsa, ou seja, representa uma contradição.

Uma refutação neste contexto reduz-se a uma seqüência de cláusulas terminando na cláusula vazia tal que cada cláusula ou pertence ao conjunto inicial dado ou é inferida a partir das cláusulas anteriores.

No contexto de Programação em Cláusulas Genéricas, o seguinte conjunto define o programa apresentado na seção 0.2.1:

CATÁLOGO (Versão 2)

- C₁. *programa(A,fortran)*
- C₂. *programa(B,prolog)*
- C₃. $\neg \text{programa}(x,\text{fortran}) \text{ procedural}(x)$
- C₄. $\neg \text{programa}(x,\text{pascal}) \text{ procedural}(x)$
- C₅. $\neg \text{programa}(x,\text{prolog}) \text{ declarativo}(x)$

Esta versão do programa é exatamente equivalente à anterior pois uma cláusula da forma " $\neg p(x) q(x)$ " é equivalente à formula " $\forall x(p(x) \rightarrow q(x))$ ".

A terceira possibilidade para o desenvolvimento de Programação em Lógica, chamada de *Programação em Cláusulas Definidas* e abordada em detalhe no capítulo 9, trabalha apenas com uma classe especial de cláusulas e adota uma notação própria. Esta variante facilita ainda mais a construção de sistemas para Programação em Lógica e forma a base da linguagem Prolog. Porém, sob alguns aspectos, perde poder de expressão quando comparada com Programação em Cláusulas Genéricas.

Mais precisamente, um programa neste enfoque é um conjunto finito de cláusulas definidas e uma consulta é uma conjunção de fórmulas atômicas.

Uma *cláusula definida* por sua vez é uma expressão da forma $A \leftarrow B_1 \dots B_n$ onde A, B_1, \dots, B_n são fórmulas atômicas. Em particular, quando $n=0$, a cláusula se reduz então à expressão $A \leftarrow$.

Se x_1, \dots, x_k são as variáveis ocorrendo na cláusula definida, então ela deve ser interpretada como:

para todo x_1, \dots, x_k , A vale se B_1 e ... e B_n valerem

Se $n=0$, então a interpretação se resume a:

para todo x_1, \dots, x_k , A vale

Uma *cláusula-objetivo*, é uma expressão da forma $\leftarrow B_1 \dots B_n$ onde B_1, \dots, B_n são fórmulas atômicas e $n > 0$. Se x_1, \dots, x_k são as variáveis ocorrendo na cláusula-objetivo, então ela deve ser interpretada como:

para nenhum x_1, \dots, x_k , B_1 e ... e B_n valem

ou seja, a cláusula nega a existência de objetos satisfazendo simultaneamente às condições B_1, \dots, B_n . Dito de outra forma, a cláusula representa a negação da consulta expressa pela fórmula $B_1 \wedge \dots \wedge B_n$.

A cláusula vazia, " \square ", também é considerada como uma cláusula-objetivo, sendo novamente sempre falsa.

Assim, o programa consistindo das cláusulas definidas abaixo especifica o catálogo:

CATÁLOGO (Versão 3)

- D₁. *programa(A,fortran) ←*
- D₂. *programa(B,prolog) ←*
- D₃. *procedimental(x) ← programa(x,fortran)*
- D₄. *procedimental(x) ← programa(x,pascal)*
- D₅. *declarativo(x) ← programa(x,prolog)*

Novamente, esta versão do programa é exatamente equivalente às anteriores pois uma cláusula definida da forma " $q(x) \leftarrow p(x)$ " é equivalente à formula " $\forall x(p(x) \rightarrow q(x))$ ". Porém, nem sempre é possível obter um conjunto de cláusulas definidas que seja equivalente a uma dada fórmula.

Este programa possui ainda a seguinte interpretação intuitiva. As cláusulas D₁ e D₂ correspondem às afirmações I₁ e I₂ da seção 0.2.1 e capturam informação diretamente conhecida; as cláusulas D₃ e D₄ correspondem a I₃ e I₄ e reduzem o problema de provar que x é um programa procedural ao problema de provar que x é escrito em Fortran ou ao problema de provar que x é escrito em Pascal; finalmente, a cláusula D₅ corresponde a I₅ e reduz o problema de provar que x é um programa declarativo ao problema de provar que x é escrito em Prolog.

Recorde que a execução de um programa para uma dada consulta corresponde à pesquisa de uma refutação a partir das cláusulas do programa e de cláusulas representando a negação da consulta. No caso do exemplo, uma particular refutação seria:

- R₁. *programa(A,fortran) ←*
- R₂. *procedimental(x) ← programa(x,fortran)*
- R₃. *←procedimental(x)*
- R₄. *←programa(x,fortran)*
- R₅. □

Esta refutação é idêntica à apresentada informalmente na seção 0.2.1. As cláusulas definidas em R₁ e R₂ pertencem ao programa e a cláusula-objetivo em R₃ representa a negação da consulta P. A cláusula em R₄ segue de R₂ e R₃ essencialmente porque:

- R₃ afirma que não há programas procedimentais;
- R₂ afirma que todo programa em Fortran é procedural;
- logo, R₂ e R₃ implicam em que não haja programas em Fortran, o que é expresso pela cláusula em R₄.

A cláusula em R₅ segue de R₁ e R₄ porque:

- R₄ afirma que não há programas em Fortran;
- R₁ afirma que A é um programa em Fortran;
- logo, R₁ e R₄ levam a uma contradição, o que é expresso pela cláusula vazia em R₅.

Os capítulos 10 e 11 apresentarão a linguagem Prolog como uma particular concretização das idéias de Programação em Cláusulas Definidas.

PARTE I - FUNDAMENTOS DE LÓGICA

CAPÍTULO 1: LÓGICA SENTENCIAL

Este capítulo apresenta em três seções os principais conceitos da Lógica Sentencial. As seções 1.2 e 1.3 apresentam a sintaxe e a semântica das linguagens proposicionais, respectivamente, e a seção 1.4 descreve o método da tabela-verdade.

Por conter material possivelmente conhecido, este capítulo não constitui pré-requisito obrigatório em nenhum nível de leitura.

1.1 INTRODUÇÃO

Lógica Sentencial ou Cálculo Proposicional formaliza a estrutura lógica mais elementar do discurso matemático, definindo precisamente o significado dos conectivos lógicos *não*, *e*, *ou*, *se ... então* e outros. A definição da Lógica Sentencial desdobra-se na especificação do que seja uma linguagem proposicional e na descrição de uma abstração adequada para os princípios lógicos que governam os conectivos.

Brevemente, o alfabeto de uma linguagem proposicional consiste dos conectivos lógicos, dos parênteses e de um conjunto de símbolos proposicionais. As regras sintáticas da linguagem definem o conjunto de fórmulas (bem formadas) como sendo ou os próprios símbolos proposicionais ou expressões construídas ligando tais símbolos através dos conectivos lógicos. As regras semânticas da linguagem capturam o

significado pretendido dos conectivos e associam a cada fórmula um dos valores-verdade, "falso" ou "verdadeiro".

Há várias formas de abstrair os princípios lógicos que governam os conectivos. A forma mais divulgada, o método da tabela-verdade, permite decidir, entre outras aplicações, se uma fórmula é sempre verdadeira. Uma outra forma consiste em definir um "cálculo" para inferir novas fórmulas a partir de outras. Por fim, existem ainda métodos de refutação para determinar se um conjunto de fórmulas leva a contradições, entre os quais encontramos o método dos tableaux analíticos e o método de resolução para Lógica Sentencial.

A origem da Lógica Sentencial remonta aos trabalhos de Boole (1815-1864) e de De Morgan (1806-1871) sobre o que veio a ser chamado de Álgebra de Boole. Porém, como o próprio nome indica, estes trabalhos estavam mais próximos de outras teorias matemáticas do que de Lógica. Devemos a Frege, no seu "Begriffsschrift", o enfoque de Lógica Sentencial como uma ferramenta para formalizar princípios lógicos (Frege chegou a ser criticado por esta mudança de enfoque).

1.2 SINTAXE DAS LINGUAGENS PROPOSICIONAIS

Em uma linguagem proposicional podemos capturar parte da estrutura lógica de trechos de discurso. Ou seja, podemos abstrair parágrafos consistindo de sentenças concatenadas por partículas *e*, *ou*, *se ... então* e outras com a mesma função. As sentenças recebem nomes, tirados de um conjunto de *símbolos proposicionais*. As partículas correspondem símbolos especiais, chamados de *conectivos*.

Dado um alfabeto *A*, uma *cadeia* sobre *A* é uma seqüência de símbolos de *A*. Uma *linguagem* sobre *A* é um conjunto de cadeias de *A*. De posse destas noções, definimos então:

Definição 1.1:

Um *alfabeto proposicional A* consiste de

símbolos lógicos:

<i>pontuação:</i>	(,)
<i>conectivos:</i>	\neg (negação) \wedge (conjunção) \vee (disjunção) \rightarrow (implicação) \equiv (bi-implicação ou bi-condicional)

símbolos não-lógicos:

um conjunto enumerável *P* de *símbolos proposicionais* diferentes dos símbolos lógicos.

Definição 1.2:

O conjunto das *fórmulas proposicionais* (ou simplesmente *fórmulas*) sobre um alfabeto proposicional *A* é o menor conjunto de cadeias de *A* satisfazendo às seguintes condições:

- (i) todo símbolo proposicional de *A* é uma fórmula sobre *A*;
- (ii) se *P* e *Q* são fórmulas sobre *A*, então $(\neg P)$, $(P \wedge Q)$, $(P \vee Q)$, $(P \rightarrow Q)$ e $(P \equiv Q)$ também são fórmulas sobre *A*.

Definição 1.3:

Uma fórmula *Q* é uma *subfórmula* de uma fórmula *P* se e somente se *Q* é uma subcadeia de *P*.

Definição 1.4:

A *linguagem proposicional* sobre um alfabeto proposicional *A*, denotada por *L(A)*, é o conjunto das fórmulas proposicionais sobre *A*.

O conjunto dos símbolos lógicos e as regras de formação para as fórmulas são fixados para todos os alfabetos proposicionais. Assim, para especificar um alfabeto proposicional *A* e, portanto, a linguagem proposicional *L(A)*, basta fixar o conjunto *P* dos símbolos proposicionais de *A*. Baseando-se nesta observação, frequentemente não faremos distinção entre *A*, *L(A)* e *P*, como nos textos tradicionais de Lógica. Por exemplo, especificaremos *L(A)* listando apenas *P* e diremos que os símbolos proposicionais em *P* são

símbolos proposicionais de $L(A)$. Finalmente, se não for necessário identificar o alfabeto A , denotaremos a linguagem proposicional $L(A)$ sobre A simplesmente por L .

Adotaremos as seguintes convenções notacionais:

OBJETO SINTÁTICO	CONVENÇÃO NOTACIONAL
símbolos proposicionais	letras maiúsculas do início do alfabeto (A, B, C, \dots)
fórmulas	letras maiúsculas do meio para o fim do alfabeto (P, Q, R, \dots)
conjuntos de fórmulas	letras maiúsculas em negrito do meio para o fim do alfabeto ($\mathbf{P}, \mathbf{Q}, \mathbf{R}, \dots$)
linguagens proposicionais	letras maiúsculas em itálico e negrito do meio do alfabeto (\mathcal{L}, \dots)

De acordo com a definição do conjunto de fórmulas, toda fórmula deverá estar completamente parentetizada. Porém, adotaremos as seguintes convenções sobre omissão de parênteses:

1. os parênteses mais externos podem ser omitidos;
2. a negação aplica-se à menor fórmula possível

Exemplo: $\neg A \wedge B$ abrevia $((\neg A) \wedge B)$

3. a conjunção e a disjunção aplicam-se à menor fórmula possível

Exemplo: $A \wedge B \rightarrow \neg C \vee D$ abrevia $((A \wedge B) \rightarrow ((\neg C) \vee$

4. quando um conectivo é usado repetidamente, o agrupamento é feito pela direita

Exemplo: $A \rightarrow B \rightarrow C$ abrevia $(A \rightarrow (B \rightarrow C))$

Apresentaremos a seguir dois exemplos simples do uso de linguagens proposicionais para capturar trechos do discurso. Em ambos os casos, o passo inicial consiste em selecionar um conjunto de símbolos proposicionais adequado, fixando assim a linguagem proposicional a ser usada. Cada símbolo está associado a uma frase, que é o seu significado pretendido. O

passo seguinte consiste em traduzir o trecho em questão para uma ou mais fórmulas, respeitando o significado pretendido dos símbolos.

Exemplo 1.1:

Considere a seguinte afirmação:

"Suponhamos que Sócrates está em tal situação que ele estaria disposto a visitar Platão, só se Platão estivesse disposto a visitá-lo; e que Platão está em tal situação que ele não estaria disposto a visitar Sócrates, se Sócrates estivesse disposto a visitá-lo, mas estaria disposto a visitar Sócrates, se Sócrates não estivesse disposto a visitá-lo".

Pergunta-se então "Sócrates está disposto a visitar Platão ou não?"

Podemos analisar este problema da seguinte forma. Construamos um alfabeto proposicional A cujos símbolos proposicionais são A e B , com o seguinte significado pretendido:

A - "Sócrates está disposto a visitar Platão"

B - "Platão está disposto a visitar Sócrates"

O trecho anterior desdobra-se então nos seguintes fatos, já expressos como fórmulas de $L(A)$:

Sócrates: $(B \rightarrow A)$

Platão: $(A \rightarrow \neg B) \wedge (\neg A \rightarrow B)$

Ou seja, estas duas fórmulas axiomatizam em Lógica Sentencial o nosso conhecimento acerca do estado de espírito de Sócrates e Platão.

A pergunta original resume-se então a saber qual das duas fórmulas A ou $\neg A$ segue desta axiomatização, independentemente do verdadeiro significado atribuído aos símbolos A e B .

Capturaremos, após os exemplos, precisamente o que significa "seguir de uma axiomatização" através do conceito de consequência lógica. Além disto, apresentaremos no final deste capítulo um procedimento mecânico, chamado de método da tabela-verdade, que permitirá resolver o problema apresentado neste exemplo de uma forma simples.

Exemplo 1.2:

Os sofistas, uma espécie de professores viajantes ('sofista' significava mais ou menos o mesmo que 'professor'), tornaram-se famosos na Grécia clássica por visitarem cidades ensinando, por um soldo, a arte de argumentar. Eles alcançaram grande fama e grande habilidade em argumentar a favor ou contra qualquer afirmação, não importando a sua veracidade. A arte que ensinavam, refletida hoje em dia no termo 'sofismático', que tem um sentido quase pejorativo, naquela época poderia ser vital pois, se um cidadão fosse acusado de um delito e tivesse que se apresentar perante um tribunal, caberia a ele se defender. Outra pessoa poderia naturalmente preparar a sua defesa, mas não o substituir na defesa em si. Além disto, os juízes não eram profissionais treinados, mas simples cidadãos escolhidos aleatoriamente e, portanto, bastante influenciáveis por defesas bem arquitetadas.

O mais famoso dos sofistas, Protágoras, nasceu em Abdera por volta de 480 AC e morreu por volta de 420 AC. Consta que Protágoras teve um discípulo brilhante, chamado Euathlus. Protágoras o ensinou a arte de argumentar por uma certa quantia, metade da qual seria paga imediatamente e metade após Euathlus ganhar o seu primeiro caso. Euathlus, porém, demorou a pagar a Protágoras, que o processou então.

Protágoras argumentou que se Euathlus ganhasse o caso, ganharia então o seu primeiro caso, logo deveria pagá-lo. Mas, se Euathlus não ganhasse o caso, deveria pagá-lo também pois era esta a questão em jogo.

Euathlus, que foi um bom aluno, argumentou da seguinte forma. Se ele ganhasse o caso, não deveria pagar pois Protágoras perderia a causa. Mas, se ele não ganhasse o caso, não estaria ganhando o seu primeiro caso e, portanto, também não deveria pagar a Protágoras.

Quem está com a razão então?

Os argumentos de Protágoras e Euathlus podem ser expressos em uma linguagem proposicional da seguinte forma. Os símbolos proposicionais do alfabeto são:

- A - Euathlus ganha o caso
- B - Euathlus ganha o seu primeiro caso
- C - Euathlus deve pagar a Protágoras

As fórmulas que capturam os argumentos são:

Argumento de Protágoras	Argumento de Euathlus
$A \rightarrow B$	$A \rightarrow \neg C$
$B \rightarrow C$	$\neg A \rightarrow \neg B$
$\neg A \rightarrow C$	$\neg B \rightarrow \neg C$
logo C	logo $\neg C$

ou, sob forma de fórmulas:

$$\text{Protágoras: } ((A \rightarrow B) \wedge (B \rightarrow C) \wedge (\neg A \rightarrow C)) \rightarrow C$$

$$\text{Euathlus : } ((A \rightarrow \neg C) \wedge (\neg A \rightarrow \neg B) \wedge (\neg B \rightarrow \neg C)) \rightarrow \neg C$$

A análise em Lógica Sentencial é inconclusiva, pois nenhum dos dois argumentos está incorreto e a pergunta sobre quem deveria ganhar a causa permanece tão em suspenso quanto antes.

Uma solução para Protágoras receber a quantia devida, sugerida por um advogado a Smullyan, um matemático que já foi mágico profissional, seria proceder da seguinte forma. Protágoras deveria processar Euathlus imediatamente após terminadas as aulas, exatamente como descrito acima, e deixar Euathlus ganhar o caso, que seria o seu primeiro. Em seguida, deveria processar novamente Euathlus para que lhe pagasse a quantia estipulada. Desta vez não haveria dúvida de que Protágoras sairia vencedor, pois Euathlus já havia ganho o seu primeiro caso.

1.3 SEMÂNTICA DAS LINGUAGENS PROPOSIACIONAIS

As fórmulas de uma linguagem proposicional, incluindo os símbolos proposicionais, terão como significado os valores-verdade **FALSO** ou **VERDADEIRO**, abreviados **F** e **V**, respectivamente. O significado de uma fórmula não é fixado a priori, mas dependerá de uma atribuição de

valores-verdade, especificada em separado, para os símbolos proposicionais da linguagem.

Recorde que uma linguagem proposicional $L(A)$ é o conjunto das fórmulas proposicionais sobre um alfabeto proposicional A .

Definição 1.5:

Seja A um alfabeto proposicional e P o conjunto de símbolos proposicionais de A . Uma *atribuição de valores-verdade* para A (ou simplesmente uma *atribuição* para A) é uma função $v: P \rightarrow \{F, V\}$.

Definição 1.6:

Seja A um alfabeto proposicional e v uma atribuição de valores-verdade para A . A *função de avaliação* para $L(A)$ induzida por v é a função $v: L(A) \rightarrow \{F, V\}$ definida da seguinte forma:

$$v(A) = v(A), \quad \text{se } A \text{ é um símbolo proposicional de } A$$

$$\begin{aligned} v(\neg P) &= V, && \text{se } v(P) = F \\ &= F, && \text{se } v(P) = V \end{aligned}$$

$$\begin{aligned} v(P \wedge Q) &= V, && \text{se } v(P) = v(Q) = V \\ &= F, && \text{em caso contrário} \end{aligned}$$

$$\begin{aligned} v(P \vee Q) &= F, && \text{se } v(P) = v(Q) = F \\ &= V, && \text{em caso contrário} \end{aligned}$$

$$\begin{aligned} v(P \rightarrow Q) &= F, && \text{se } v(P) = V \text{ e } v(Q) = F \\ &= V, && \text{em caso contrário} \end{aligned}$$

$$\begin{aligned} v(P \equiv Q) &= V, && \text{se } v(P) = v(Q) \\ &= F, && \text{em caso contrário} \end{aligned}$$

Note que, dada uma atribuição de valores-verdade, existe uma única função satisfazendo as condições acima. Esta função será sempre denotada pela letra manuscrita correspondente àquela usada para denotar a atribuição. Por abuso de linguagem, diremos que uma atribuição de valores-verdade v para A é uma atribuição de valores-verdade para $L(A)$, ou mesmo para L , quando o alfabeto A for subentendido.

A tabela abaixo resume a definição da semântica dos conectivos:

P	Q	$\neg P$	$P \wedge Q$	$P \vee Q$	$P \rightarrow Q$	$P \equiv Q$
V	V	F	V	V	V	V
V	F	F	F	V	F	F
F	V	V	F	V	V	F
F	F	V	F	F	V	V

A definição a seguir captura a noção clássica de consequência lógica. Intuitivamente, seja P um conjunto de fórmulas que expressa o nosso conhecimento proposicional acerca de um determinado universo de discurso e Q uma fórmula expressando uma hipótese acerca deste universo. Q será consequência lógica de P se e somente se, para qualquer estado do universo, se todas as fórmulas em P forem de fato verdadeiras neste estado, então Q também será verdadeira neste estado.

Os conceitos definidos abaixo apenas para conjuntos de fórmulas estendem-se naturalmente a uma fórmula Q tomando-se o conjunto como sendo $\{Q\}$.

Definição 1.7:

Seja A um alfabeto proposicional, P e Q conjuntos de fórmulas em $L(A)$ e R uma fórmula em $L(A)$.

- (a) R é *verdadeira* em uma atribuição de valores-verdade v para A se e somente se $v(R) = V$. Em caso contrário, R é *falsa*.
- (b) R é uma *tautologia* se e somente se, para toda atribuição de valores-verdade v para A , $v(R) = V$.
- (c) Uma atribuição de valores-verdade v para A *satisfaz* a P , ou v é um *modelo* para P , se e somente se, para toda fórmula S em P , $v(S) = V$.
- (d) P é *satisfatível* se e somente se existe uma atribuição de valores-verdade v para A que satisfaz P . Em caso contrário, P é *insatisfatível*.
- (e) R é uma *consequência lógica* de P , ou P *implica logicamente* R (notação: $P \models R$), se e somente se, para toda atribuição de valores-verdade v para A , se v satisfaz P então v satisfaz R .
- (f) P é *tautologicamente equivalente* a Q (notação: $P \models Q$) se e somente se toda fórmula de Q for consequência lógica de P e vice-versa.

Note que uma fórmula R é uma tautologia se e somente se R for consequência lógica do conjunto vazio. Assim, as tautologias são as verdades lógicas da Lógica Sentencial. Note ainda que, se P for um conjunto finito, digamos, $P = \{P_1, \dots, P_m\}$, então P implica logicamente R se e somente se $(P_1 \wedge \dots \wedge P_m) \rightarrow R$ for uma tautologia. Ou, na notação introduzida:

$$P \models R \text{ se e somente se } \models (P_1 \wedge \dots \wedge P_m) \rightarrow R$$

Além disto, se Q for um conjunto finito, digamos, $Q = \{Q_1, \dots, Q_n\}$, então P e Q são tautologicamente equivalentes se e somente se $(P_1 \wedge \dots \wedge P_m) \equiv (Q_1 \wedge \dots \wedge Q_n)$ for uma tautologia. Na notação introduzida:

$$P \equiv Q \text{ se e somente se } \models (P_1 \wedge \dots \wedge P_m) \equiv (Q_1 \wedge \dots \wedge Q_n)$$

Apresentaremos a seguir exemplos do uso destes conceitos.

Exemplo 1.3:

A tautologia mais trivial é chamada de *lei do terceiro excluído*: uma fórmula proposicional ou é falsa ou é verdadeira. Na verdade esta "lei" representa uma família de tautologias da forma $\neg P \vee P$, onde P é qualquer fórmula. Há uma série de outras "leis" úteis da Lógica Sentencial que podem ser apresentadas ou como tautologias da forma $P \equiv Q$, ou como pares de fórmulas P e Q que são tautologicamente equivalentes.

A tabela abaixo apresenta algumas destas leis, que serão usadas mais tarde (o símbolo '*' representa cada um dos conectivos: \wedge , \vee , \equiv):

P	Q	NOME
$A * (B * C)$	$(A * B) * C$	associatividade
$A * B$	$B * A$	comutatividade
$A \wedge (B \vee C)$	$(A \wedge B) \vee (A \wedge C)$	distributividade
$A \vee (B \wedge C)$	$(A \vee B) \wedge (A \vee C)$	distributividade
$A \rightarrow (B \rightarrow C)$	$(A \rightarrow B) \rightarrow (A \rightarrow C)$	distributividade
$A \rightarrow B$	$\neg A \vee B$	
$A \equiv B$	$(\neg A \vee B) \wedge (\neg B \vee A)$	De Morgan
$\neg(A \wedge B)$	$\neg A \vee \neg B$	De Morgan
$\neg(A \vee B)$	$\neg A \wedge \neg B$	
$\neg(A \rightarrow B)$	$A \wedge \neg B$	
$\neg(A \equiv B)$	$(A \wedge \neg B) \vee (\neg A \wedge B)$	
$\neg \neg A$	A	

Exemplo 1.4:

Recordemos que, no exemplo sobre Sócrates e Platão, a situação era descrita pelo conjunto P das duas fórmulas abaixo:

Sócrates: $(B \rightarrow A)$

Platão : $(A \rightarrow \neg B) \wedge (\neg A \rightarrow B)$

A pergunta feita no exemplo se resume então a determinar qual dos dois caso abaixo é verdade:

$$P \models A \quad \text{ou} \quad P \models \neg A$$

Ou seja, gostaríamos de saber se toda atribuição de valores-verdade que satisfaz P também satisfaz A ou, alternativamente, $\neg A$.

Note que a resposta poderá ser inconclusiva, ou seja, P pode não implicar logicamente tanto A quanto $\neg A$, ou mesmo P pode implicar logicamente A e $\neg A$ (neste caso, P será insatisfável. Por quê?).

Exemplo 1.5:

No exemplo sobre Protágoras e Euathlus, ambos os argumentos podiam ser expressos pelas fórmulas:

$$\text{Protágoras: } ((A \rightarrow B) \wedge (B \rightarrow C) \wedge (\neg A \rightarrow C)) \rightarrow C$$

$$\text{Euathlus : } ((A \rightarrow \neg C) \wedge (\neg A \rightarrow \neg B) \wedge (\neg B \rightarrow \neg C)) \rightarrow \neg C$$

Como ambas são tautologias, somos evidentemente levados a concluir que os argumentos de Protágoras e de Euathlus estão corretos, independentemente da atribuição de valores-verdade para A, B e C.

1.4 O MÉTODO DA TABELA-VERDADE

As duas seções anteriores definiram a sintaxe e a semântica das linguagens proposicionais e formalizaram ainda, do ponto de vista semântico, o conceito de implicação lógica (em Lógica Sentencial). Esta seção completa a apresentação da Lógica Sentencial, descrevendo um método sistemático para decidir implicação lógica, chamado de *método da tabela-verdade*.

O método da tabela-verdade baseia-se na observação de que, se P é um conjunto finito de fórmulas e Q é uma fórmula, há um número finito de símbolos proposicionais ocorrendo em Q e nas fórmulas em P . Logo, há um número finito de atribuições de valores-verdade distintas para estes símbolos. Assim, para decidir se Q é consequência lógica de P , basta enumerar todas estas atribuições e, para cada uma delas que satisfizer a todas as fórmulas em P , testar se ela também satisfaz a Q . Se esta condição for sempre observada, então podemos afirmar que Q é consequência lógica de P . Note que, se v não satisfizer a alguma fórmula em P , o valor que v atribui a Q não importará, pela forma como implicação lógica foi definida. Este método naturalmente também pode ser usado para decidir se uma fórmula é satisfatível ou se dois conjuntos finitos de fórmulas são tautologicamente equivalentes.

Exemplo 1.6:

Estabeleceremos através do método da tabela-verdade que $P \models Q$, onde P é a fórmula $A \rightarrow (B \rightarrow C)$ e Q é a fórmula $(A \rightarrow B) \rightarrow (A \rightarrow C)$.

A	B	C	$A \rightarrow (B \rightarrow C)$	$(A \rightarrow B) \rightarrow (A \rightarrow C)$
V	V	V	V	V
V	V	F	F	F
V	F	V	V	V
V	F	F	V	V
F	V	V	V	V
F	V	F	V	V
F	F	V	V	V
F	F	F	V	V

Cada linha da tabela corresponde a uma atribuição de valores-verdade, v , para os símbolos proposicionais A, B e C. As três primeiras entradas de uma linha exibem $v(A)$, $v(B)$ e $v(C)$. As outras entradas listam o valor de $v(P)$ e $v(Q)$ abaixo do conectivo principal de P ou Q. Como, para toda atribuição v , se $v(P) = V$ então $v(Q) = V$, temos que Q é consequência lógica de P (na verdade, P e Q são tautologicamente equivalentes. Por quê?).

Exemplo 1.7:

Voltando ao exemplo sobre Sócrates e Platão, decidiremos usando o método da tabela-verdade se P implica logicamente A ou não, onde P é o conjunto das fórmulas abaixo:

Sócrates: $(B \rightarrow A)$

Platão: $(A \rightarrow \neg B) \wedge (\neg A \rightarrow B)$

A	B	$(B \rightarrow A)$	$(A \rightarrow \neg B) \wedge (\neg A \rightarrow B)$
V	V	V	F
V	F	V	V
F	V	F	V
F	F	V	F

Apenas na segunda atribuição de valores-verdade ambas as fórmulas em P recebem o valor V. Como A também recebe o valor V nesta atribuição, temos que P implica logicamente A.

Por fim, note que, se n for o número de símbolos proposicionais ocorrendo em fórmulas de P ou na fórmula Q , a tabela-verdade possuirá 2^n linhas, pois há ao todo 2^n atribuições de valores-verdade distintas para n símbolos. Portanto, o método é exponencial no número de símbolos proposicionais ocorrendo nas fórmulas. Por outro lado, não se conhece um método uniforme para decidir o problema da implicação lógica para Lógica Sentencial que não seja exponencial. Este é, na verdade, um dos problemas em aberto mais importantes em Ciência da Computação atualmente.

NOTAS BIBLIOGRÁFICAS

Referências sobre Lógica Matemática são numerosas. Para Lógica Sentencial e Lógica Matemática em geral, usamos o texto clássico de Enderton [1972] como referência básica, complementado por Shoenfield [1967] e Kleene [1952]. Os exemplos mais interessantes deste capítulo são de DeLong [1970], com observações adicionais tiradas de Smullyan [1978].

Smullyan [1971] descreve em detalhe o método dos tableaux analíticos e Chang e Lee [1973] e Loveland [1978] descrevem o método da resolução para Lógica Sentencial.

CAPÍTULO 2: LÓGICA DE PRIMEIRA ORDEM

Este capítulo apresenta os conceitos e enuncia os resultados de Lógica de Primeira Ordem mais relacionados à Programação em Lógica. As seções 2.2 e 2.3 definem a sintaxe e a semântica das linguagens de primeira ordem, respectivamente. Em seguida, as seções 2.4 e 2.5 apresentam a noção de teoria de primeira ordem, incluindo dois exemplos deste conceito. A seção 2.6 discute brevemente a noção de sistema axiomático e os principais resultados da metateoria de Lógica de Primeira Ordem. Finalmente, a seção 2.7 discute as possibilidades de mecanização da Lógica de Primeira Ordem, sendo bastante importante em termos dos fundamentos de Programação em Lógica.

As seções recomendadas para cada nível de leitura são:

nível introdutório: 2.2, 2.3 (*em parte*)

nível intermediário: 2.2, 2.3

2.1 INTRODUÇÃO

Uma teoria consiste de um conjunto de assertivas, tradicionalmente chamadas de proposições, lemas, teoremas, etc... acerca de um universo de discurso.

A formalização de uma teoria divide-se em três etapas fundamentais. Inicialmente escolhe-se uma nova linguagem simbólica ou formal para exprimir a teoria, em substituição a uma linguagem natural. Na segunda etapa organiza-se as assertivas da teoria dedutivamente, isto é, escolhe-se algumas assertivas como axiomas ou postulados a partir das quais as outras podem ser gradativamente obtidas. Nesta etapa exprime-se também as propriedades dos termos do universo de discurso através de novos axiomas até que o significado destes termos não precise mais ser usado nas deduções. Na última etapa explicita-se o significado dos termos ordinários (fora do universo de discurso) e os princípios usados para decidir quando uma assertiva é consequência (lógica) de outras. O resultado destas etapas é uma teoria em que se abstraiu inteiramente o conteúdo ou significado dos termos, restando apenas a sua sintaxe ou forma. A teoria está então completamente formalizada.

Uma teoria formalizada contém portanto uma primeira parte relativa ao universo de discurso em questão e uma segunda parte relativa à linguagem formal e aos princípios lógicos adotados. Note que a primeira parte é intrínseca à teoria, mas a segunda parte poderia ser separada e usada novamente em outros contextos. Um sistema formal consiste exatamente de uma linguagem formal e de uma abstração adequada para os princípios usados para decidir quando uma assertiva é consequência (lógica) de outras.

Lógica de Primeira Ordem ou Cálculo dos Predicados, o tópico deste capítulo, pode ser caracterizada como um sistema formal apropriado à definição de teorias do universo de discurso da Matemática.

Para o desenvolvimento de Programação em Lógica, estes conceitos são bastante importantes pois um programa em lógica, na sua forma mais geral, nada mais é do que uma teoria completamente formalizada e descrita por um número finito de axiomas. Além disto, os sistemas para Programação em Lógica baseiam-se em resultados sobre a mecanização de Lógica de Primeira Ordem, conforme ficará evidente ao longo deste texto.

Lógica de Primeira Ordem originou-se no "Begriffsschrift" publicado por Frege em 1879. Por volta de 1930, três matemáticos, Gödel, Herbrand e Skolem obtiveram resultados importantes sobre o problema de caracterizar adequadamente os princípios lógicos de primeira ordem, partindo de resultados de Löwenheim e do próprio Skolem anteriores a 1920. Gödel, em sua tese de doutorado "Über die Vollständigkeit des Logikkalküls", defendida em 6 de fevereiro de 1930, provou a existência de um sistema

axiomático consistente e completo para Lógica de Primeira Ordem. Herbrand, também na sua tese de doutorado "Recherches sur la théorie de la démonstration", completada em abril de 1929 mas defendida apenas em 11 de junho de 1930, obteve uma solução para o problema da caracterização dos princípios lógicos de primeira ordem que se tornou fundamental para a mecanização da lógica. Finalmente, Skolem, em uma série de trabalhos publicados também em torno de 1930, ofereceu uma outra solução para o problema da caracterização de certa forma semelhante aos resultados de Herbrand.

Um segundo grupo importante de resultados refere-se aos chamados problemas de decisão para Lógica de Primeira Ordem. Por exemplo, o problema da validade consiste em determinar se existe ou não um algoritmo que recebe como entrada qualquer fórmula de primeira ordem e produz como saída SIM, se a fórmula é sempre válida, e NÃO em caso contrário. Church, em "An Unsolvable Problem of Elementary Number Theory" (1936), e independentemente Turing, no trabalho "On Computable Numbers, with an Application to the Entscheidungsproblem" (1936), provaram que tal algoritmo não pode existir, estabelecendo assim a indecidibilidade do problema da validade. Este resultado tem inúmeras ramificações imediatas para outros problemas de decisão em primeira ordem, que serão abordados na seção 2.7. No contexto deste livro, este resultado é evidentemente muito importante pois impõe uma limitação séria no que se pretende atingir com Programação em Lógica.

2.2 SINTAXE DAS LINGUAGENS DE PRIMEIRA ORDEM

Linguagens de primeira ordem permitem exprimir sentenças tais como

- (a) para todo x , $-1 \leq \text{sen}(x) \leq 1$
- (b) para todo x , $x \cup \emptyset = x$
- (c) para todo x , existe y tal que y é o gerente de x
- (d) para todo x, y e z , se x é ancestral de y e y é ancestral de z , então x é ancestral de z

Um alfabeto de primeira ordem contém: conectivos lógicos e parênteses; variáveis, como " x ", " y " e " z ", para denotar indivíduos arbitrários do domínio de discurso; constantes, como " \emptyset " ou " 1 ", para denotar indivíduos

específicos; e símbolos para denotar funções, como "sen" ou "U", ou relações, como " \leq ", "é gerente" ou "é ancestral". O alfabeto contém ainda dois símbolos especiais, chamados de quantificadores, que permitem abreviar, por exemplo, as frases "para todo..." e "existe...".

As regras de formação das linguagens de primeira ordem determinam que seqüências de símbolos formam termos e fórmulas. Termos são seqüências de símbolos, por exemplo, da forma "sen(x)" ou " $x \in \emptyset$ ", que denotarão indivíduos do domínio de discurso. As fórmulas mais simples são seqüências de símbolos, por exemplo, da forma "sen(x) ≤ 1 " ou " x é ancestral de z " que representam relacionamentos entre indivíduos e que receberão os valores-verdade F ou V. Fórmulas mais complexas são semelhantes às expressões apresentadas em (a), (b), (c) e (d) e também recebem os valores-verdade F ou V.

Definição 2.1:

Um *alfabeto de primeira ordem A* consiste de:

símbolos lógicos:

pontuação: (,)

conectivos: \neg , \wedge , \vee , \rightarrow , \equiv

quantificadores: \forall (quantificador universal - 'para todo...')
 \exists (quantificador existencial - 'existe...')

variáveis: um conjunto de símbolos distintos dos demais.

símbolo de igualdade (opcional): =

símbolos não-lógicos:

Um conjunto, possivelmente vazio, de *constantes* distintas dos demais símbolos.

Para cada $n > 0$, um conjunto, possivelmente vazio, de *símbolos funcionais n-ários* distintos dos demais símbolos.

Para cada $n > 0$, um conjunto, possivelmente vazio, de *símbolos predicativos n-ários* distintos dos demais símbolos.

Definição 2.2:

O conjunto dos *termos de primeira ordem* (ou simplesmente dos *termos*) sobre um alfabeto de primeira ordem *A* é o menor conjunto satisfazendo às seguintes condições:

- (i) toda variável em A é um termo sobre A ;
- (ii) toda constante em A é um termo sobre A ;
- (iii) se t_1, \dots, t_n são termos sobre A e f é um símbolo funcional n-ário de A , então $f(t_1, \dots, t_n)$ também é um termo sobre A .

Definição 2.3:

O conjunto das *fórmulas* sobre um alfabeto de primeira ordem A é o menor conjunto satisfazendo às seguintes condições:

- (i) se t_1, \dots, t_n são termos sobre A e p é um símbolo predicativo n-ário de A , então $p(t_1, \dots, t_n)$ é uma fórmula sobre A , chamada de *fórmula atômica*.
- (ii) se t_1 e t_2 são termos sobre A e " $=$ " é um símbolo de A , então $(t_1 = t_2)$ é uma fórmula sobre A , também chamada de *fórmula atômica*.
- (iii) se P e Q são fórmulas sobre A , então $(\neg P)$, $(P \wedge Q)$, $(P \vee Q)$, $(P \rightarrow Q)$ e $(P \equiv Q)$ também são fórmulas sobre A .
- (iv) se P é uma fórmula sobre A e x é uma variável de A , então $\forall x(P)$ e $\exists x(P)$ também são fórmulas sobre A .

Definição 2.4:

Uma fórmula Q é uma *sub-fórmula* de uma fórmula P se e somente se Q é uma subcadeia de P .

Definição 2.5:

A *linguagem de primeira ordem* sobre um alfabeto de primeira ordem A , denotada por $L(A)$, é o conjunto de termos e fórmulas de primeira ordem sobre A .

Todo alfabeto de primeira ordem contém necessariamente todos os símbolos lógicos, exceto o símbolo de igualdade, que é opcional. Portanto, a especificação de um alfabeto de primeira ordem A e, logo, de $L(A)$, fixará apenas o conjunto N dos símbolos não-lógicos e indicará a inclusão ou não do símbolo de igualdade. Identificaremos então, quando conveniente, A , $L(A)$ e N , como é usual nos textos de Lógica. Diremos ainda que os símbolos não-lógicos de A são símbolos não-lógicos de $L(A)$. Quando não for necessário especificar A , denotaremos $L(A)$ simplesmente por L .

Dentro do possível, seguiremos as seguintes convenções notacionais:

OBJETO SINTÁTICO	CONVENÇÃO NOTACIONAL
variáveis	letras do fim do alfabeto, freqüentemente subscritas (...x,y,z)
constants	letras do início do alfabeto (a,b,c,...)
símbolos funcionais	letras do início para o meio do alfabeto (...f,g,h,...)
símbolos predicativos	letras do meio para o fim do alfabeto (...p,q,r,...)
termos	letras perto do fim do alfabeto (...t,u,v,...)
fórmulas	letras maiúsculas do meio para o fim do alfabeto (...P,Q,R,...)
conjuntos de fórmulas	letras maiúsculas em negrito do meio para o fim do alfabeto (...P,Q,R,...)

Além disto, colocaremos entre aspas as ocorrências de símbolos não-lógicos, termos ou fórmulas dentro de um texto explicativo sempre que for necessário destacá-las.

Os termos e as fórmulas atômicas seguem a notação prefixa, ou seja, o símbolo funcional ou predutivo precede a lista de termos. Porém, convém adotar a notação infixa para certos símbolos funcionais ou predicativos binários, como "+" ou "<". Assim, em lugar de "+(a,b)" e de "<(x,y)", usaremos "a + b" e "x < y", respectivamente.

Como nas linguagens proposicionais, as fórmulas são, em princípio, completamente parentetizadas. Porém, seguiremos as mesmas convenções sobre omissão de parênteses já adotadas na seção 1.2.

Os dois exemplos abaixo ilustram a sintaxe das linguagens de primeira ordem:

Exemplo 2.1:

Considere o alfabeto de primeira ordem A contendo como símbolos não-lógicos a constante "1", dois símbolos funcionais unários, "-" e " sen ", e um símbolo predicativo binário, " \leq ".

Podemos então gradativamente transformar a sentença

$$\text{para todo } x, -1 \leq \text{sen}(x) \leq 1$$

em uma fórmula de $L(A)$ da seguinte forma:

1. $\forall x(-1 \leq \text{sen}(x) \leq 1)$
(introdução do quantificador)
2. $\forall x(-1 \leq \text{sen}(x) \wedge \text{sen}(x) \leq 1)$
(desmembramento da expressão $-1 \leq \text{sen}(x) \leq 1$ na conjunção de duas fórmulas atômicas)
3. $\forall x(\leq(-1,\text{sen}(x)) \wedge \leq(\text{sen}(x),1))$
(substituição da expressão $-1 \leq \text{sen}(x)$ pela fórmula atômica $\leq(-1,\text{sen}(x))$ na forma prefixa, e semelhantemente para a expressão $\text{sen}(x) \leq 1$)
4. $\forall x(\leq(-(1),\text{sen}(x)) \wedge \leq(\text{sen}(x),1))$
(substituição da expressão -1 pelo termo $-(1)$, pois $-$ é um símbolo funcional unário)

Exemplo 2.2: Linguagem da Teoria dos Conjuntos

O alfabeto A da linguagem da Teoria dos Conjuntos é extremamente sucinto, contendo apenas um símbolo não-lógico: o símbolo predicativo binário " \in ". Assumiremos por simplicidade que A possui ainda o símbolo de igualdade.

Nas fórmulas de $L(A)$ apresentadas em seguida, adotaremos a notação infixa tradicional ($t \in u$), com os parênteses externos omitidos opcionalmente. Usaremos ainda " \in " com o seu significado usual (pertinência), observando apenas que no domínio de discurso pretendido não há distinção entre conjuntos e elementos de conjuntos.

Seguem-se dois exemplos da formalização de sentenças em português. Para facilitar a compreensão, faremos a conversão gradativamente:

(a) 'Não há um conjunto que contenha todos os conjuntos'

$$1. \neg \exists x(\text{todo conjunto é membro de } x)$$

$$2. \neg \exists x(\forall y(y \in x))$$

(b) 'Para quaisquer dois conjuntos, há um conjunto cujos membros são os dois conjuntos dados'

$$1. \forall x(\forall y(\text{há um conjunto cujos membros são } x \text{ e } y))$$

$$2. \forall x(\forall y(\exists z(\text{os membros de } z \text{ são } x \text{ e } y)))$$

$$3. \forall x(\forall y(\exists z(\forall u(u \in z \equiv (u = x \vee u = y))))))$$

ou, omitindo os parênteses supérfluos:

$$4. \forall x \forall y \exists z \forall u(u \in z \equiv (u = x \vee u = y))$$

Pelas regras de formação das fórmulas, as variáveis podem ocorrer em uma fórmula associadas a um quantificador ou não. Esta observação origina uma série de definições auxiliares listadas a seguir.

Definição 2.6:

- (a) Em uma fórmula da forma $\forall x(Q)$ (ou da forma $\exists x(Q)$), Q é o *escopo* de $\forall x$ (ou de $\exists x$).
- (b) Uma ocorrência de uma variável x em uma fórmula P é *ligada* (*amarrada* ou *governada*) em P, se a ocorrência se dá em uma subfórmula de P da forma $\forall x(Q)$ ou da forma $\exists x(Q)$. Caso contrário, a ocorrência de x é *livre*.
- (c) Uma variável x é *livre* em P se existe uma ocorrência livre de x em P.
- (d) Uma fórmula P é uma *sentença* (ou uma fórmula *fechada*) se e somente se nenhuma variável ocorre livre em P.

Por exemplo, na fórmula $p(y) \vee \forall x(p(x))$ a ocorrência de y é livre e a ocorrência de x é ligada. Na fórmula $p(x) \vee \forall x(p(x))$ a primeira ocorrência de x é livre e a segunda ocorrência é ligada.

A lista de definições abaixo introduz certas classes especiais de fórmulas que serão referenciadas em capítulos posteriores.

Definição 2.7:

- (a) Dada uma fórmula P , com variáveis livres x_1, \dots, x_n , o *fecho universal* de P é a fórmula $\forall x_1 \dots \forall x_n (P)$ e o *fecho existencial* de P é a fórmula $\exists x_1 \dots \exists x_n (P)$.
- (b) Uma fórmula P está em *forma normal prenex* se e somente se P for da forma $Q(M)$ onde Q , o *prefixo* de P , é uma cadeia de quantificadores e M , a *matriz* de P , é uma fórmula sem ocorrências de quantificadores.
- (c) Uma fórmula P é uma *conjunção* se e somente se, omitindo-se os parênteses, for da forma $P_1 \wedge \dots \wedge P_n$.
- (d) Uma fórmula P é uma *disjunção* se e somente se, omitindo-se os parênteses, for da forma $P_1 \vee \dots \vee P_n$.
- (e) Uma fórmula P está em *forma normal conjuntiva* se e somente se estiver em forma normal prenex e a sua matriz for uma conjunção de disjunções de fórmulas atômicas, negadas ou não.

Por exemplo, a fórmula

$$\forall x \exists y (p(x,y) \rightarrow (q(x) \wedge q(y)))$$

está em forma normal prenex. O prefixo desta fórmula é $\forall x \exists y$ e a matriz é $p(x,y) \rightarrow (q(x) \wedge q(y))$. Já a fórmula

$$\forall x \exists y ((\neg p(x,y) \vee q(x)) \wedge (\neg p(x,y) \vee q(y)))$$

está em forma normal conjuntiva.

Por fim introduziremos a noção de substituição de variáveis por termos. Se u é um termo, $u[x/t]$ denotará o termo obtido substituindo-se cada ocorrência de x em u pelo termo t . Da mesma forma, se P é uma fórmula, $P[x/t]$ denotará a fórmula obtida substituindo-se cada ocorrência livre de x em P pelo termo t . Se x_1, \dots, x_n são variáveis distintas e t_1, \dots, t_n são termos, a expressão $u[x_1/t_1, \dots, x_n/t_n]$ denotará a substituição simultânea de x_i por t_i , $i = 1, \dots, n$, no termo u (e semelhantemente para fórmulas).

Uma variável x é *substitutível* por t em uma fórmula P se, para cada variável y ocorrendo em t , nenhuma subfórmula de P da forma $\forall y(Q)$ ou $\exists y(Q)$ contém uma ocorrência livre de x . Se isto ocorresse, y poderia se tornar inadvertidamente ligada em Q após a substituição. Por exemplo, se substituirmos x por $f(y)$ em $\forall y(p(x,y))$, obteremos $\forall y(p(f(y),y))$ com a primeira ocorrência de y , oriunda da substituição, tornando-se indevidamente ligada.

2.3 SEMÂNTICA DAS LINGUAGENS DE PRIMEIRA ORDEM

O significado das fórmulas de uma linguagem de primeira ordem depende da escolha de uma estrutura para o alfabeto da linguagem, isto é, de um mapeamento associando a cada símbolo não-lógico do alfabeto um elemento de um conjunto, uma função ou uma relação sobre este conjunto, dependendo do tipo do símbolo. Mais precisamente, temos (onde U^n denota o produto cartesiano de um conjunto U por ele mesmo n vezes):

Definição 2.8:

Seja A um alfabeto de primeira ordem e U um conjunto não vazio.

Uma *estrutura* para A sobre o universo U é uma função I satisfazendo às seguintes condições:

- (i) a cada constante c de A , I associa um elemento de U , denotado por $I(c)$.
- (ii) a cada símbolo funcional n -ário f de A , I associa uma função de U^n em U , denotada por $I(f)$.
- (iii) a cada símbolo predicativo n -ário p de A , I associa uma relação sobre U^n , denotada por $I(p)$.

Nota:

1. Alguns textos reservam o termo 'estrutura' apenas para a coleção de funções e relações e definem uma *interpretação* como um par (E, β) , onde E é uma estrutura neste novo sentido e β é um mapeamento entre os símbolos não-lógicos de A e as funções e relações de E tal que β segue as restrições da nossa definição.
2. O termo 'interpretação' também é frequentemente usado em lugar de 'estrutura'.

3. Como na maioria dos textos, por abuso de linguagem, diremos que uma estrutura para A é uma estrutura para $L(A)$, quando não for importante identificar o alfabeto A .

Voltando à Definição 2.8, se f for um símbolo funcional n-ário, $I(f)$ será uma função total cujo domínio é U^n e cujo contra-domínio é U . Não se consideram portanto funções parcialmente definidas dentro do contexto de estruturas de primeira ordem. Se p for um símbolo predicativo n-ário, $I(p)$ será então um subconjunto, possivelmente vazio, de U^n .

É extremamente importante observar que um alfabeto de primeira ordem pode admitir estruturas completamente diferentes. Por esta razão fomos cuidadosos em explicitar o significado pretendido, ou seja, a estrutura em questão, para os símbolos do alfabeto em todos os exemplos da seção 2.2. O exemplo a seguir enfatiza este ponto.

Exemplo 2.3:

Seja A um alfabeto contendo apenas um símbolo predicativo binário, p . Uma estrutura I para A é determinada então apenas por um conjunto não-vazio, o seu universo, e por uma relação binária $I(p)$ sobre o universo, o significado de p . Não existe portanto uma única estrutura para A .

Por exemplo, denote por N o conjunto dos naturais e por P^N o conjunto dos conjuntos de naturais. Seja " \leq " a relação de menor ou igual entre naturais e " \subseteq " a relação de subconjunto entre conjuntos de naturais. As estruturas I e J definidas abaixo, embora bastante diferentes, são ambas estruturas para A :

Estrutura I :

Universo : N

Valor de p : $I(p) = \{(m,n) \in N \times N / m \leq n\}$

Estrutura J :

Universo : P^N

Valor de p : $J(p) = \{(M,N) \in P^N \times P^N / M \subseteq N\}$

O fato do alfabeto A admitir estruturas bastante diferentes, em lugar de ser um inconveniente, facilita a formalização de propriedades genéricas de relações binárias.

Por exemplo, uma relação binária R é *transitiva* se e somente se para todo x, y e z , se (x,y) e (y,z) pertencem a R , então (x,z) também pertence a R . Assim, a seguinte fórmula da linguagem do alfabeto A formaliza transitividade:

$$\forall x \forall y \forall z (p(x,y) \wedge p(y,z) \rightarrow p(x,z))$$

Esta fórmula captura a definição de transitividade no sentido de que se ela for verdadeira em uma dada estrutura para A então a relação associada a p será transitiva. Por exemplo, as relações " \leq " e " \subseteq " são transitivas pois a fórmula acima é verdadeira em I e em J .

Como um segundo exemplo, uma relação binária R é *anti-simétrica* se e somente se para todo x e y , se (x,y) e (y,x) pertencem a R , então $x = y$. Assim, a fórmula abaixo captura esta propriedade:

$$\forall x \forall y (p(x,y) \wedge p(y,x) \rightarrow x = y)$$

Além da noção de estrutura, necessitaremos também da noção de uma função de avaliação para variáveis a fim de especificar a semântica das fórmulas de primeira ordem. Intuitivamente, uma função de avaliação fixará o significado das variáveis livres de uma fórmula.

Definição 2.9:

Seja A um alfabeto de primeira ordem e I uma estrutura para A com universo U . Uma *função de avaliação* (ou simplesmente uma *avaliação*) para A sobre I é qualquer função v que associa a cada variável de A um elemento de U .

Naturalmente, há várias avaliações diferentes para A sobre uma mesma estrutura I .

O significado, em I e v , dos termos e fórmulas sobre A é definido através de uma função $v[v, I]$ que associa a cada termo um elemento de U e a cada fórmula P o valor V , caso P seja verdadeira em I para v , e o valor F em caso contrário. Como sugere a notação, a função dependerá de I e de v . A definição formal da função $v[v, I]$ não será usada diretamente em nenhum ponto do texto, de modo que o leitor poderá passar diretamente à Definição 2.11 e confiar apenas no entendimento intuitivo dos termos e fórmulas.

Recorde que a linguagem de primeira ordem $L(A)$ é o conjunto dos termos e fórmulas sobre um alfabeto A .

Definição 2.10:

A função de avaliação para $L(A)$ induzida por uma estrutura I e uma função de avaliação v para o alfabeto A é a função

$$v[v, I]: L(A) \rightarrow U \cup \{F, V\}$$

definida indutivamente como:

AVALIAÇÃO DOS TERMOS

$$v[v, I](x) = v(x), \quad \text{para cada variável } x \text{ de } A$$

$$v[v, I](c) = I(c), \quad \text{para cada constante } c \text{ de } A$$

$$v[v, I](f(t_1, \dots, t_n)) = I(f)(v[v, I](t_1), \dots, v[v, I](t_n))$$

AVALIAÇÃO DAS FÓRMULAS

$$\begin{aligned} v[v, I](t_1 = t_2) &= V, & \text{se } v[v, I](t_1) &= v[v, I](t_2) \\ &= F, & \text{em caso contrário} \end{aligned}$$

$$\begin{aligned} v[v, I](p(t_1, \dots, t_n)) &= V, & \text{se } (d_1, \dots, d_n) \in I(p) \\ &= F, & \text{em caso contrário} \end{aligned}$$

onde $d_i = v[v, I](t_i)$, para $i = 1, \dots, n$

$$\begin{aligned} v[v, I](\neg P) &= V, & \text{se } v[v, I](P) &= F \\ &= F, & \text{se } v[v, I](P) &= V \end{aligned}$$

$$\begin{aligned} v[v, I](P \wedge Q) &= V, & \text{se } v[v, I](P) &= v[v, I](Q) = V \\ &= F, & \text{em caso contrário} \end{aligned}$$

$$\begin{aligned} v[v, I](P \vee Q) &= F, & \text{se } v[v, I](P) &= v[v, I](Q) = F \\ &= V, & \text{em caso contrário} \end{aligned}$$

$$\begin{aligned} v[v, I](P \rightarrow Q) &= F, & \text{se } v[v, I](P) &= V \text{ e } v[v, I](Q) = F \\ &= V, & \text{em caso contrário} \end{aligned}$$

$$\begin{aligned} v[v, I](P \equiv Q) &= V, & \text{se } v[v, I](P) &= v[v, I](Q) \\ &= F, & \text{em caso contrário} \end{aligned}$$

- $v[v, I](\forall x(P)) = V$, se $v[u, I](P) = V$
 para toda função de avaliação u para A
 diferindo de v apenas no valor de x
 $= F$, em caso contrário
- $v[v, I](\exists x(P)) = V$, se $v[u, I](P) = V$
 para alguma função de avaliação u para A
 diferindo de v apenas no valor de x
 $= F$, em caso contrário

Os exemplos abaixo ilustram os princípios desta definição, porém alguns comentários preliminares serão úteis. Inicialmente observe que usamos o sinal “=” para denotar três objetos diferentes: o símbolo de igualdade do alfabeto A , a relação de igualdade (por definição) da metalinguagem (i.e., da linguagem que estamos usando para estudar Lógica de Primeira Ordem) e a relação de igualdade entre elementos do conjunto U . Este uso múltiplo, desde que cuidadoso, é perfeitamente aceitável. Uma fórmula atômica da forma $p(t_1, \dots, t_n)$ será verdadeira quando $(v[v, I](t_1), \dots, v[v, I](t_n))$ pertencer a $I(p)$. O significado dos conectivos segue como em Lógica Sentencial. Uma fórmula da forma $\forall x(P)$ será verdadeira quando P for verdadeira para todas as formas possíveis de associar um elemento do universo de I à variável x , que é intuitivamente o significado do quantificador universal. Finalmente, uma fórmula da forma $\exists x(P)$ será verdadeira se pudermos encontrar algum elemento d do universo de I de tal forma que P se torne verdadeira quando x receber d como valor.

Exemplo 2.4:

Seja A um alfabeto de primeira ordem com um símbolo predicativo binário, p , e duas constantes, a e b . Seja I uma estrutura para este alfabeto sobre o universo $U = \{d_1, d_2, d_3, d_4\}$ tal que

$$\begin{aligned}
 I(a) &= d_1 \\
 I(b) &= d_2 \\
 I(p) &= \{(d_1, d_2), (d_3, d_2)\}
 \end{aligned}$$

Seja v uma função de avaliação qualquer para as variáveis de A . Mostraremos, em detalhe, que

$$v[v, I](p(a, b)) = V$$

Observemos inicialmente que pela Definição 2.10 temos:

$$1. \quad v[v, I](p(a, b)) = V \text{ se e somente se } (v[v, I](a), v[v, I](b)) \in I(p)$$

Por definição de I e pela Definição 2.10, temos que:

$$\begin{aligned} 2. \quad v[v, I](a) &= I(a) = d_1 \\ 3. \quad v[v, I](b) &= I(b) = d_2 \end{aligned}$$

Logo, por (2) e (3), (1) é equivalente a:

$$4. \quad v[v, I](p(a, b)) = V \text{ se e somente se } (d_1, d_2) \in I(p)$$

Mas, pela definição de $I(p)$, temos:

$$5. \quad (d_1, d_2) \in I(p)$$

Podemos então concluir, por (4) e (5), que

$$6. \quad v[v, I](p(a, b)) = V$$

como se queria demonstrar.

Mostraremos agora que

$$v[v, I](\exists x(p(a, x))) = V.$$

De fato, pela Definição 2.10, temos que

$$1. \quad v[v, I](\exists x(p(a, x))) = V \text{ se e somente se } v[u, I](p(a, x)) = V, \text{ para alguma função } u \text{ idêntica a } v, \text{ exceto em } x.$$

Tomemos então u como a seguinte função:

$$\begin{aligned} 2. \quad u(x) &= d_2 \\ 3. \quad u(y) &= v(y), \text{ para toda variável } y \text{ tal que } y \neq x \end{aligned}$$

Pela Definição 2.10 e pela definição de u e I , temos que:

$$\begin{aligned} 4. \quad v[u, I](p(a, x)) &= V \text{ se e somente se } (v[u, I](a), v[u, I](x)) \in I(p) \\ 5. \quad v[u, I](x) &= u(x) = d_2 \\ 6. \quad v[u, I](a) &= I(a) = d_1 \end{aligned}$$

$$(d_1, d_2) \in I(p)$$

Logo, por (4), (5), (6) e (7):

$$8. \quad v[u, I](p(a, x)) = V$$

Assim, por (1) e (8):

$$9. \quad v[v, I](\exists x(p(a, x))) = V$$

Terminaremos esta seção com a definição de implicação lógica e conceitos correlatos, observando que os conceitos definidos apenas para conjuntos de fórmulas estendem-se naturalmente a uma fórmula Q tomando-se o conjunto como sendo $\{Q\}$.

Definição 2.11:

Seja A um alfabeto de primeira ordem, Q uma fórmula de $L(A)$ e P um conjunto de fórmulas de $L(A)$.

- (a) Q é verdadeira em uma estrutura I de A , para uma avaliação v das variáveis de A , se e somente se $v[v, I](Q) = V$.
- (b) Q é falsa em I para v se e somente se $v[v, I](Q) = F$.
- (c) Q é válida em I se e somente se, para toda avaliação v das variáveis de A , Q é verdadeira em I para v .
- (d) Q é válida se e somente se, para toda estrutura I de A , Q é válida em I .
- (e) I satisfaz P para v se e somente se, para toda fórmula R em P , R é verdadeira em I para v .
- (f) I satisfaz P se e somente se existe uma avaliação v das variáveis de A tal que I satisfaz P para v .
- (g) P é satisfatível se e somente se existe uma estrutura que satisfaz P .
- (h) P é insatisfatível se e somente se P não é satisfatível.
- (i) Q é consequência lógica de P , ou P implica logicamente Q , se e somente se, para toda estrutura I de A , para toda avaliação v das variáveis de A , se I satisfaz P para v , então I também satisfaz Q para v .

Estes conceitos são em geral acompanhados de uma notação especial, como indicado abaixo:

$$Q \text{ é verdadeira em } I \text{ para } v \quad I \models Q[v]$$

$$Q \text{ é válida em } I \quad I \models Q$$

$$Q \text{ é válida} \quad \models Q$$

$$P \text{ implica logicamente } Q \quad P \models Q$$

Recorde que uma sentença é uma fórmula sem variáveis livres.

Definição 2.12:

Uma estrutura I de um alfabeto de primeira ordem A é um *modelo* para um conjunto de sentenças S sobre A se e somente se I satisfaz S .

Note que se P é satisfatível então R é satisfatível, para todo $R \in P$, mas o converso não é verdadeiro, por exemplo, tomando-se $P = \{p(x), \neg p(x)\}$. Note ainda que estas definições não são independentes. Em particular temos:

Proposição 2.1:

Seja A um alfabeto de primeira ordem, Q uma fórmula de $L(A)$ e P um conjunto de fórmulas de $L(A)$. Então:

- (a) $\models Q$ se e somente se $\neg Q$ é insatisfatível.
- (b) $P \models Q$ se e somente se $P \cup \{\neg Q\}$ é insatisfatível.

Demonstração

Como (a) segue de (b) tomando-se $P = \emptyset$, provaremos apenas (b).

(\rightarrow) Suponha que Q seja consequência lógica de P mas que $P \cup \{\neg Q\}$ seja satisfatível. Logo, existe uma estrutura I de A e uma avaliação v para as variáveis de A tais que todas as fórmulas em P e a fórmula $\neg Q$ são verdadeiras em I para v . Mas se $\neg Q$ é verdadeira em I para v , Q é falsa em I para v . Logo existe uma estrutura I e uma avaliação v tal que todas

as fórmulas em P são verdadeiras em I para v , mas Q é falsa. Contradição, pois supusemos que Q é consequência lógica de P .

(\leftarrow) Suponha que $P \cup \{\neg Q\}$ seja insatisfatível, mas que Q não seja consequência lógica de P . Logo, existe uma estrutura I de A e uma avaliação v para as variáveis de A tais que todas as fórmulas em P são verdadeiras em I para v , mas Q é falsa. Então $\neg Q$ tem que ser verdadeira em I para v . Logo, todas as fórmulas em $P \cup \{\neg Q\}$ são verdadeiras em I para v , ou seja, o conjunto é satisfatível. Contradição.

2.4 TEORIAS DE PRIMEIRA ORDEM

Esta seção apresenta o conceito de teoria de primeira ordem e discute a introdução de novos símbolos em uma teoria por definição, deixando para a seção seguinte os exemplos.

Dado um conjunto C , diremos que um subconjunto C' de C é *decidível* se e somente se existir um procedimento efetivo tal que, dado qualquer elemento c de C , o procedimento pára com SIM, se c pertencer a C' , e pára com NÃO, se c não pertencer a C' . Diremos ainda que um conjunto Q de sentenças é *fechado* sobre implicação lógica se e somente se, para toda sentença P , se P é consequência lógica de Q , então $P \in Q$.

Definição 2.13:

- (a) Uma *teoria de primeira ordem* (ou simplesmente uma *teoria*) é um par $T = (A, Q)$ onde A é um alfabeto de primeira ordem e Q é um conjunto de sentenças de $L(A)$ fechado sobre implicação lógica.
- (b) Uma teoria $T = (A, Q)$ é *axiomatizável* se e somente se existe um subconjunto decidível Q' de Q tal que $P \in Q$ se e somente se P é consequência lógica de Q' . Neste caso, denotaremos a teoria simplesmente pelo par $T = (A, Q')$ e diremos que as fórmulas em Q' são os *axiomas* de T .
- (c) Uma teoria T é *finitamente axiomatizável* se e somente se T for axiomatizável por um conjunto finito de sentenças.
- (d) Um *modelo* para uma teoria $T = (A, Q)$ é um modelo para Q .

Discutiremos brevemente no resto desta seção o problema de como introduzir, por definição, novos símbolos predicativos em uma teoria,

observando que a introdução de novos símbolos funcionais segue de forma semelhante.

Estaremos interessados em formalizar o mecanismo ilustrado pelo seguinte exemplo. Suponha que " $=$ " e " $>$ " sejam símbolos predicativos binários do alfabeto de uma teoria T e que desejemos utilizar o símbolo " \geq " com o seu significado usual. Podemos então tratar " \geq " como uma conveniência sintática no sentido de que $t \geq u$ abrevia a fórmula $t = u \vee t > u$. Porém, uma forma mais elegante consiste em incluir " \geq " no alfabeto como um legítimo símbolo predicativo binário e em adicionar ao conjunto de axiomas de T a fórmula

$$\forall x \forall y (x \geq y \equiv (x = y \vee x > y))$$

que é a definição (formal) de " \geq ".

Note que a definição de " \geq " tem um formato bem definido: ela é uma bi-condicional em que o lado esquerdo é uma fórmula atômica formada pelo novo símbolo e por variáveis e o lado direito é uma fórmula da linguagem original (isto é, em que o novo símbolo não ocorre) cujas variáveis livres são exatamente aquelas que ocorrem do lado esquerdo. Assim, sempre que precisarmos provar alguma propriedade envolvendo uma fórmula atômica da forma $t \geq u$, deveremos substituí-la por $t = u \vee t > u$.

Este tratamento pode ser generalizado da seguinte forma. Seja $T = (A, P)$ uma teoria. Seja p um símbolo predicativo n -ário e suponha que p não ocorre em A . Introduzir p por definição em T consiste em criar uma nova teoria $T' = (A', P')$ tal que A' é o alfabeto A acrescido do símbolo predicativo n -ário p e P' é o conjunto P acrescido de uma nova sentença, o *axioma de definição* de p , da forma

$$\forall x_1 \dots \forall x_n (p(x_1, \dots, x_n) \equiv Q)$$

onde Q é uma fórmula de $L(A)$ cujas únicas variáveis livres são x_1, \dots, x_n .

Observe que se I for um modelo de T então o axioma de definição de p determina univocamente uma estrutura I' para A' tal que:

I' possui o mesmo domínio que I

$I'(s) = I(s)$, para todo símbolo não-lógico s de A

$$I'(p) = \{(v(x_1), \dots, v(x_n)) / v \text{ é uma avaliação para } A \text{ e } v[v, I](Q) = V\}$$

A estrutura I' é chamada da *expansão* de I para o alfabeto de T' e é possível mostrar que I' é um modelo para T' .

2.5 EXEMPLOS DE TEORIAS

Esta seção exibe dois exemplos, apresentados sob forma de teorias, do uso de linguagens de primeira ordem em Ciência da Computação.

2.5.1 Um Dicionário de Programas e Arquivos

(A) DESCRIÇÃO DO PROBLEMA

Considere um dicionário simples contendo os programas e arquivos usados em um determinado sistema. Cada programa possui como atributo apenas a linguagem de programação em que foi escrito e cada arquivo apenas o tipo de organização física. O dicionário mantém ainda os arquivos usados e os programas chamados por cada programa.

A formalização do dicionário envolve duas questões distintas:

1. como descrever a organização lógica do dicionário;
2. como descrever o estado do dicionário em um particular instante;

O primeiro problema será resolvido definindo-se um alfabeto de primeira ordem para descrever a organização do dicionário. Quanto ao segundo, receberá dois tratamentos diferentes, ambos formalizados como teorias.

(B) O ALFABETO DO DICIONÁRIO

Para descrever a organização lógica do dicionário, utilizamos o alfabeto de primeira ordem (com igualdade), A , cujos símbolos não-lógicos são:

constants: todas as cadeias (finitas) de letras minúsculas do alfabeto da Língua Portuguesa.

símbolos predicativos binários: *programa*, *arquivo*, *chama*, *usa*

símbolo predutivo binário (opcional): *depende*

Usaremos estes símbolos com o seguinte significado pretendido:

constantes: possíveis nomes de programas, arquivos, linguagens de programação e métodos de organização de arquivos

programa(n,m) n é o nome de um programa escrito na linguagem m

arquivo(n,m) n é o nome de um arquivo cuja organização é m

chama(n,m) o programa n chama o programa m

usa(n,m) o programa n usa o arquivo m

depende(n,m) o programa n usa ou chama direta ou indiretamente m

(C) ESTADOS DO DICIONÁRIO COMO MODELOS DE UMA TEORIA

O alfabeto *A* descreve a organização lógica do dicionário, mas deixa em aberto que restrições os dados deverão satisfazer para refletir uma configuração possível da aplicação. Exemplos de restrições são: "dois programas não podem ter o mesmo nome" e "se n chama m, ambos devem ser programas cadastrados". Um estado do dicionário que satisfaz a todas as restrições é chamado de *consistente*.

Completaremos então a descrição do dicionário definindo uma teoria $\mathbf{D} = (\mathcal{A}, \mathcal{P})$, onde *A* é o alfabeto introduzida no ítem (B) e *P* é o seguinte conjunto de fórmulas de $L(A)$ descrevendo as restrições desejadas (o enunciado informal segue cada fórmula):

P1. $\forall x \forall y (\text{programa}(x,y) \rightarrow$

$(y = \text{fortran} \vee y = \text{cobol} \vee y = \text{pascal} \vee y = \text{prolog}))$

("as únicas linguagens de programação admitidas no momento são Fortran, Cobol, Pascal e Prolog")

P2. $\forall x \forall y \forall z (\text{programa}(x,y) \wedge \text{programa}(x,z) \rightarrow y = z)$

("todo programa é escrito em uma única linguagem")

A1. $\forall x \forall y (\text{arquivo}(x,y) \rightarrow (y = \text{sequencial} \vee y = \text{direto} \vee y = \text{indexado}))$

("as únicas organizações de arquivo admitidas no momento são sequencial, direta e indexada")

A2. $\forall x \forall y \forall z (\text{arquivo}(x,y) \wedge \text{arquivo}(x,z) \rightarrow y = z)$
 ("todo arquivo possui uma única organização física")

C1. $\forall x \forall y (\text{chama}(x,y) \rightarrow (\exists z (\text{programa}(x,z)) \wedge \exists w (\text{programa}(y,w))))$
 ("se x chama y então x e y são programas no dicionário")

U1. $\forall x \forall y (\text{usa}(x,y) \rightarrow (\exists z (\text{programa}(x,z)) \wedge \exists w (\text{arquivo}(y,w))))$
 ("se x usa y então x é um programa e y é um arquivo no dicionário")

D1. $\forall x \forall y (\text{chama}(x,y) \rightarrow \text{depende}(x,y))$
 ("se x chama y então x depende de y")

D2. $\forall x \forall y (\text{usa}(x,y) \rightarrow \text{depende}(x,y))$
 ("se x usa y então x depende de y")

D3. $\forall x \forall y \forall z (\text{depende}(x,z) \wedge \text{depende}(z,y) \rightarrow \text{depende}(x,y))$
 ("se x depende de z e z depende de y então x depende de y")

A teoria **D** é adequada no sentido de que os estados consistentes do dicionário podem ser interpretados como modelos de **D**. Por exemplo, um estado consistente do dicionário corresponderia ao seguinte modelo **I** de **D**:

Universo: conjunto das cadeias de letras maiúsculas do alfabeto da Língua Portuguesa.

Significado das Constantes: as constantes denotarão elas próprias, apenas em maiúsculo. Por exemplo, a constante *fortran* denotará o elemento do universo *FORTRAN*.

Significado dos Símbolos Predicativos Binários: fixado através das seguintes relações binárias finitas sobre o universo:

$$\begin{aligned} I(\text{programa}) &= \{(A,\text{FORTRAN}), (B,\text{PASCAL}), (C,\text{FORTRAN})\} \\ I(\text{arquivo}) &= \{(D,\text{SEQUENCIAL}), (E,\text{DIRETO})\} \\ I(\text{chama}) &= \{(A,B), (A,C)\} \\ I(\text{usa}) &= \{(A,D), (B,E)\} \\ I(\text{depende}) &= \{(A,B), (A,C), (A,D), (B,E), (A,E)\} \end{aligned}$$

Note que, pela forma como estas relações são definidas, **I** satisfaz a todos os axiomas de **D**.

Os axiomas de **D**, no entanto, não capturam algumas das restrições informalmente expostas na descrição de **I**, nem mesmo aquela relativa ao significado de *depende*. Considere, por exemplo, a estrutura **J** idêntica a **I**, exceto que:

$$J(\text{depende}) = \{(A,B), (A,C), (A,D), (B,E), (A,E), (A,A)\}$$

A estrutura **J**, como a estrutura **I**, satisfaz a todos os axiomas da teoria **D**. Em particular, note que **J** satisfaz os axiomas D1, D2 e D3. Mas, **J(depende)** contém o par (A,A) que não deveria existir pelo significado pretendido para *depende*. Portanto, os axiomas D1, D2 e D3 não evitam a inclusão de (A,A) , ou de outros pares espúrios semelhantes.

(D) EXPANSÃO DO DICIONÁRIO

Para criar uma tabela dos programas em *FORTRAN*, podemos introduzir *prog_fort* por definição em **D** como um símbolo predicativo unário cujo axioma de definição é:

$$\forall x (\text{prog_fort}(x) \equiv \text{programa}(x, \text{fortran}))$$

Seja **D'** a nova teoria assim obtida. O modelo **I'** para **D'**, resultante da expansão do modelo **I** descrito em (C), corresponderia a **I** acrescido de:

$$I'(\text{prog_fort}) = \{A, C\}$$

(E) ESTADOS DO DICIONÁRIO COMO TEORIAS

Uma abordagem alternativa para formalizar o dicionário seria descrever cada estado consistente por uma teoria cujos axiomas representam

- os dados armazenados no dicionário (através de fórmulas atômicas da linguagem);
- as propriedades desejadas de *depende*.

Esta forma é, portanto, puramente sintática.

Assim, por exemplo, o estado descrito pela estrutura **I** do item (C) corresponderia à teoria **E=(A,Q)**, onde **Q** é o conjunto de fórmulas atômicas a seguir:

1. *programa(a,fortran)*
2. *programa(b,pascal)*
3. *programa(c,fortran)*
4. *arquivo(d,sequencial)*
5. *arquivo(e,direto)*
6. *chama(a,b)*
7. *chama(a,c)*
8. *usa(a,d)*
9. *usa(b,e)*

acrescido das propriedades de *depende*:

10. $\forall x \forall y (chama(x,y) \rightarrow depende(x,y))$
11. $\forall x \forall y (usa(x,y) \rightarrow depende(x,y))$
12. $\forall x \forall y \forall z (depende(x,z) \wedge depende(z,y) \rightarrow depende(x,y))$

Note que, pela bijeção entre o universo da estrutura *I* do item (C) e o conjunto de constantes do alfabeto *A* do item (B), é realmente possível estabelecer uma bijeção trivial entre as relações da estrutura *I*, exceto *I(depende)*, e as fórmulas em (1) a (9). Este ponto será explorado novamente nas seções 3.4 e 3.5 onde identificaremos diretamente o universo das estruturas com o conjunto de termos da linguagem, evitando a distinção artificial via maiúsculas e minúsculas introduzida aqui.

Conforme veremos, a teoria **E** corresponde a um programa em lógica, onde as fórmulas em (1) a (9) descrevem os fatos de interesse e as em (10), (11) e (12) são usadas para computar o relacionamento associado a *depende* quando necessário.

2.5.2 Uma Teoria para Listas

(A) DESCRIÇÃO DO PROBLEMA

Dado um conjunto não-vazio *A*, cujos elementos chamaremos de *átomos*, e um dado elemento não pertencente a *A*, denotado por *nil*, o conjunto *L(A)* das *listas* sobre *A* é o menor conjunto satisfazendo às seguintes propriedades:

- (a) *nil* pertence a *L(A)*;

- (b) se t pertence a A ou a $L(A)$, e u pertence a $L(A)$, então o par ordenado (t,u) pertence a $L(A)$.

Note que, por definição, toda lista termina em *nil* e toda lista está completamente parentetizada. Por exemplo, se $A = \{a_1, a_2, a_3, \dots\}$, as expressões abaixo são então listas sobre A :

1. *nil*
2. (a_1, \textit{nil})
3. $(a_4, ((a_1, \textit{nil}), \textit{nil}))$
4. $(\textit{nil}, ((a_4, ((a_1, \textit{nil}), \textit{nil})), \textit{nil}))$

Convencionando-se que o agrupamento dos elementos se dá pela direita, podemos diminuir então o número de parênteses de tal forma que as duas últimas listas acima, por exemplo, se reduzam a:

5. $(a_4, (a_1, \textit{nil}), \textit{nil})$
6. $(\textit{nil}, (a_4, (a_1, \textit{nil})), \textit{nil}), \textit{nil})$

Desejamos então formalizar o tratamento de listas e as suas principais operações, que são: decomposição de uma lista (x,y) na sua *cabeça* x e na sua *cauda* y ; teste de pertinência para uma lista; e *concatenação* de listas.

(B) UMA TEORIA INICIAL PARA LISTAS

Introduziremos neste item uma teoria, \mathbf{L} , para descrever listas sobre um conjunto arbitrário de átomos. O alfabeto de \mathbf{L} contém os seguintes símbolos não-lógicos:

constantes: *nil*, c_1, c_2, c_3, \dots

símbolo funcional binário: $\cdot\cdot$

símbolos predicativos unários: *lista*, *átomo*

Adotaremos a notação infixa para o símbolo funcional binário $\cdot\cdot$, de tal forma que o termo $\cdot\cdot(t,u)$ corresponderá à expressão $(t.u)$.

O significado pretendido dos símbolos não-lógicos será o seguinte:

constantes: nil denota a lista vazia e c_1, c_2, c_3, \dots denotam átomos

(t,u) denota a lista (t,u)

lista(t) será verdadeiro se t for um termo que denota uma lista; caso contrário será falso

átomo(t) será verdadeiro se t denotar um átomo; caso contrário será falso

O conjunto de axiomas de \mathcal{L} consiste das seguintes fórmulas descrevendo as propriedades de *átomo* e *lista* (o enunciado informal segue cada fórmula):

A0. $\text{átomo}(c_i)$, para cada constante c_i
("cada constante c_i denota um átomo")

L0. *lista(nil)*
("nil é uma lista")

L1. $\forall x \forall y ((\text{átomo}(x) \vee \text{lista}(x)) \wedge \text{lista}(y) \equiv \text{lista}((x,y)))$
("se x é um átomo ou uma lista, e y é uma lista, então o par ordenado (x,y) é uma lista", e vice-versa).

AL. $\neg \exists x (\text{átomo}(x) \wedge \text{lista}(x))$
("nenhum objeto é ao mesmo tempo um átomo e uma lista")

O resto da discussão justifica a escolha desta teoria.

Inicialmente observe que há uma correspondência imediata entre listas e um conjunto de termos da linguagem. Por exemplo, se o conjunto de átomos for $A = \{a_1, a_2, a_3, \dots\}$ e a constante c_i denotar o átomo a_i , os termos abaixo denotam as listas apresentadas em (1) a (4) do ítem (A):

7. nil
8. (c_1 .nil),
9. (c_4 .((c_1 .nil).nil))
10. (nil.((c_4 .((c_1 .nil).nil)).nil))

A semelhança entre os termos em (7) a (10) e as expressões em (1) a (4) é proposital e resulta da escolha das constantes do alfabeto e da adoção da

notação infixa para o símbolo funcional “.” (nada mais fizemos do que trocar vírgulas por pontos!). Porém, deve ficar bem clara a diferença: as expressões em (1) a (4) denotam listas na nossa definição informal, enquanto que as expressões em (7) a (10) são termos da nossa linguagem formal.

Convencionando-se novamente que o agrupamento dos elementos se dá pela direita, os termos em (9) e (10) podem ser simplificados para:

11. $(c_4.(c_1.nil).nil)$
12. $(nil.(c_4.(c_1.nil).nil).nil)$

Porém nem todo termo corresponde a uma lista. Por exemplo o termo $"(c_1.c_2)"$ não corresponde a nenhuma lista pois não termina por “nil” e c_2 não é uma lista.

Este problema pode ser resolvido pelo menos de duas maneiras. Poderíamos redefinir listas de tal forma que não precisassem terminar por nil, o que cria dois inconvenientes:

1. impossibilita a representação de listas com apenas um elemento através de um termo construído com o símbolo funcional “.”, pois este exige dois argumentos;
2. dificulta a definição recursiva das operações sobre listas (ver item (C)), pois não haveria uma identificação única, a ocorrência de nil, para o término de uma lista.

Adotamos então uma segunda solução, baseada na introdução dos símbolos predicativos *lista* e *átomo* para separar os termos que denotam listas dos que não denotam, e na definição adequada das suas propriedades.

(C) OPERAÇÕES SOBRE LISTAS

Estenderemos a teoria **L** para descrever parte das operações usuais sobre listas da seguinte forma. Acrescentaremos inicialmente ao seu alfabeto os seguintes símbolos não-lógicos:

símbolos predicativos binários: *cabeça*, *cauda*, *membro*

símbolo predutivo ternário: *concat*

O significado pretendido para estes símbolos é o seguinte:

cabeça(t,u) será verdadeiro se *u* for uma lista da forma (*t.v*), para algum *v*; caso contrário será falso

cauda(t,u) será verdadeiro se *u* for uma lista da forma (*v.t*), para algum *v*; caso contrário será falso

membro(t,u) será verdadeiro se *t* for um membro da lista *u*; caso contrário será falso

concat(t,u,v) será verdadeiro se *v* for a concatenação da lista *t* com a lista *u*; caso contrário será falso

Exceto por *membro*, é mais usual considerar todos os outros símbolos como denotando funções, em lugar de relações, como fizemos aqui. Note porém que o nosso enfoque é mais geral, pois *cabeça*, por exemplo, representa tanto a função que dada uma lista retorna o seu primeiro elemento, quanto o mapeamento inverso. A mesma observação vale para *cauda* e *concat*.

Formalizaremos o significado pretendido destas operações acrescentando os seguintes axiomas à teoria **L**:

CA1. $\forall x(\neg \text{cabeça}(x,\text{nil}))$
("nil não tem cabeça")

CA2. $\forall x \forall y (\text{lista}((x.y)) \rightarrow \text{cabeça}(x,(x.y)))$
("a cabeça de (x.y) é x")

CD1. $\text{cauda}(\text{nil},\text{nil})$
("a cauda de nil é nil")

CD2. $\forall x \forall y (\text{lista}((x,y)) \rightarrow \text{cauda}(y,(x,y)))$
 ("a cauda de (x,y) é y")

ME1. $\forall x (\neg \text{membro}(x, \text{nil}))$
 ("nil não tem membros")

ME2. $\forall x \forall y (\text{lista}((x,y)) \rightarrow \text{membro}(x, (x,y)))$
 ("a cabeça de uma lista é um membro da lista")

ME3. $\forall x \forall y \forall z (\text{lista}((x,y)) \rightarrow (\text{membro}(z,y) \rightarrow \text{membro}(z, (x,y))))$
 ("se z é membro da cauda de uma lista, então z é membro da lista")

CO1. $\forall x (\text{lista}(x) \rightarrow \text{concat}(\text{nil}, x, x))$
 ("a concatenação da lista vazia a x é x")

CO2. $\forall x \forall y \forall z \forall u (\text{lista}((u,x)) \wedge \text{lista}(y) \wedge \text{lista}((u,z)) \rightarrow$
 $(\text{concat}(x,y,z) \rightarrow \text{concat}((u,x), y, (u,z))))$
 ("se a lista x concatenada com a lista y resulta na lista z, então a lista
 (u,x) concatenada com a lista y resulta na lista (u,z)")

2.6 UM SISTEMA AXIOMÁTICO

Esta seção apresenta, de forma breve, um "cálculo" permitindo verificar se uma fórmula de primeira ordem é consequência lógica de um conjunto de fórmulas. O "cálculo" assume a forma de um sistema axiomático composto de um conjunto (infinito) de fórmulas, chamados de *axiomas lógicos*, e de uma *regra de inferência*, chamada de Modus Ponens (MP), permitindo gerar uma nova fórmula a partir de duas outras.

Diremos que P é uma *generalização* de Q se e somente se P for da forma $\forall x_1 \dots \forall x_n (Q)$, para $n \geq 0$ e variáveis x_1, \dots, x_n . Diremos ainda que uma fórmula de primeira ordem é uma *tautologia* se ela puder ser mapeada em uma tautologia da Lógica Sentencial. Assim, por exemplo, a fórmula de primeira ordem $(\forall x(P) \vee \neg \forall x(P))$ é uma tautologia pois ela pode ser mapeada na tautologia da Lógica Sentencial $A \vee \neg A$, associando-se o símbolo proposicional A à fórmula $\forall x(P)$.

O sistema axiomático adotado, AX , consiste de:

Axiomas: todas as generalizações de fórmulas da forma:

Grupo 0: tautologias

Grupo 1:

$$\neg \exists x(P) \rightarrow \forall x(\neg P)$$

$$\exists x(P) \rightarrow \neg \forall x(\neg P)$$

Grupo 2: $\forall x_1 \dots \forall x_n(P) \rightarrow P[x_1/t_1, \dots, x_n/t_n]$
se x_i for substitutível por t_i em P , para $i = 1, \dots, n$

Grupo 3: $\forall x(P \rightarrow Q) \rightarrow (\forall x(P) \rightarrow \forall x(Q))$

Grupo 4: $P \rightarrow \forall x(P)$, se x não ocorre livre em P

e, caso o alfabeto inclua a igualdade,

Grupo 5: $x = x$

Grupo 6: $x = y \rightarrow (P \rightarrow P')$
se P for uma fórmula atômica e P' for obtida de P substituindo-se zero ou mais ocorrências de x por y

Regra de Inferência (Modus Ponens):

MP. a partir de P e de $(P \rightarrow Q)$, deduza Q

Os axiomas do grupo 0 transferem toda a Lógica Sentencial para o sistema AX . Em particular, eles permitem transformar todos os conectivos em " \neg " e " \rightarrow ", já que as seguintes fórmulas da Lógica Sentencial são tautologias:

$$(A \vee B) \equiv (\neg A \rightarrow B)$$

$$(A \wedge B) \equiv \neg(A \rightarrow \neg B)$$

$$(A \equiv B) \equiv (A \rightarrow B) \wedge (B \rightarrow A)$$

Os axiomas do grupo 1 permitem por sua vez eliminar o quantificador existencial traduzindo-o para o universal. Os axiomas dos grupos 2, 3 e 4

capturam propriedades do quantificador universal, e os axiomas dos grupos 5 e 6, as propriedades da igualdade.

A partir da noção de sistema axiomático podemos então formalizar as noções de teorema e de dedução:

Definição 2.14:

Uma fórmula Q é um *teorema* de um conjunto de fórmulas P em AX (denotado por $P \vdash Q$) se e somente se existir uma seqüência (Q_0, \dots, Q_n) de fórmulas de L tal que $Q_n = Q$ e, para todo $i \leq n$, um dos casos abaixo é satisfeito:

- (i) Q_i é um axioma de AX ou Q_i ocorre em P ;
- (ii) existem $j, k < i$ tais que Q_i pode ser obtida por Modus Ponens de Q_j e Q_k .

A seqüência (Q_0, \dots, Q_n) é uma *dedução* em AX de Q a partir de P .

A tarefa de estabelecer que Q é um teorema de P consiste exatamente em exibir uma dedução, isto é, uma seqüência $s = (Q_0, \dots, Q_n)$ satisfazendo às condições acima. Note que, uma vez exibida a seqüência s , a tarefa de testar se s satisfaz à definição de dedução é puramente mecânica. Porém, encontrar uma dedução é em geral bastante difícil pois o propósito do sistema axiomático AX não é levar a procedimentos para dedução automática de teoremas, mas sim servir de base para investigações sobre a natureza do conceito fundamental de implicação lógica, conforme veremos no final desta seção e na seção seguinte.

A escolha dos axiomas lógicos e da regra de inferência do sistema AX não é única. Porém, para esta escolha é possível provar os seguintes importantes resultados:

Teorema 2.1: (Correção)

Para toda linguagem de primeira ordem L , para todo conjunto de fórmulas P de L , para toda fórmula Q de L , se $P \vdash Q$ então $P \models Q$

Teorema 2.2: (Completude)

Para toda linguagem de primeira ordem L , para todo conjunto de fórmulas P de L , para toda fórmula Q de L , se $P \models Q$ então $P \vdash Q$

Teorema 2.3: (Compacidade)

Um conjunto P de fórmulas é satisfatível se e somente se todo subconjunto finito de P for satisfatível.

O Teorema da Correção nos diz que se Q é um teorema de P então Q é uma consequência lógica de P e o Teorema da Completude nos diz que que se Q é uma consequência lógica de P então Q é um teorema de P . Os dois teoremas em conjunto nos levam a concluir que o sistema axiomático AX oferece uma caracterização sintática exata da noção semântica de implicação lógica. Este resultado era um dos objetivos fundamentais do estudo da metateoria de Lógica de Primeira Ordem.

O Teorema da Compacidade é uma das características fundamentais de Lógica de Primeira Ordem. Ele segue do Teorema da Completude e da definição de satisfatibilidade e será usado para provar o Teorema de Herbrand no capítulo 3.

Para comparações futuras, apresentaremos a seguir uma dedução no sistema AX .

Exemplo 2.5:

Seja P a formalização do conteúdo do dicionário em um dado instante apresentada no exemplo da seção 2.5.1(E) (para facilitar a dedução, a fórmula em (D3) do exemplo original foi substituída pela sua equivalente em (12) abaixo):

1. *programa(a,fortran)*
2. *programa(b,pascal)*
3. *programa(c,fortran)*
4. *arquivo(d,sequencial)*
5. *arquivo(e,direto)*
6. *chama(a,b)*
7. *chama(a,c)*
8. *usa(a,d)*
9. *usa(b,e)*
10. $\forall x \forall y (chama(x,y) \rightarrow depende(x,y))$
11. $\forall x \forall y (usa(x,y) \rightarrow depende(x,y))$
12. $\forall x \forall y \forall z (depende(x,z) \rightarrow (depende(z,y) \rightarrow depende(x,y)))$

Seja Q a seguinte fórmula:

depende(a,e)

Provaremos, usando o sistema AX, que Q é um teorema de P.

1. *chama(a,b)* . em P
2. *usa(b,e)* . em P
3. $\forall x \forall y (chama(x,y) \rightarrow depende(x,y))$. em P
4. $\forall x \forall y (usa(x,y) \rightarrow depende(x,y))$. em P
5. $\forall x \forall y \forall z (depende(x,z) \rightarrow (depende(z,y) \rightarrow depende(x,y)))$. em P
6. $(\forall x \forall y (chama(x,y) \rightarrow depende(x,y))) \rightarrow (chama(a,b) \rightarrow depende(a,b))$. grupo 2
7. $(chama(a,b) \rightarrow depende(a,b))$. 3, 6 por MP
8. *depende(a,b)* . 1, 7 por MP
9. $(\forall x \forall y (usa(x,y) \rightarrow depende(x,y))) \rightarrow (usa(b,e) \rightarrow depende(b,e))$. grupo 2
10. $(usa(b,e) \rightarrow depende(b,e))$. 4, 9 por MP
11. *depende(b,e)* . 2, 10 por MP
12. $\forall x \forall y \forall z (depende(x,z) \rightarrow (depende(z,y) \rightarrow depende(x,y))) \rightarrow (depende(a,b) \rightarrow (depende(b,e) \rightarrow depende(a,e)))$. grupo 2
13. $(depende(a,b) \rightarrow (depende(b,e) \rightarrow depende(a,e)))$. 5, 12 por MP
14. $(depende(b,e) \rightarrow depende(a,e))$. 8, 13 por MP
15. *depende(a,e)* . 11, 14 por MP

2.7 PROBLEMAS DE DECISÃO

Esta seção aborda os chamados problemas de decisão para Lógica de Primeira Ordem e enuncia o Teorema de Church, um resultado limitativo extremamente importante para a mecanização da Lógica de Primeira Ordem. A discussão será informal, evitando uma definição precisa do conceito de procedimento efetivo, entendido como sinônimo de algoritmo.

No contexto desta seção, um *problema de decisão*, ou simplesmente um *problema*, consiste de um par $PR = (C, P)$, onde C é um conjunto, cujos elementos são chamados de *instâncias* de PR, e P é um subconjunto de C , cujos elementos são chamados de *instâncias solúveis* de PR.

Um problema PR será *decidível* se existir um *procedimento de decisão* para PR, ou seja, um procedimento efetivo que receba como entrada qualquer instância p de PR e pare com SIM, se p for uma instância solúvel, e pare com NÃO, em caso contrário; será *indecidível* se não existir tal procedimento; e será *parcialmente decidível* se existir um *procedimento de decisão parcial* para PR, ou seja, um procedimento efetivo que receba como entrada qualquer instância p de PR e pare com SIM, se p for uma instância solúvel, e não pare ou pare com NÃO, em caso contrário. Portanto, todo problema decidível é parcialmente decidível. Além disto, um problema pode ser ao mesmo tempo indecidível e parcialmente decidível.

A discussão a seguir se desenvolve em torno dos seguintes problemas de decisão para Lógica de Primeira Ordem

Problema da Implicação Lógica:

conjunto de instâncias: pares (P, Q) tais que P é um conjunto de fórmulas e Q é uma fórmula de uma linguagem de primeira ordem arbitrária

conjunto das instâncias solúveis: instâncias (P, Q) tais que P implica logicamente Q

Problema da Insatisfatibilidade:

conjunto de instâncias: conjuntos P tais que P é um conjunto de fórmulas de uma linguagem de primeira ordem arbitrária

conjunto das instâncias solúveis: instâncias P tais que P é insatisfatível

Problema da Validade:

conjunto de instâncias: fórmulas de primeira ordem

conjunto das instâncias solúveis: instâncias Q tais que Q é uma fórmula válida

Em todos os enunciados em que aparece um conjunto P de fórmulas, supõem-se que P seja finito ou, de alguma forma, "efetivamente apresentado" (embora a formalização deste ponto seja importante, não nos deteremos nela).

Através dos resultados enunciados na seção 2.6 podemos imediatamente estabelecer o seguinte resultado:

Resultado 2.1:

O Problema da Implicação Lógica para linguagens de primeira ordem é parcialmente decidível.

(Esboço da Demonstração)

Considere o seguinte procedimento para estabelecer se Q é um teorema de P em AX :

- enumere de forma apropriada todos as provas em AX a partir de P ;
- pare com SIM, se uma prova para Q ocorrer na enumeração;
- pare com NÃO, se uma prova para $\neg Q$ ocorrer na enumeração;

Tal procedimento decide então parcialmente se Q é um teorema de P em AX , pois ele sempre pára com SIM, se Q for um teorema de P (para isto a enumeração tem que ser "apropriada"), mas pode não parar em caso contrário, pois tanto Q quanto $\neg Q$ poderão não ser teoremas de P em AX .

Mas, pelos Teoremas da Correção e da Completude para AX , sabemos que Q é um teorema de P em AX se e somente se P implica logicamente Q . Logo, o procedimento acima é um procedimento de decisão parcial para o problema da implicação lógica.

Este argumento intuitivo tem interesse apenas para o estudo da metateoria de Lógica de Primeira Ordem pois seria extremamente ineficiente tentar testar implicação lógica em primeira ordem adotando o procedimento sugerido, já que a enumeração de todos os teoremas de P usando AX é inaceitável do ponto de vista prático.

Porém, um problema mais fundamental aparece neste ponto: O problema da implicação lógica para linguagens de primeira ordem é decidível? Ou seja, será possível substituir o procedimento acima por outro semelhante mas que sempre pare? A resposta a esta questão é NÃO, mesmo para o caso mais simples do problema da validade.

Resultado 2.2: (Teorema de Church)

O problema da validade para linguagens de primeira ordem é indecidível.

Note que este é um resultado importante para o desenvolvimento de Programação em Lógica pois impõe um limite sobre o que podemos esperar da mecanização dos princípios lógicos para primeira ordem.

A partir destes dois resultados iniciais podemos obter outros explorando um pouco mais a noção de problema.

Dois problemas PR' e PR'' são *complementares* (ou *co-problemas*) se e somente se PR' e PR'' tem o mesmo conjunto de instâncias e os conjuntos de instâncias solúveis de PR' e PR'' são conjuntos complementares. Assim, segue-se que:

Resultado 2.3:

Sejam PR' e PR'' problemas complementares.

- (a) PR' é decidível se e somente se PR' e PR'' são parcialmente decidíveis.
- (b) Se PR' é indecidível mas parcialmente decidível, então PR'' é indecidível mas não é parcialmente decidível.

Demonstração

O item (b) é consequência de (a). Para estabelecer (a), observe que se PR' é decidível então trivialmente PR'' também é decidível. Conversamente, suponha que existam procedimentos de decisão parcial P' e P'' para PR' e PR'' . Construa um procedimento P tal que, dada uma instância p' de PR' , P execute concorrentemente (e de forma justa) P' e P'' com entrada p' , pare com SIM se P' parar com SIM e pare com NÃO se P'' parar com SIM. Assim, P sempre pára com SIM, se p' for uma instância solúvel de PR' , e sempre pára com NÃO, se p' não for uma instância solúvel de PR' (pois PR'' é o co-problema de PR' e P'' é um procedimento de decisão parcial para PR''). Portanto, P é um procedimento de decisão para PR' , ou seja, PR' é decidível.

Um problema PR' é *redutível* a um outro problema PR'' se e somente se existir um *procedimento de redução* de PR' para PR'' , isto é, um procedimento efetivo para construir, para toda instância p' de PR' , uma

instância p'' de PR'' tal que p' é uma instância solúvel de PR' se e somente se p'' for uma instância solúvel de PR'' . A partir desta definição temos imediatamente os seguintes resultados:

Resultado 2.4:

- (a) Se PR' for redutível a PR'' e se PR'' for (parcialmente) decidível então PR' também será (parcialmente) decidível.
- (b) Se PR' for redutível a PR'' e se PR' for indecidível então PR'' também será indecidível.

Demonstração

Para estabelecer (a), observe que a composição do procedimento de redução de PR' para PR'' com o procedimento de decisão (parcial) para PR'' será um procedimento de decisão (parcial) para PR' . Para obter (b), suponha que PR' seja indecidível mas que PR'' seja decidível. Por (a), PR' será então decidível. Contradição.

Utilizando estes resultados auxiliares, podemos então estabelecer que (observando que o problema da satisfatibilidade é o co-problema do problema da insatisfatibilidade):

Resultado 2.5:

- (a) O problema da validade é parcialmente decidível.
- (b) O problema da implicação lógica é indecidível.
- (c) O problema da insatisfatibilidade é indecidível, mas parcialmente decidível.
- (d) O problema da satisfatibilidade é indecidível e não é parcialmente decidível.

Demonstração

Como P é uma fórmula válida se e somente se P é consequência lógica do conjunto vazio, o problema da validade se reduz ao problema da implicação lógica. Portanto, (a) e (b) seguem dos resultados 2.1, 2.2 e 2.4.

Como P é consequência lógica de P se e somente se $P \cup \{\neg P\}$ é insatisfatível, o problema da implicação lógica se reduz ao problema da insatisfatibilidade e vice-versa. Logo, (c) segue de (b) e dos resultados 2.1 e 2.4.

Por fim, como o problema da satisfatibilidade é o co-problema do problema da insatisfatibilidade, (d) segue de (c) e do resultado 2.3.

NOTAS BIBLIOGRÁFICAS

Este capítulo baseia-se principalmente nos textos de Enderton [1972, Cap. 2] e Shoenfield [1967], a menos dos exemplos. Em particular, Enderton [1972] oferece uma formalização interessante sobre técnicas de prova usuais em Matemática em termos do sistema axiomático AX . As questões de indecidibilidade para Lógica de Primeira Ordem são discutidas em detalhe em Shoenfield [1967, Cap. 6], Bell e Machover [1977, Cap. 7], Machtey e Young [1978, Cap. 4] e Brainerd e Landweber [1974, Cap. 7], por exemplo. Uma introdução rápida a estes problemas pode ser encontrada em Manna [1974]. Seguimos Zimbarg [1973] na tradução dos termos técnicos.

CAPÍTULO 3: NOTAÇÃO CLAUSAL E O TEOREMA DE HERBRAND

Este capítulo divide-se em duas partes. As duas primeiras seções introduzem a notação clausal para fórmulas enquanto que três últimas seções apresentam duas provas para o Teorema de Herbrand e discutem procedimentos de refutação baseados diretamente neste resultado.

As seções recomendadas para cada nível de leitura são:

nível introdutório: 3.2

nível intermediário: 3.2, 3.3

3.1 INTRODUÇÃO

Este capítulo introduz a notação clausal para fórmulas, apresenta um algoritmo para obter a representação clausal de uma fórmula e prova o Teorema de Herbrand, pontos fundamentais para a mecanização da Lógica a ser apresentada nos próximos capítulos. Apenas linguagens de primeira ordem sem o símbolo de igualdade serão adotadas.

Os principais resultados deste capítulo apareceram originalmente no capítulo 5 da tese de doutorado de Herbrand, "Recherches sur la théorie de la démonstration" (1930). O processo de eliminação dos quantificadores existenciais é oriundo de vários trabalhos de Skolem entre 1920 e 1928. O

trabalho de 1928, "Über die mathematische Logik", contém ainda resultados bem semelhantes aos de Herbrand. Os resultados sobre árvores semânticas da seção 3.5.2 originam-se de Robinson [1968] e de Kowalski e Hayes [1969]. Procedimentos de refutação baseados diretamente no Teorema de Herbrand foram implementados por volta de 1960 por Gilmore, Davis e Putnam e outros, mas mostraram ser bastante inefficientes.

3.2 CLÁUSULAS

Esta seção introduz a noção de cláusula através de uma nova linguagem formal, cuja semântica segue diretamente da semântica das linguagens de primeira ordem.

Uma cláusula é, intuitivamente, uma representação sintática muito simplificada de uma classe especial de fórmulas. Por exemplo, a fórmula

$$\forall x \forall y \forall z (\text{depende}(x,y) \vee \neg \text{chama}(x,y) \vee \neg \text{depende}(z,y))$$

pode ser representada pela seguinte seqüência

$$\text{depende}(x,y) \neg \text{chama}(x,y) \neg \text{depende}(z,y)$$

convencionando-se que as variáveis estão sempre universalmente quantificadas e que os elementos da seqüência formam uma disjunção. Esta seqüência é chamada de uma cláusula.

Em lugar de definir uma nova família de alfabetos especiais para cláusulas, no que se segue, por simplicidade, suporemos apenas que os alfabetos de primeira ordem agora contêm um novo símbolo lógico, " \square ", para denotar a cláusula vazia.

Definição 3.1:

Seja A um alfabeto de primeira ordem.

- (a) Um *literal positivo* sobre A é uma fórmula atômica sobre A .
- (b) Um *literal negativo* sobre A é a negação de uma fórmula atômica sobre A .
- (c) Um *literal* sobre A é ou um literal positivo ou um literal negativo sobre A .

- (d) Dois literais tem *sinais opostos* se e somente se um deles for positivo e o outro for negativo.
- (e) Dois literais são *complementares* se e somente se um deles for a negação do outro.
- (f) Uma fórmula atômica F é o *átomo* de um literal L , denotado por $|L|$, se e somente se L for F ou $\neg F$.

Definição 3.2:

Uma *cláusula* sobre um alfabeto de primeira ordem A é ou uma seqüência não vazia de literais sobre A ou a *cláusula vazia*, denotada por " \square ".

Se não for explicitado, tomaremos como alfabeto de um conjunto de cláusulas S o alfabeto de primeira ordem cujos símbolos não lógicos são aqueles que ocorrem em S .

Em quase toda discussão sobre a metateoria, principalmente na definição de regras de inferência, trataremos uma cláusula não-vazia " $L_1 \dots L_n$ " como o conjunto finito $\{L_1, \dots, L_n\}$ e a cláusula vazia " \square " como o conjunto vazio. Assim, utilizaremos as operações usuais de teoria dos conjuntos para definir novas cláusulas a partir de outras.

Letras maiúsculas do meio do alfabeto (L, M, N, \dots) indicarão literais, letras maiúsculas do começo do alfabeto (A, B, C, \dots) indicarão cláusulas, e letras maiúsculas em negrito do meio para o fim do alfabeto ($**S, R, \dots**$) denotarão conjuntos arbitrários de cláusulas, finitos ou não.

Definição 3.3:

A *linguagem de cláusulas* sobre um alfabeto de primeira ordem A é o conjunto de todas as cláusulas sobre A .

A semântica das linguagens de cláusulas segue diretamente da semântica das linguagens de primeira ordem através da seguinte definição:

Definição 3.4:

Uma estrutura I de um alfabeto de primeira ordem A satisfaaz uma cláusula não-vazia C sobre A (denotado por $I \models C$) se e somente se I satisfaaz a sentença F definida como

$$\forall x_1 \dots \forall x_m (L_1 \vee \dots \vee L_n)$$

onde x_1, \dots, x_m são as variáveis ocorrendo em C e L_1, \dots, L_n são os literais de C . Diz-se ainda que C e F são *equivalentes*.

Por convenção, a cláusula vazia é sempre insatisfatível.

Note que esta definição está bem formada pois F é uma fórmula sobre A . Note ainda que todas as variáveis de F são universalmente quantificadas, logo as avaliações das variáveis de A não afetam a satisfatibilidade de F .

Portanto, em termos de avaliação em uma estrutura, uma cláusula se comporta como uma sentença de primeira ordem, ou seja, como uma fórmula fechada. Assim, através das sentenças equivalentes às cláusulas, as noções de satisfatibilidade e implicação lógica estendem-se a conjuntos contendo indistintamente cláusulas e sentenças de primeira ordem.

Dado um conjunto S de cláusulas sobre um alfabeto A , por abuso de linguagem, as estruturas para A serão chamadas de estruturas de S .

Exemplo 3.1: Exemplos de Literais e Cláusulas

As seguintes expressões são literais:

$$\begin{aligned} & \text{depende}(x,y) \\ & \neg \text{chama}(x,z) \\ & \neg \text{depende}(z,y) \end{aligned}$$

A seqüência de literais abaixo é uma cláusula:

$$\text{depende}(x,y) \neg \text{chama}(x,z) \neg \text{depende}(z,y)$$

Esta cláusula é equivalente à seguinte fórmula:

$$\forall x \forall y \forall z (\text{depende}(x,y) \vee \neg \text{chama}(x,z) \vee \neg \text{depende}(z,y))$$

3.3 REPRESENTAÇÃO CLAUSAL DE FÓRMULAS E TEORIAS

3.3.1 Representação Clausal de Fórmulas

Quando comparadas com as fórmulas de primeira ordem, as cláusulas possuem a vantagem de uma sintaxe mais simples. Porém, esta simplicidade tornar-se-ia inútil se levasse a um decréscimo de expressividade. Os resultados desta seção mostram que, ao contrário, cláusulas podem substituir fórmulas se identificarmos a noção de expressividade com a noção de satisfatibilidade no seguinte sentido:

Definição 3.5:

Um conjunto de cláusulas S é uma *representação clausal* para uma fórmula P se e somente se P é satisfatível se e somente se S é satisfatível.

A obtenção da representação clausal de uma fórmula é um processo mecânico, como descrito a seguir:

Algoritmo 3.1: Algoritmo de Representação Clausal

entrada: uma fórmula P

saída: uma representação clausal S para P

(a) Tome o fecho existencial de P

Se P contiver uma variável livre x , substitua P por $\exists x(P)$. Repita este passo até que a fórmula não tenha variáveis livres.

(b) Elimine quantificadores redundantes

Elimine todo quantificador " $\forall x$ " ou " $\exists x$ " que não contenha nenhuma ocorrência livre de x no seu escopo. (Ou seja, elimine todo quantificador desnecessário).

(c) Renomeie variáveis quantificadas mais de uma vez

Se houver dois quantificadores governando a mesma variável, substitua a variável de um deles e todas as suas ocorrências livres no escopo do quantificador por uma nova variável que não ocorra na fórmula. Repita o processo até que todos os quantificadores governem variáveis diferentes.

(d) Elimine os conectivos " \rightarrow " e " \equiv "

Substitua:

$$\begin{array}{ll} (Q \rightarrow R) & \text{por } (\neg Q \vee R) \\ (Q \equiv R) & \text{por } (\neg Q \vee R) \wedge (Q \vee \neg R) \\ \neg(Q \rightarrow R) & \text{por } (Q \wedge \neg R) \\ \neg(Q \equiv R) & \text{por } (Q \wedge \neg R) \vee (\neg Q \wedge R) \end{array}$$

(e) Mova " \neg " para o interior da fórmula

Até que cada ocorrência de " \neg " preceda imediatamente uma fórmula atômica, substitua:

$$\begin{array}{ll} \neg \forall x(Q) & \text{por } \exists x(\neg Q) \\ \neg \exists x(Q) & \text{por } \forall x(\neg Q) \\ \\ \neg(Q \wedge R) & \text{por } (\neg Q \vee \neg R) \\ \neg(Q \vee R) & \text{por } (\neg Q \wedge \neg R) \\ \\ \neg \neg Q & \text{por } Q \end{array}$$

(f) Mova os quantificadores para o interior da fórmula (opcional)

Substitua, caso x não seja livre em R,

$$\begin{array}{ll} \forall x(Q \vee R) & \text{por } \forall x(Q) \vee R \\ \forall x(R \vee Q) & \text{por } R \vee \forall x(Q) \\ \forall x(Q \wedge R) & \text{por } \forall x(Q) \wedge R \\ \forall x(R \wedge Q) & \text{por } R \wedge \forall x(Q) \end{array}$$

$\exists x(Q \vee R)$	por	$\exists x(Q) \vee R$
$\exists x(R \vee Q)$	por	$R \vee \exists x(Q)$
$\exists x(Q \wedge R)$	por	$\exists x(Q) \wedge R$
$\exists x(R \wedge Q)$	por	$R \wedge \exists x(Q)$

Para sistematizar o processo, não aplique uma regra de conversão a um quantificador que contenha no seu escopo um outro quantificador ao qual uma regra se aplica.

Nota: este passo tenta minimizar o número de argumentos das funções introduzidas no passo (g) e justifica-se apenas por motivos práticos. No entanto, ele não garante que se obtenha no passo (g) funções com um número mínimo de argumentos.

(g) Elimine os quantificadores existenciais

Seja Q a fórmula corrente. Crie a nova fórmula corrente substituindo a subfórmula de Q da forma " $\exists y(R)$ ", que se situa mais à esquerda, por " $R[y/f(x_1, \dots, x_n)]$ ", onde x_1, \dots, x_n é uma lista de todas as variáveis livres de " $\exists y(R)$ " e " f " é qualquer símbolo funcional n -ário que não ocorre em Q . Quando não houver variáveis livres em " $\exists y(R)$ ", substitua " $\exists y(R)$ " por " $R[y/c]$ ", onde " c " é uma constante que não ocorre em Q . Repita o processo até que todos os quantificadores existenciais tenham sido eliminados.

Os novos símbolos funcionais assim introduzidos são chamados de *funções de Skolem* e o processo de substituição é chamado de *Skolemização*.

(h) Obtenha a forma normal prenex

Mova os quantificadores universais para a esquerda.

(i) Obtenha a forma normal conjuntiva

Até que a matriz da fórmula seja uma conjunção de disjunções, substitua:

$(Q \wedge R) \vee S$	por	$(Q \vee S) \wedge (R \vee S)$
$Q \vee (R \wedge S)$	por	$(Q \vee R) \wedge (Q \vee S)$

(j) Simplifique (opcional)

Transforme a fórmula Q resultante do passo (i) em outra mais simples S tal que S ainda esteja em forma normal conjuntiva (sem quantificadores existenciais) e Q seja satisfatível se e somente se S o for.

Por exemplo:

1. Elimine todas as ocorrências duplicadas de um literal em uma das disjunções de Q ;
2. Elimine as disjunções que contêm um literal e a sua negação (estas disjunções são sempre satisfatóveis).

(k) Obtenha a representação clausal

A representação clausal S da fórmula inicial P será o conjunto das cláusulas da forma $L_1 \dots L_n$ tais que $L_1 \vee \dots \vee L_n$ é uma disjunção da matriz da fórmula resultante do passo anterior.

Exemplo 3.2:

Seja P a fórmula abaixo:

$$\forall x \forall y (chama(x,y) \rightarrow (\exists u (programa(x,u)) \wedge \exists u (programa(y,u))))$$

Aplicaremos as transformações acima a P da seguinte forma:

Passos (a) e (b): Não são necessários pois P não possui variáveis livres ou quantificadores redundantes.

Passo (c): Renomeie u pois u ocorre quantificada mais de uma vez:

$$\forall x \forall y (chama(x,y) \rightarrow (\exists u (programa(x,u)) \wedge \exists v (programa(y,v))))$$

Passo (d): Elimine " \rightarrow :

$$\forall x \forall y (\neg chama(x,y) \vee (\exists u (programa(x,u)) \wedge \exists v (programa(y,v))))$$

Passo (e): Não é necessário mover " \neg " para o interior da fórmula pois a única ocorrência de " \neg " precede imediatamente uma fórmula atômica.

Passo (f): Não é necessário pois a última fórmula acima não permite mover quantificadores.

Passo (g): Elimine quantificadores existenciais da seguinte forma. A subfórmula " $\exists u (programa(x,u))$ " é a subfórmula mais à esquerda começando por um quantificador existencial. Como "x" é a única variável livre desta subfórmula, substitua "u" por "f(x)", onde "f" é um novo símbolo funcional, obtendo:

$$\forall x \forall y (\neg chama(x,y) \vee (programa(x,f(x)) \wedge \exists v (programa(y,v))))$$

Da mesma forma, substitua "v" por "g(y)", onde "g" é um novo símbolo funcional, obtendo:

$$\forall x \forall y (\neg chama(x,y) \vee (programa(x,f(x)) \wedge programa(y,g(y))))$$

Passo (h): Todos os quantificadores universais já estão à esquerda.

Passo (i): Obtenha a forma normal conjuntiva:

$$\begin{aligned} \forall x \forall y (& (\neg chama(x,y) \vee programa(x,f(x))) \wedge \\ & (\neg chama(x,y) \vee programa(y,g(y)))) \end{aligned}$$

Passo (j): Não há simplificações a fazer.

Passo (k): A representação clausal de P será então composta das cláusulas:

$$\begin{aligned} & \neg chama(x,y) \quad programa(x,f(x)) \\ & \neg chama(x,y) \quad programa(y,g(y)) \end{aligned}$$

Note que, em vista do passo de Skolemização, o conjunto final de cláusulas não é necessariamente sobre o mesmo alfabeto de primeira ordem que a fórmula original. Porém, o primeiro resultado importante deste capítulo

nos diz que se P é a entrada e S é a saída para o Algoritmo 3.1, então S realmente é uma representação clausal de P .

Inicialmente enunciaremos uma proposição auxiliar.

Proposição 3.1:

Seja Q uma sentença e suponha que implicações ou bicondicionais não ocorram em Q . Seja R uma subfórmula de Q e suponha que R não ocorre em Q no escopo de uma negação. Seja Q' a sentença obtida substituindo-se em Q a fórmula R por uma outra fórmula qualquer R' . Seja P o fecho universal de $(R \rightarrow R')$. Então, se $\{P, Q\}$ for satisfável, Q' também será satisfável.

Teorema 3.1: Teorema da Representação Clausal

Seja P a entrada e S a saída do Algoritmo de Representação Clausal. Então, P é satisfável se e somente se S é satisfável.

Demonstração

Mostraremos inicialmente que se A e A' são as fórmulas inicial e final dos passos de (a) a (j) do algoritmo, então A é satisfável se e somente se A' também é satisfável.

Argumentos simples estabelecem que as duas fórmulas de cada regra de transformação em todos os passos, exceto o passo (g), são logicamente equivalentes. Assim, se A e A' são a fórmula inicial e final de cada um destes passos, então A é satisfável se e somente se A' é satisfável.

A eliminação dos quantificadores existenciais é o único passo que necessita uma justificativa mais detalhada. Sejam A e A' a fórmula inicial e final do passo (g). Mostraremos, por indução sobre i , que A é satisfável se e somente se Q_i é, onde Q_i é a fórmula resultante da i -ésima eliminação de um quantificador existencial. Assim, teremos que A é satisfável se e somente se A' também é satisfável.

Base: para $i = 0$ não há nada a provar.

Passo de Indução: Suponha que A seja satisfável se e somente se Q_i é satisfável, para $i > 0$. Mostraremos que A é satisfável se e somente se Q_{i+1} é satisfável. Suponha que Q_i contém uma subfórmula R da forma

" $\exists y(S)$ ". Suponha ainda que esta seja a ocorrência mais à esquerda de um quantificador existencial em Q_i . Sejam x_1, \dots, x_n as variáveis livres de R e seja f o novo símbolo funcional n -ário resultante da Skolemização de R . Seja Q' a fórmula obtida de Q_i substituindo-se R por R' , onde R' é a fórmula $S[y/f(x_1, \dots, x_n)]$.

Mostraremos primeiro que se Q_i é satisfatível então Q' também é satisfatível. Suponha então que exista uma estrutura I , com universo U , que satisfaça Q_i . Construa uma estrutura I' idêntica a I , exceto que $I'(f)$ é a função definida da seguinte forma. Seja d_0 um elemento arbitrário de U . Para cada ênupla (d_1, \dots, d_n) em U^n , seja v uma avaliação tal que $v(x_i) = d_i$. Suponha que R não seja satisfeita por I para v . Então $I'(f)(d_1, \dots, d_n) = d_0$. Neste caso, temos ainda trivialmente que $(R \rightarrow R')$ é satisfeita por I' para v . Suponha agora que R seja satisfeita por I para v . Então, como R é a fórmula $\exists y(S)$, deve existir uma avaliação u idêntica a v , exceto em y , tal que S é satisfeita por I para u . Construa então $I'(f)(d_1, \dots, d_n) = u(y)$. Logo, como R' é a fórmula $S[y/f(x_1, \dots, x_n)]$ e como S é satisfeita por I para u , R' é satisfeita por I' para v , por construção de I' . Assim, $(R \rightarrow R')$ é satisfeita por I' para v .

Como consequência da construção de I' temos então que $(R \rightarrow R')$ é satisfeita por I' para v , para toda avaliação v . Portanto, I' satisfaz o fecho universal de $(R \rightarrow R')$. Além disto, I' também satisfaz Q'_i pois I satisfaz Q_i , I' coincide com I em todos os símbolos, exceto em f , e f não ocorre em Q_i .

Mas Q' foi obtida de Q_i substituindo-se R por R' . Além disto, Q'_i é uma fórmula sem variáveis livres, pelo passo (a) do algoritmo, e sem implicações e bicondiccionais, pelo passo (d). Também R não ocorre no escopo de uma negação em Q' , pelo passo (e). Logo, pela proposição anterior, I' satisfaz Q' .

Mostraremos agora que se Q' é satisfatível então Q_i também é satisfatível. Suponha então que exista uma estrutura I , com universo U , que satisfaça Q' .

Mostraremos inicialmente que I' satisfaz o fecho universal de $(R' \rightarrow R)$. Seja v uma avaliação. Suponha que R' não é satisfeita por I' para v . Então trivialmente $(R' \rightarrow R)$ é satisfeita por I' para v . Suponha que R' é satisfeita por I' para v . Seja u a avaliação tal que $u(z) = v(z)$, para toda variável z , exceto y , e $u(y) = I'(f)(v(x_1), \dots, v(x_n))$. Como R' é a fórmula $S[y/f(x_1, \dots, x_n)]$, temos então que S é satisfeita por I' para u , por construção

de u . Mas R é a fórmula $\exists y(S)$. Logo, R é satisfeita por I' para v . Assim, $(R' \rightarrow R)$ é satisfeita por I' para v .

Portanto, para toda avaliação v , $(R' \rightarrow R)$ é satisfeita por I' para v . Logo, I' satisfaz o fecho universal de $(R' \rightarrow R)$. Recorde ainda que I' satisfaz Q' por suposição.

Mas Q_i pode ser obtida de Q' substituindo-se R' por R . Além disto, Q' é uma fórmula sem variáveis livres e sem implicações e bicondicionais e R' não ocorre no escopo de uma negação em Q' . Logo, pela proposição anterior, I' satisfaz Q_i .

Por estes argumentos, Q_i é satisfatível se e somente se Q' o for. Pela hipótese de indução, A é satisfatível se e somente se Q_i o for. Logo A é satisfatível se e somente se Q' o for. Mas Q' é o resultado da eliminação do $(i+1)$ -ésimo quantificador existencial de A , ou seja, Q' é na verdade Q_{i+1} , logo a indução está completa.

Assim, se A e A' são a fórmula inicial e final do passo (g), então A é satisfatível se e somente se A' é satisfatível.

Temos então que se P é a fórmula submetida ao algoritmo e Q é a fórmula resultante do passo (j), então P é satisfatível se e somente se Q é satisfatível.

Finalmente, seja $S = \{C_1, \dots, C_k\}$ o conjunto de cláusulas resultante do passo (k). Seja C'_i a disjunção dos literais em C_i . Por construção de S , Q tem que ser da forma:

$$\forall x_1 \dots \forall x_n (C'_1 \wedge \dots \wedge C'_k)$$

Logo, Q é logicamente equivalente a

$$\forall x_1 \dots \forall x_n (C_1) \wedge \dots \wedge \forall x_1 \dots \forall x_n (C_k)$$

Como, por definição, C_i é satisfatível se e somente o fecho universal de C'_i for satisfatível, temos que trivialmente Q é satisfatível se e somente se S é satisfatível.

Assim, podemos concluir que P é satisfatível se e somente se S é satisfatível.

Como corolário deste resultado temos:

Corolário 3.1:

O problema da validade é redutível ao problema da insatisfatibilidade para conjuntos de cláusulas.

Demonstração

Dada uma fórmula F , seja S o conjunto de cláusulas resultante da aplicação do Algoritmo de Representação Clausal a $\neg F$. Temos que F é válida se e somente se $\neg F$ é insatisfatível. Mas, pelo Teorema 3.1, $\neg F$ é insatisfatível se e somente se S é insatisfatível. Logo, F é válida se e somente se S é insatisfatível. Assim, o Algoritmo de Representação Clausal leva a um procedimento de redução do problema da validade para o problema da insatisfatibilidade para conjuntos de cláusulas.

3.3.2 Representação Clausal de Teorias

Esta seção generaliza o resultado da seção anterior mostrando que é possível obter um conjunto de cláusulas que representa um conjunto de fórmulas em termos de satisfatibilidade.

A noção de representação clausal utilizada neste caso será a seguinte:

Definição 3.6:

Um conjunto de cláusulas S é uma *representação clausal* para um conjunto de fórmulas P se e somente se P é satisfatível se e somente se S é satisfatível.

Como na seção anterior, a questão principal consiste em definir um procedimento para obter a representação clausal de um conjunto de fórmulas.

Note inicialmente que, por meio de um artifício simples, é possível utilizar o Algoritmo de Representação Clausal para obter a representação de um conjunto finito $P = \{P_1, \dots, P_n\}$ de fórmulas. De fato, a aplicação do algoritmo à fórmula $P_1 \wedge \dots \wedge P_n$ resultará em uma representação clausal para P , pois P é satisfatível se e somente se a fórmula $P_1 \wedge \dots \wedge P_n$ o for. Porém nada se pode afirmar no caso de P ser um conjunto infinito de fórmulas.

Além disto se, por um lado, a formação explícita da conjunção $P_1 \wedge \dots \wedge P_n$ é inconveniente quando n for grande, por outro lado, a simulação implícita de tal conjunção poderá levar facilmente a enganos.

O principal resultado desta seção, que independe da cardinalidade do conjunto P , evita a formação da conjunção das fórmulas em P simulando explicitamente alguns passos do algoritmo. O resultado mostra que, com certos cuidados adicionais, é possível aplicar o Algoritmo de Representação Clausal individualmente a cada fórmula Q em P , obtendo uma representação clausal $CL(Q)$, e depois concluir que S , a união de todos os conjuntos $CL(Q)$, para $Q \in P$, é uma representação clausal de P .

O algoritmo abaixo captura os cuidados adicionais necessários à criação de S .

Algoritmo 3.2: Algoritmo Geral de Representação Clausal

- entrada:** um conjunto de fórmulas P
- saída:** uma representação clausal S para P

(a) Eliminação das variáveis livres.

Seja x_1, x_2, \dots uma ordenação das variáveis livres ocorrendo nas fórmulas em P e c_1, c_2, \dots uma lista de constantes distintas não ocorrendo nas fórmulas em P . Para cada $Q \in P$, se x_i ocorre livre em Q , substitua x_i por c_i .

(b) Obtenção da representação clausal

Seja P' o conjunto obtido no passo anterior. Para cada $Q \in P'$, aplique o Algoritmo de Representação Clausal a Q obtendo um conjunto $CL(Q)$ de cláusulas. Ao aplicar o algoritmo, certifique-se de que, para todo par de fórmulas Q e R em P' , as funções de Skolem introduzidas em $CL(Q)$ e em $CL(R)$ sejam distintas. O conjunto S será a união dos conjuntos $CL(Q)$, para todo $Q \in P'$.

Note que, no caso das fórmulas em P serem todas fechadas, o passo (a) é desnecessário. Logo, só é necessário certificar-se de que as funções de

Skolem introduzidas em $CL(Q)$ e em $CL(R)$ são distintas, para todo par de sentenças Q e R em P.

A condição imposta à utilização do Algoritmo de Representação Clausal no passo (b) apenas estende a política de seleção de funções de Skolem a conjuntos de fórmulas. Já o passo (a) evita problemas como os ilustrados no seguinte exemplo.

Exemplo 3.3:

Seja $P = \{p(x), \neg p(x)\}$ um conjunto de fórmulas. Aplicando o Algoritmo de Representação Clausal da seção anterior a " $p(x)$ " obtemos:

passo (a): tome o fecho existencial, obtendo " $\exists x(p(x))$ "

passo (g): elimine o quantificador existencial através da escolha da constante "a", obtendo " $p(a)$ "

Logo, " $p(a)$ " é uma representação clausal de " $p(x)$ ".

Aplicando agora o Algoritmo de Representação Clausal a " $\neg p(x)$ " obtemos:

passo (a): tome o fecho existencial, obtendo " $\exists x(\neg p(x))$ "

passo (g): elimine o quantificador existencial através da escolha da constante "b", obtendo " $\neg p(b)$ "

Logo, " $\neg p(b)$ " é uma representação clausal de " $\neg p(x)$ ".

Note agora que $S = \{p(a), \neg p(b)\}$ é satisfatível, enquanto que o conjunto P não é satisfatível. Portanto, S não é uma representação clausal para P.

No entanto, o Algoritmo Geral de Representação Clausal gera corretamente uma representação clausal para P:

passo (a): Substitua uniformemente cada variável livre das fórmulas em P por constantes, obtendo o conjunto $\{p(a), \neg p(a)\}$

passo (b): Aplique o Algoritmo de Representação Clausal separadamente a cada fórmula em \mathbf{P} , mas escolhendo funções de Skolem distintas. O resultado será o mesmo conjunto acima, que é uma representação clausal correta de \mathbf{P} .

A correção do Algoritmo Geral de Representação Clausal segue do seguinte resultado.

Teorema 3.2:

Seja \mathbf{P} a entrada e \mathbf{S} a saída do Algoritmo Geral de Representação Clausal. Então, \mathbf{P} é satisfatível se e somente se \mathbf{S} é satisfatível.

Demonstração

Caso 1: Suponha que \mathbf{P} seja finito.

Considere a aplicação do algoritmo geral a um conjunto finito \mathbf{P} . Suponha que $\mathbf{P}' = \{\mathbf{P}_1', \dots, \mathbf{P}_n'\}$ seja o conjunto obtido ao final passo (a). Logo, pela escolha das constantes e pelo processo de substituição, \mathbf{P} é satisfatível se e somente se \mathbf{P}' é satisfatível. Seja \mathbf{S} o conjunto de cláusulas obtido pelo passo (b) e \mathbf{C} a fórmula $\mathbf{P}_1' \wedge \dots \wedge \mathbf{P}_n'$. Pelas restrições impostas à escolha das funções de Skolem no passo (b), \mathbf{S} é uma possível saída para a aplicação do Algoritmo de Representação Clausal a \mathbf{C} . Logo, pelo Teorema 3.1, \mathbf{C} é satisfatível se e somente se \mathbf{S} é satisfatível. Além disto, \mathbf{P}' é satisfatível se e somente se \mathbf{C} é satisfatível. Logo, \mathbf{P} é satisfatível se e somente se \mathbf{S} é satisfatível.

Caso 2: Suponha que \mathbf{P} não seja finito.

Suponha que \mathbf{S} seja satisfatível. Seja \mathbf{G} um subconjunto finito de \mathbf{P} e $CL(\mathbf{G})$ a união dos conjuntos $CL(Q)$, para $Q \in \mathbf{G}$. Como \mathbf{S} é satisfatível, $CL(\mathbf{G})$ é satisfatível. Logo, pelo caso 1, \mathbf{G} é satisfatível. Assim, todo subconjunto finito de \mathbf{P} é satisfatível. Logo, pelo Teorema da Compacidade, \mathbf{P} é satisfatível.

Suponha que \mathbf{P} seja satisfatível. Seja \mathbf{F} um subconjunto finito de \mathbf{S} . Logo, existe um subconjunto finito e satisfatível \mathbf{G} de \mathbf{P} tal que $\mathbf{F} \subset CL(\mathbf{G})$. Pelo caso 1, $CL(\mathbf{G})$ e, logo, \mathbf{F} são satisfatíveis. Assim, todo subconjunto finito de \mathbf{S} é satisfatível. Novamente pelo Teorema da Compacidade, que vale também para conjuntos de cláusulas, \mathbf{S} é satisfatível.

Portanto, **S** é satisfatível se e somente se **P** é satisfatível.

O Teorema 3.2 possui um corolário simples, mas importante, enunciado da seguinte forma:

Corolário 3.2:

O problema da implicação lógica é reduzível ao problema da insatisfatibilidade para conjuntos de cláusulas.

Demonstração

Dados um conjunto de fórmulas **R** e uma fórmula **Q**, construa um conjunto de cláusulas **S** da seguinte forma:

1. Forme o conjunto $\mathbf{P} = \mathbf{R} \cup \{\neg Q\}$;
2. Aplique o algoritmo geral a **P**, obtendo o conjunto **S**.

Por definição, **R** implica logicamente **Q** se e somente se **P** é insatisfatível. Mas, pelo Teorema 3.2, **P** é insatisfatível se e somente se **S** é insatisfatível. Logo, **R** implica logicamente **Q** se e somente se **S** é insatisfatível. Assim, o Algoritmo Geral de Representação Clausal leva a um procedimento de redução do problema da implicação lógica para o problema da insatisfatibilidade para conjuntos de cláusulas.

A aplicação mais frequente deste corolário ocorre no contexto de uma teoria, ou seja, quando **R** são os axiomas de uma teoria e **Q** é uma hipótese que se deseja testar.

Note que a construção de **P** exige que todos os quantificadores universais iniciais, freqüentemente omitidos, sejam explicitados. No caso das fórmulas em **R**, esta providência é necessária para evitar que o passo (a) do algoritmo geral substitua erroneamente as variáveis universalmente quantificadas por constantes. No caso da fórmula **Q**, isto evita a inversão de quantificadores. De fato, suponha que x seja uma variável de **Q** governada por um quantificador universal omitido do prefixo de **Q**. Explicitando-se o quantificador, a hipótese se transforma em $\forall x(Q)$, cuja negação é $\exists x(\neg Q)$. Se **Q** primeiro fosse negada e depois o quantificador universal explicitado, o resultado final seria erroneamente $\forall x(\neg Q)$.

Concluiremos esta seção com a representação clausal de algumas teorias anteriormente apresentadas.

Exemplo 3.4: Representação Clausal de um Estado do Dicionário

O seguinte conjunto de cláusulas é uma representação clausal do estado do dicionário apresentado na Seção 2.5.1, item (E):

1. *programa(a,fortran)*
2. *programa(b,pascal)*
3. *programa(c,fortran)*
4. *arquivo(d,sequencial)*
5. *arquivo(e,direto)*
6. *chama(a,b)*
7. *chama(a,c)*
8. *usa(a,d)*
9. *usa(b,e)*
10. $\neg chama(x,y) \text{ depende}(x,y)$
11. $\neg usa(x,y) \text{ depende}(x,y)$
12. $\neg depende(x,z) \neg depende(z,y) \text{ depende}(x,y)$

Exemplo 3.5: Representação Clausal da Teoria de Listas

Seja **A** o conjunto de átomos em questão. O seguinte conjunto de cláusulas é uma representação clausal para a teoria de listas apresentada na seção 2.5.2, itens (B) e (C), seria:

- A0. *átomo(a)* , para cada $a \in A$
- L0. *lista(nil)*
- L1a. $\neg átomo(x) \neg lista(y) lista((x.y))$
- L1b. $\neg lista(x) \neg lista(y) lista((x.y))$
- L1c. $\neg lista((x.y)) átomo(x) lista(x)$
- L1d. $\neg lista((x.y)) lista(y)$
- AL. $\neg átomo(x) \neg lista(x)$
- CA1. $\neg cabeça(x,nil)$
- CA2. $\neg lista((x.y)) cabeça(x,(x.y))$
- CD1. *cauda(nil,nil)*
- CD2. $\neg lista((x.y)) cauda(y,(x.y))$
- ME1. $\neg membro(x,nil)$
- ME2. $\neg lista((x.y)) membro(x,(x.y))$
- ME3. $\neg lista((x.y)) \neg membro(z,y) membro(z,(x.y))$

CO1. $\neg lista(x) concat(nil,x,x)$

CO2. $\neg lista((u.x)) \neg lista(y) \neg lista((u.z)) \neg concat(x,y,z)$
 $concat((u.x),y,(u.z))$

Note que se A não for finito, tanto a teoria das listas quanto o conjunto acima de cláusulas não serão finitos.

3.4 PRIMEIRO ENUNCIADO DO TEOREMA DE HERBRAND

3.4.1 Estruturas de Herbrand

Esta seção introduz a classe das estruturas de Herbrand, que são estruturas bastante simples construídas a partir dos elementos sintáticos das próprias cláusulas. Esta classe de estruturas é interessante pois o problema da insatisfatibilidade para conjuntos de cláusulas se reduz ao problema da insatisfatibilidade para conjuntos de cláusulas considerando apenas a classe das estruturas de Herbrand.

Definição 3.7:

Seja S um conjunto de cláusulas. O *universo de Herbrand* para S , $UH[S]$, é o menor conjunto tal que

- (i) toda constante ocorrendo em S pertence a $UH[S]$. Se nenhuma constante ocorrer em S , então $UH[S]$ conterá uma constante escolhida arbitrariamente;
- (ii) se f é um símbolo funcional n -ário ocorrendo em S e se t_1, \dots, t_n pertencem a $UH[S]$, então o termo $f(t_1, \dots, t_n)$ também pertence a $UH[S]$.

Um *termo de Herbrand* ou *termo básico* é um elemento de $UH[S]$.

O universo de Herbrand de S coincide então com o conjunto dos termos sem variáveis gerados a partir das constantes (ou de uma constante escolhida arbitrariamente, se nenhuma ocorrer em S) e dos símbolos funcionais ocorrendo em S .

Registraremos a seguir, para uso futuro, alguns conceitos correlatos.

Definição 3.8:

Seja \mathbf{S} um conjunto de cláusulas e $UH[\mathbf{S}]$ o universo de Herbrand para \mathbf{S} .

- (a) A *base de Herbrand* para \mathbf{S} é o conjunto $BH[\mathbf{S}]$ de todas as fórmulas atômicas da forma $p(t_1, \dots, t_n)$, onde p é um símbolo predicativo n -ário ocorrendo em \mathbf{S} e t_1, \dots, t_n são termos em $UH[\mathbf{S}]$, para todo $n > 0$.
- (b) Um *literal básico* de \mathbf{S} é ou um elemento ou a negação de um elemento de $BH[\mathbf{S}]$.

Definição 3.9:

Uma estrutura I para o alfabeto A de um conjunto \mathbf{S} de cláusulas é uma *estrutura de Herbrand* para \mathbf{S} se e somente se

- (i) o universo de I é o universo de Herbrand $UH[\mathbf{S}]$;
- (ii) para cada constante c de A , $I(c) = c$;
- (iii) para cada símbolo funcional n -ário f de A , para cada ênupla (t_1, \dots, t_n) em $UH[\mathbf{S}]$, $I(f)(t_1, \dots, t_n) = f(t_1, \dots, t_n)$,

Definição 3.10:

Uma estrutura de Herbrand I para \mathbf{S} é um *modelo de Herbrand* para \mathbf{S} se e somente se I satisfaz a todas as cláusulas em \mathbf{S} .

Os termos de Herbrand representam um duplo papel pois, como termos do alfabeto de \mathbf{S} , são objetos sintáticos e, como elementos do universo de I , são objetos semânticos. A valoração das constantes e dos símbolos funcionais em I reflete diretamente este duplo papel, já que uma constante "c" denota ela própria em I , e a função associada a "f" mapeia os termos t_1, \dots, t_n , como elementos de $UH[\mathbf{S}]$, no termo " $f(t_1, \dots, t_n)$ ", também como elemento de $UH[\mathbf{S}]$. Note ainda que todas as estruturas de Herbrand para \mathbf{S} possuem o mesmo universo e a mesma valoração para as constantes e símbolos funcionais do alfabeto de \mathbf{S} , diferindo portanto apenas na valoração dos símbolos predicativos.

Exemplo 3.6:

Seja \mathbf{S} o seguinte conjunto de cláusulas:

1. $\neg chama(x,y) \text{ programa}(x,f(x))$
2. $\neg chama(x,y) \text{ programa}(y,g(y))$

O universo de Herbrand para \mathbf{S} é o seguinte conjunto, onde "a" é uma constante escolhida arbitrariamente:

$$UH[\mathbf{S}] = \{a, f(a), g(a), f(f(a)), f(g(a)), \dots\}$$

Toda estrutura de Herbrand I satisfará então às seguintes condições:

$$\begin{aligned} I(a) &= a \\ I(f)(t) &= f(t), \text{ para todo } t \in UH[\mathbf{S}] \\ I(g)(t) &= g(t), \text{ para todo } t \in UH[\mathbf{S}] \end{aligned}$$

A especificação de uma estrutura de Herbrand se resume então à descrição do significado de *chama* e *programa*. Por exemplo, a especificação de uma estrutura I seria:

$$\begin{aligned} I(chama) &= \{(a,a), (a,f(a))\} \\ I(programa) &= \{(a,a), (a,g(f(a)))\} \end{aligned}$$

3.4.2 Uma Primeira Prova do Teorema de Herbrand

Esta seção enuncia um resultado fundamental para o desenvolvimento de procedimentos de refutação, o Teorema de Herbrand. Este resultado mostra que é possível reduzir o problema de testar a insatisfatibilidade de um dado conjunto \mathbf{S} de cláusulas ao problema de testar a existência de um conjunto finito e insatisfatível de fórmulas da Lógica Sentencial, gerado de forma sistemática a partir de \mathbf{S} . Duas demonstrações para este resultado serão apresentadas: uma bastante concisa, baseada no Teorema da Compacidade, e outra baseada no conceito de árvores semânticas (ver seção 3.5.2).

A prova do Teorema de Herbrand baseia-se em certas propriedades básicas das estruturas de Herbrand. A primeira delas indica que, em termos de satisfatibilidade de conjuntos de cláusulas, basta trabalhar com estruturas de Herbrand.

Lema 3.1:

Um conjunto de cláusulas \mathbf{S} é satisfatível se e somente se \mathbf{S} é satisfatível por uma estrutura de Herbrand.

Demonstração

(\leftarrow) Se \mathbf{S} é satisfatível por uma estrutura de Herbrand, então \mathbf{S} é satisfatível.

(\rightarrow) Suponha que exista uma estrutura I que satisfaz \mathbf{S} . Seja m um mapeamento de $UH[\mathbf{S}]$ no universo U de I . Seja H a estrutura de Herbrand para \mathbf{S} construída da seguinte forma: para cada $n > 0$, para cada símbolo proposicional n -ário p da linguagem de \mathbf{S} , para cada ênupla (h_1, \dots, h_n) de elementos de $UH[\mathbf{S}]$, $(h_1, \dots, h_n) \in H(p)$ se e somente se $(m(h_1), \dots, m(h_n)) \in I(p)$. Então, por construção, H satisfaz \mathbf{S} .

Corolário 3.3:

O problema da insatisfatibilidade para conjuntos de cláusulas é redutível ao problema da insatisfatibilidade para conjuntos de cláusulas na classe das estruturas de Herbrand.

A segunda propriedade mostra que testar satisfatibilidade de uma cláusula em uma estrutura de Herbrand se reduz a uma tarefa simples.

Definição 3.11:

Uma *instância básica* de uma cláusula C em um conjunto \mathbf{S} de cláusulas é uma cláusula obtida substituindo-se uniformemente cada variável de C por um termo de Herbrand de \mathbf{S} .

Lema 3.2:

Seja C uma cláusula em um conjunto \mathbf{S} de cláusulas. Uma estrutura de Herbrand I para \mathbf{S} satisfaz C se e somente se I satisfaz a todas as instâncias básicas de C .

Demonstração

(\rightarrow) Suponha que I satisfaça C . Por convenção, C representa a fórmula $\forall x_1 \dots \forall x_n (D)$, onde x_1, \dots, x_n são as variáveis livres de C e D é a disjunção dos literais de C . Logo I satisfaz C se e somente se

$$(1) \quad v[v, I](\forall x_1 \dots \forall x_n (D)) = V, \text{ para alguma valorização } v \text{ sobre } UH[\mathbf{S}]$$

Por definição, (1) implica em:

$$(2) \quad v[u, I](D) = V, \text{ para toda valorização } u \text{ sobre } UH[S]$$

Seja B uma instância básica de C . Logo existem t_1, \dots, t_n em $UH[S]$ tais que $B = C[x_1/t_1, \dots, x_n/t_n]$. Seja w uma valorização sobre $UH[S]$ tal que $w(x_i) = t_i, i=1, \dots, n$. Logo, por (2) e por construção de w , temos:

$$(3) \quad v[w, I](D) = V$$

Seja $E = D[x_1/t_1, \dots, x_n/t_n]$. Logo, por construção de w e por (3):

$$(4) \quad v[w, I](E) = V$$

Logo, I satisfaz a E . Mas, como E não tem variáveis livres, B é equivalente a E . Assim, por definição, I satisfaz B . Portanto, I satisfaz a toda instância básica de C .

(\leftarrow) Suponha que I satisfaça B , para toda instância básica B de C . Por argumento semelhante à primeira parte da prova, temos:

$$(1) \quad v[u, I](D) = V, \text{ para toda valorização } u \text{ sobre } UH[S]$$

Logo,

$$(2) \quad v[v, I](\forall x_1 \dots \forall x_n(D)) = V, \text{ para alguma valorização } v \text{ sobre } UH[S]$$

Assim, I satisfaz o fecho universal de D . Logo, por definição, I satisfaz C .

O Teorema de Herbrand é uma consequência direta destes dois lemas e do Teorema da Compacidade:

Teorema 3.3: (Teorema de Herbrand)

Um conjunto de cláusulas S é insatisfatível se e somente se existe um conjunto finito de instâncias básicas das cláusulas de S que é insatisfatível.

Demonstração

(\leftarrow) Provaremos que se S é satisfatível, então todo conjunto finito de instâncias básicas é satisfatível. Suponha que S seja satisfatível. Pelo Lema 3.1, existe então uma estrutura de Herbrand I para S tal que I satisfaz toda cláusula C em S . Pelo Lema 3.2, I satisfaz toda instância básica de C . Logo, I satisfaz todas as instâncias básicas de S . Em particular, I satisfaz qualquer conjunto finito de instâncias básicas de S .

(\rightarrow) Provaremos que se todo conjunto finito B de instâncias básicas de S é satisfatível, S também é satisfatível. Mas se todo conjunto finito B de instâncias básicas de S é satisfatível, pelo Teorema da Compacidade, o conjunto das instâncias básicas de S também é satisfatível. Pelo Lema 3.1, este conjunto é então satisfatível por uma estrutura de Herbrand I . Logo, pelo Lema 3.2, I satisfaz todas as cláusulas em S . Assim, I satisfaz S .

3.5 SEGUNDO ENUNCIADO DO TEOREMA DE HERBRAND

3.5.1 H-Interpretações

Esta seção apresenta uma caracterização surpreendentemente simples para satisfatibilidade, baseada no conceito de H-interpretação. Intuitivamente, uma H-interpretação para um conjunto de cláusulas é um conjunto de literais básicos capturando a mesma informação que uma estrutura de Herbrand. Porém, para testar se uma H-interpretação satisfaz uma cláusula, não é necessário usar a definição original de satisfatibilidade: basta realizar algumas operações elementares com conjuntos. O exemplo abaixo desenvolve um pouco mais estas observações.

Exemplo 3.7:

Seja S o seguinte conjunto de cláusulas:

1. $\neg \text{chama}(x,y) \text{ programa}(x,f(x))$
2. $\neg \text{chama}(x,y) \text{ programa}(y,g(y))$

cujo universo de Herbrand é ("a" é uma constante escolhida arbitrariamente):

$$UH[\mathbf{S}] = \{a, f(a), g(a), f(f(a)), f(g(a)), \dots\}$$

e cuja base de Herbrand é

$$BH[\mathbf{S}] = \{chama(a,a), programa(a,a), chama(a,f(a)), \dots\}$$

Defina o conjunto de instâncias básicas de cláusulas de \mathbf{S} , $IB[\mathbf{S}]$, tal que uma cláusula B pertence a $IB[\mathbf{S}]$ se e somente se existem termos t, u em $UH[\mathbf{S}]$ tais que B é da forma

$$\neg chama(t,u) \quad programa(t,f(t))$$

ou da forma

$$\neg chama(t,u) \quad programa(u,g(u))$$

Seja I a estrutura de Herbrand para \mathbf{S} tal que:

$$I(chama) = \emptyset$$

$$I(programa) = \{(a,a)\}$$

Mostraremos que I é um modelo de Herbrand para \mathbf{S} . Inicialmente construa o seguinte conjunto de literais básicos em $BH[\mathbf{S}]$:

$$H = \{programa(a,a)\}$$

Note que H captura completamente a informação em I . Construa agora o conjunto $exp(H)$ contendo todos os literais em H e a negação de todos os literais em $BH[\mathbf{S}]$ que não estão em H :

$$exp(H) = H \cup \{\neg L / L \in (BH[\mathbf{S}] - H)\}$$

É possível mostrar que:

$$exp(H) = H \cup$$

$$\{\neg chama(t,u) / t, u \in UH[\mathbf{S}]\} \cup$$

$$\{\neg programa(t,u) / t, u \in UH[\mathbf{S}] \text{ e } (t,u) \neq (a,a)\}$$

I é um modelo de S se e somente se I satisfizer cada cláusula C em S . Mas, pelo Lema 3.2, I satisfaz C se e somente se I satisfizer a toda instância básica B de C . Além disto, como B não possui variáveis e como B representa a disjunção dos seus literais, I satisfaz B se e somente se I satisfizer a algum literal L de B . Por construção de H , I satisfará então L se e somente se $L \in \exp(H)$. Assim, I satisfará S se e somente se, para toda cláusula C em S , para toda instância básica B de C , algum literal de B pertence a $\exp(H)$.

Por este argumento, I satisfaz S pois, para toda instância básica B em $IB[S]$, B contém um literal da forma $\neg chama(t,u)$ que pertence a $\exp(H)$, para algum $t,u \in UH[S]$.

A generalização do exemplo anterior leva às seguintes definições.

Definição 3.12:

Seja S um conjunto de cláusulas.

- (a) Uma H -interpretação para S é um conjunto H de literais em $BH[S]$.
- (b) A expansão de uma H -interpretação H é o conjunto:

$$\exp(H) = H \cup \{ \neg L / L \in (BH[S] - H) \}$$

- (c) Uma H -interpretação H para S é um H -modelo para S se e somente se, para toda instância básica B de uma cláusula em S , B possui algum literal em $\exp(H)$.
- (d) S é H -satisfatível se e somente se S tem um H -modelo.
- (e) S é H -insatisfatível se e somente se S não é H -satisfatível.

Note como esta definição de modelo é extremamente mais simples do que a original dada na Seção 2.3 pois um modelo agora é apenas uma coleção de literais e satisfatibilidade se reduz a um teste simples. Provaremos a seguir que, em certo sentido, esta definição de modelo coincide com a original quando estamos trabalhando com conjuntos de cláusulas.

Lema 3.3:

S é satisfatível se e somente se S é H -satisfatível.

Demonstração

(\rightarrow) Suponha que \mathbf{S} seja satisfatível. Pelo Lema 3.1, \mathbf{S} é então satisfatível por uma estrutura de Herbrand I . Seja H a H-interpretação de \mathbf{S} tal que, para todo $n > 0$, para todo símbolo predicativo n -ário p , para todo t_1, \dots, t_n em $UH[\mathbf{S}]$, $p(t_1, \dots, t_n) \in H$ se e somente se $(t_1, \dots, t_n) \in I(p)$. Provaremos por contradição que H é um H-modelo de \mathbf{S} , logo que \mathbf{S} é H-satisfatível. Suponha que H não seja um H-modelo para \mathbf{S} . Então existe uma instância básica B de uma cláusula C de \mathbf{S} tal que B não possui um literal em $\text{exp}(H)$. Por construção de H , I não satisfaz então B . Pelo Lema 3.2, I não satisfaz então C , logo não satisfaz \mathbf{S} . Contradição.

(\leftarrow) Suponha que \mathbf{S} seja H-satisfatível. Seja H um H-modelo de \mathbf{S} . Seja I a interpretação de Herbrand tal que, para todo $n > 0$, para todo símbolo predicativo n -ário p , para todo t_1, \dots, t_n em $UH[\mathbf{S}]$, $(t_1, \dots, t_n) \in I(p)$ se e somente se $p(t_1, \dots, t_n) \in H$. Por um argumento semelhante ao anterior podemos então mostrar que I satisfaz \mathbf{S} .

3.5.2 Árvores Semânticas

Árvores semânticas oferecem um segundo caminho para provar o Teorema de Herbrand. Elas serão também o ponto de partida para a prova da completude do método da resolução apresentado no capítulo seguinte.

Uma árvore semântica binária A para um conjunto \mathbf{S} de cláusulas deve ser entendida como uma enumeração sistemática das H-interpretações de \mathbf{S} . A árvore será completa se a enumeração for exaustiva e será fechada se \mathbf{S} for H-insatisfatível. As definições a seguir formalizam estes conceitos.

Definição 3.13:

Seja \mathbf{S} um conjunto de cláusulas. Uma árvore binária A com as arestas rotuladas por literais básicos de \mathbf{S} é uma *árvore semântica binária* para \mathbf{S} se e somente se

- (i) para todo nó N de A , os rótulos das arestas ligando N a seus filhos são literais básicos complementares;
- (ii) duas arestas em um mesmo ramo de A não são rotuladas com literais complementares ou com o mesmo literal.

Se r for um caminho em A , $\text{rot}(r)$ denotará o conjunto de todos os literais que rotulam arestas de r . Da mesma forma, se N for um nó de A , $\text{rot}(N)$ denotará o conjunto de todos os literais que rotulam as arestas do caminho da raiz de A até N .

Definição 3.14:

Uma árvore semântica binária A para um conjunto S de cláusulas é *completa* se e somente se, para todo ramo r de A , existe uma H-interpretação H de S tal que $\text{rot}(r) = \text{exp}(H)$.

No que se segue, usaremos $\text{comp}(C)$ para denotar o conjunto dos literais complementares aos literais de uma cláusula C . Por exemplo, se C é a cláusula " $\neg \text{chama}(x,y) \text{ programa}(x,f(x))$ ", então

$$\text{comp}(C) = \{\text{chama}(x,y), \neg \text{programa}(x,f(x))\}.$$

Definição 3.15:

Seja A uma árvore semântica binária para um conjunto S de cláusulas.

- (a) um nó N de A é um *nó de fracasso* se e somente se
 - (i) existe uma instância básica B de uma cláusula de S tal que $\text{comp}(B) \subset \text{rot}(N)$;
 - (ii) nenhum ancestral de N é um nó de fracasso.
- (b) A é *fechada* se e somente se todo ramo de A possui um nó de fracasso a uma distância finita da raiz, isto é, se todo ramo de A é *finitamente fechado*.

Exemplo 3.8:

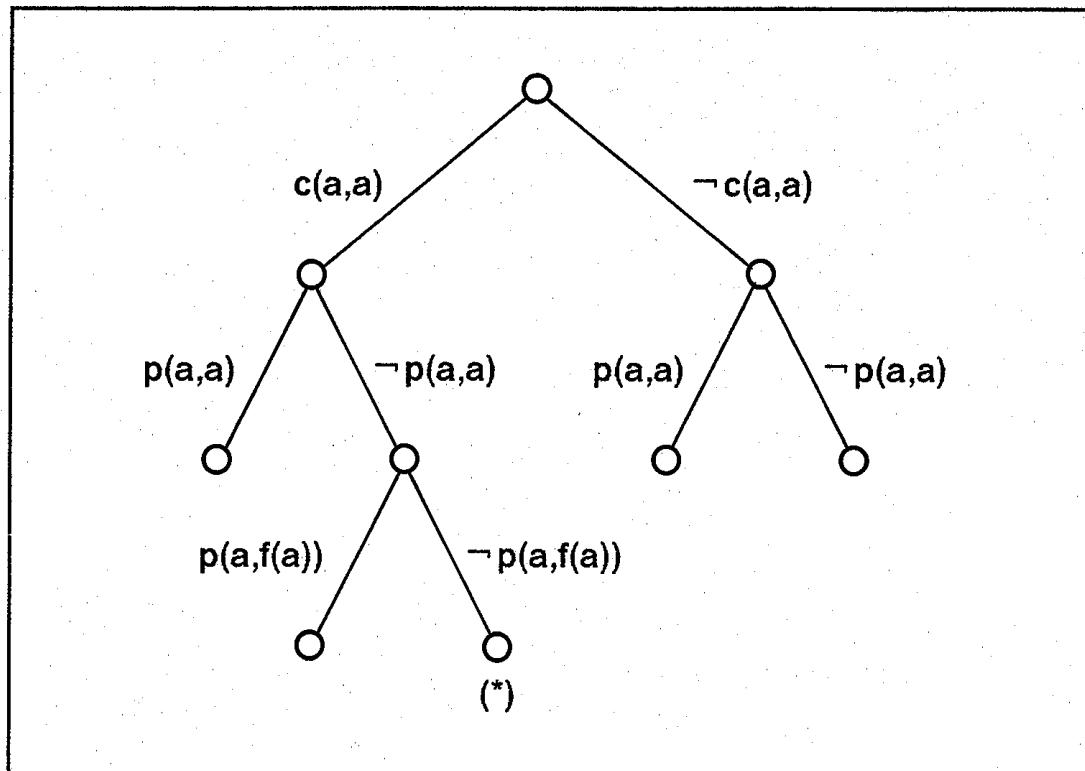
Seja S o seguinte conjunto de cláusulas:

1. $\neg \text{chama}(x,y) \text{ programa}(x,f(x))$
2. $\neg \text{chama}(x,y) \text{ programa}(y,g(y))$

cujo universo de Herbrand é ("a" é uma constante escolhida arbitrariamente):

$$UH[S] = \{a, f(a), g(a), f(f(a)), f(g(a)), \dots\}$$

A árvore abaixo é uma árvore semântica binária (incompleta) para **S** (abreviando "chama" por "c" e "programa" por "p"):



Seja N o nó marcado com (*). Então:

$$\text{rot}(N) = \{\text{chama}(a,a), \neg\text{programa}(a,a), \neg\text{programa}(a,f(a))\}$$

Seja B a seguinte instância básica da primeira cláusula:

$$\neg\text{chama}(a,a) \text{ programa}(a,f(a))$$

Como todos os literais em $\text{comp}(B)$ ocorrem em $\text{rot}(N)$ e esta propriedade não vale para os ancestrais de N , N é um nó de fracasso. Isto significa que, se H é uma H -interpretação tal que $\text{exp}(H)$ contém $\text{rot}(N)$, então B é falsa em H .

Os resultados abaixo generalizam esta última observação.

Proposição 3.2:

Uma H -interpretação H para um conjunto de cláusulas **S** é um H -modelo para **S** se e somente se, para toda instância básica B de uma cláusula em **S**, $\text{comp}(B)$ não é um subconjunto de $\text{exp}(H)$.

Demonstração

(\rightarrow) Suponha que H seja um H-modelo para S . Seja B uma instância básica de uma cláusula em S . Logo, por definição de H-modelo, existe um literal L de B tal que $L \in \exp(H)$. Seja L' o literal complementar a L . Como L pertence a $\exp(H)$, L' não pertence a $\exp(H)$. Logo, como $L' \in \text{comp}(B)$, $\text{comp}(B)$ não é um subconjunto de $\exp(H)$.

(\leftarrow) Segue de forma semelhante.

Proposição 3.3:

Seja A uma árvore semântica binária para um conjunto de cláusulas S . Para toda H-interpretação H para S , existe um ramo r de A tal que $\text{rot}(r) \subset \exp(H)$.

Demonstração

Seja A uma árvore semântica binária para S e H uma H-interpretação para S . Provaremos inicialmente, por indução sobre n , que existe um caminho r em A de comprimento n tal que $\text{rot}(r) \subset \exp(H)$, para todo $n \geq 0$.

Base: Para $n=0$ não há nada a provar pois $\text{rot}(r) = \emptyset$.

Passo de Indução: Suponha que exista um caminho r em A de comprimento $n-1$ tal que $\text{rot}(r) \subset \exp(H)$. Sejam L e $\neg L$ os literais rotulando as arestas saindo do último nó de r . Por definição de árvore semântica, tanto L quanto $\neg L$ não rotulam arestas de r . Porém, se $L \in H$ então $L \in \exp(H)$ senão $\neg L \in \exp(H)$. Logo, existe um caminho r' de comprimento n , estendendo r , tal que $\text{rot}(r') \subset \exp(H)$.

Portanto, para todo $n > 0$, existe um caminho r de comprimento n em A tal que $\text{rot}(r) \subset \exp(H)$. Logo, existe um ramo s em A tal que $\text{rot}(s) \subset \exp(H)$.

Lema 3.4:

Seja \mathbf{S} um conjunto de cláusulas.

- (a) se \mathbf{S} é insatisfatível então toda árvore semântica binária completa para \mathbf{S} é fechada.
- (b) se existe uma árvore semântica binária fechada para \mathbf{S} então \mathbf{S} é insatisfatível.

Demonstração

(a) Suponha que \mathbf{S} seja insatisfatível. Logo, \mathbf{S} é H-insatisfatível, pelo Lema 3.3. Seja A uma árvore semântica binária para \mathbf{S} e suponha que A seja completa. Mostraremos que A é fechada. Seja r um ramo de A . Como A é completa, existe uma H-interpretação H para \mathbf{S} tal que $\exp(H) = \text{rot}(r)$. Como \mathbf{S} é H-insatisfatível, H não é um H-modelo para \mathbf{S} . Logo, pela Proposição 3.2, existe uma instância básica B de uma cláusula C em \mathbf{S} tal que $\text{comp}(B) \subset \exp(H) \subset \text{rot}(r)$. Logo, r possui um nó de fracasso N . Além disto, como $\text{comp}(B)$ é um conjunto finito de literais, N está a uma distância finita da raiz de A . Logo, r é finitamente fechado. Portanto, todo ramo de A é finitamente fechado, o que significa que A é fechada.

(b) Suponha que exista uma árvore semântica binária A fechada para \mathbf{S} . Provaremos que \mathbf{S} é insatisfatível. Seja H uma H-interpretação para \mathbf{S} . Pela Proposição 3.3, existe um ramo r de A tal que $\text{rot}(r) \subset \exp(H)$. Como A é fechada por suposição, r é finitamente fechado. Assim, existe uma instância básica B de uma cláusula de \mathbf{S} tal que $\text{comp}(B) \subset \text{rot}(r) \subset \exp(H)$. Logo, H não é um H-modelo para \mathbf{S} , pela Proposição 3.2. Assim, não existe um H-modelo para \mathbf{S} , ou seja, \mathbf{S} é H-insatisfatível. Pelo Lema 3.3, \mathbf{S} é então insatisfatível.

3.5.3 Uma Segunda Prova do Teorema de Herbrand

O Teorema de Herbrand segue diretamente do Lema 3.4, independentemente do Teorema da Compacidade.

Teorema 3.4: (Teorema de Herbrand - segunda demonstração)

Um conjunto de cláusulas \mathbf{S} é insatisfatível se e somente se existe um conjunto finito de instâncias básicas das cláusulas de \mathbf{S} que é insatisfatível.

Demonstração

(\leftarrow) Segue como na demonstração do Teorema 3.3 (que é, nesta parte, independente do Teorema da Compacidade).

(\rightarrow) Suponha que \mathbf{S} seja insatisfatível. Mostraremos que existe um conjunto finito de instâncias básicas de \mathbf{S} que é insatisfatível. Como \mathbf{S} é insatisfatível, pelo Lema 3.3, \mathbf{S} é H-insatisfatível. Seja A uma árvore semântica binária completa para \mathbf{S} . Logo, pelo Lema 3.4, A é fechada, o que significa que todo ramo r é finitamente fechado. Como A é binária e todo nó de fracasso está a uma distância finita da raiz, há um número finito de nós de fracasso, digamos, N_1, \dots, N_k . Para cada N_i seja \mathbf{B}_i a instância básica de uma cláusula em \mathbf{S} tal que $\text{comp}(\mathbf{B}_i) \subset \text{rot}(N_i)$ (\mathbf{B}_i existe pois N_i é um nó de fracasso). Seja $\mathbf{B} = \{\mathbf{B}_1, \dots, \mathbf{B}_k\}$. Mostraremos que \mathbf{B} é H-insatisfatível. Seja H uma H-interpretação de \mathbf{B} . Como \mathbf{B} é um conjunto de instâncias básicas de \mathbf{S} , H também é uma H-interpretação para \mathbf{S} . Pela Proposição 3.3, existe um ramo r de A tal que $\text{rot}(r) \subset \text{exp}(H)$. Por construção de \mathbf{B} , existe então $\mathbf{B} \in \mathbf{B}$ tal que $\text{comp}(\mathbf{B}) \subset \text{rot}(r) \subset \text{exp}(H)$. Logo, pela Proposição 3.2, H não é um H-modelo de \mathbf{B} . Assim, para toda H-interpretação H de \mathbf{B} , H não é um H-modelo de \mathbf{B} , ou seja, \mathbf{B} é H-insatisfatível. Mas, novamente pelo Lema 3.3, temos finalmente que \mathbf{B} é insatisfatível.

3.6 PROCEDIMENTOS DE REFUTAÇÃO BASEADOS NO TEOREMA DE HERBRAND

Os resultados deste capítulo devem ser entendidos como uma investigação sobre o problema da insatisfatibilidade para Lógica de Primeira Ordem pois contribuem decisivamente para a construção de procedimentos de decisão parcial para este problema, isto é, para a construção de procedimentos que recebem como entrada conjuntos de fórmulas e param com SIM, se o conjunto for insatisfatível, e podem não parar ou parar com NÃO, se o conjunto for satisfatível. Tais procedimentos são usualmente chamados de *procedimentos de refutação*.

A dificuldade do problema da insatisfatibilidade origina-se principalmente na complexidade da sintaxe das linguagens de primeira ordem e na grande liberdade de escolha do significado dos símbolos, implícita no conceito de estrutura. Os resultados deste capítulo contornam estes pontos através de

uma cadeia de reduções de um problema de decisão a um outro mais simples. Todos os problemas de decisão envolvidos são variações do problema da insatisfatibilidade, restrito ou a uma classe de fórmulas, ou a uma classe de estruturas, ou restrito de ambas as formas.

Os resultados das seções 3.2 e 3.3 permitem reduzir inicialmente o problema da insatisfatibilidade geral ao problema da insatisfatibilidade para conjuntos de cláusulas. Em seguida, o resultado da seção 3.4 permite trabalhar apenas com uma classe de estruturas com características muito especiais, as estruturas de Herbrand. Finalmente, o Teorema de Herbrand, apresentado nas seções 3.4 e 3.5, permite reduzir o problema da insatisfatibilidade para conjuntos de cláusulas ao problema de gerar um conjunto finito e insatisfatível de instâncias básicas das cláusulas. Tal conjunto de instâncias sempre existirá se o conjunto inicial de cláusulas for realmente insatisfatível, mas poderá não existir em caso contrário.

Em linhas gerais, o procedimento de refutação sugerido pelo Teorema de Herbrand tem a seguinte forma:

Procedimento de Herbrand

1. dado um conjunto **S** de cláusulas, gere sistematicamente todos os conjuntos finitos de instâncias básicas de cláusulas em **S**;
 2. para cada conjunto **B** gerado, teste se **B** é insatisfatível;
 3. pare com SIM, se **B** for insatisfatível, e pare com NÃO, se não houver novos conjuntos a gerar.
-

Note que este procedimento:

- sempre pára com SIM quando **S** for realmente insatisfatível;
- nunca pára quando **S** for satisfatível e existir um conjunto infinito de instâncias básicas de cláusulas de **S**;
- sempre pára com NÃO quando **S** for satisfatível mas o conjunto de instâncias básicas de cláusulas de **S** é finito;

Este procedimento é então um procedimento de decisão parcial para o problema da insatisfatibilidade de conjuntos de cláusulas. Mais ainda,

pela observação acima, ele é um procedimento de decisão para o problema da insatisfatibilidade de conjuntos de cláusulas cujo conjunto de instâncias básicas é finito. Note ainda que o conjunto de instâncias básicas de um conjunto finito **S** de cláusulas será finito se e somente se o universo de Herbrand para **S** for finito.

Apresentaremos a seguir, a guisa de exemplo, uma aplicação destes conceitos a um problema já resolvido através do sistema axiomático da Seção 2.6.

Exemplo 3.9:

Seja **P** a formalização de um estado do dicionário apresentada no Exemplo 3.4:

1. *programa(a,fortran)*
2. *programa(b,pascal)*
3. *programa(c,fortran)*
4. *arquivo(d,sequencial)*
5. *arquivo(e,direto)*
6. *chama(a,b)*
7. *chama(a,c)*
8. *usa(a,d)*
9. *usa(b,e)*
10. $\neg \text{chama}(x,y) \text{ depende}(x,y)$
11. $\neg \text{usa}(x,y) \text{ depende}(x,y)$
12. $\neg \text{depende}(x,z) \neg \text{depende}(z,y) \text{ depende}(x,y)$

Seja **Q** a seguinte cláusula:

13. $\neg \text{depende}(a,e)$

O universo de Herbrand para **S** = **P** U {**Q**} será então:

$$UH[\mathbf{S}] = \{a, \dots, e, \text{fortran}, \dots, \text{direto}\}$$

Desejamos estabelecer que **S** é insatisfatível utilizando o Procedimento de Herbrand, o que consiste essencialmente em enumerar sistematicamente todos os conjuntos finitos de instâncias básicas de cláusulas em **S**.

Suponha, por exemplo, que o seguinte conjunto **B** de instâncias seja eventualmente gerado:

- a. $chama(a,b)$
- b. $usa(b,e)$
- c. $\neg chama(a,b) \ depende(a,b)$
- d. $\neg usa(b,e) \ depende(b,e)$
- e. $\neg depende(a,b) \ \neg depende(b,e) \ depende(a,e)$
- f. $\neg depende(a,e)$

Note que estas cláusulas podem ser traduzidas para um conjunto de fórmulas da Lógica Sentencial associando-se os símbolos proposicionais A, B, C, D e E aos literais " $chama(a,b)$ ", " $usa(b,e)$ ", " $depende(a,b)$ ", " $depende(b,e)$ ", " $depende(a,e)$ ", respectivamente. O conjunto resultante, **B'**, será:

- a'. A
- b'. B
- c'. $\neg A \vee C$
- d'. $\neg B \vee D$
- e'. $\neg C \vee \neg D \vee E$
- f'. $\neg E$

Utilizando o método da tabela-verdade (ou simplesmente por inspeção) podemos estabelecer facilmente que **B'** e, logo, **B** são insatisfatóveis. Portanto, quando o conjunto **B**, por exemplo, for gerado, o Procedimento de Herbrand parará com SIM (outros conjuntos insatisfatóveis de instâncias básicas de **S** naturalmente existem). Note porém que uma enumeração sistemática dos conjuntos de instâncias básicas de cláusulas de **S** poderia levar muitos passos até gerar um conjunto como o exibido acima.

Recorde que $rot(N)$ denota o conjunto de todos os literais que rotulam arestas no caminho da raíz ao nó N . O Lema 3.4 sugere ainda um segundo procedimento de refutação para conjuntos de cláusulas:

Procedimento de Herbrand - II

1. seja \mathbf{S} o conjunto de cláusulas que se deseja testar para insatisfatibilidade e seja B_1, B_2, \dots uma enumeração para a base de Herbrand de \mathbf{S} ;
 2. inicie uma árvore semântica binária para \mathbf{S} apenas com a raiz (considerada de nível 1) e faça $i = 1$;
 3. repita enquanto a base de Herbrand contiver i elementos ou mais e alguma folha de nível i não for um nó de fracasso:
 - a. para cada folha N de nível i que não for um nó de fracasso faça:
 - 1) adicione o filho à esquerda NE e o filho à direita ND de N (ambos terão nível $i + 1$) rotulando as arestas (N, NE) e (N, ND) com B_i e $\neg B_i$ respectivamente;
 - 2) se $rot(NE)$ (resp. $rot(ND)$) contiver $comp(B)$, onde B é uma instância básica de alguma cláusula em \mathbf{S} , marque NE (resp., ND) como um nó de fracasso.
 - b. faça $i := i + 1$;
 4. pare com:
 - SIM, se todas as folhas da árvore forem nós de fracasso;
 - NÃO, em caso contrário;
-

Note que este procedimento:

- sempre pára com SIM quando \mathbf{S} for realmente insatisfável;
- nunca pára quando \mathbf{S} for satisfável e a base de Herbrand de \mathbf{S} for infinita;
- sempre pára com NÃO quando \mathbf{S} for satisfável e a base de Herbrand de \mathbf{S} for finita;

Portanto ele tem um comportamento semelhante ao procedimento anteriormente apresentado.

Para finalizar esta seção, enfatizamos que ambos os procedimentos de refutação esboçados acima, tal como os procedimentos de prova baseados em sistemas axiomáticos, naturalmente não evitam as limitações impostas pelo Teorema de Church e suas consequências. Além disto, tanto um enfoque quanto o outro geram uma explosão combinatorial difícil de controlar, mesmo em casos simples. Porém, desenvolvimentos mais recentes, discutidos nos capítulos seguintes, mostraram ser a construção de procedimentos de refutação muito mais promissora do que a implementação de provadores automáticos de teoremas baseados em sistemas axiomáticos.

NOTAS BIBLIOGRÁFICAS

Este capítulo baseia-se em dois textos clássicos sobre prova automática de teoremas, os textos de Chang e Lee [1973, Cap. 4] e de Loveland [1978, Cap. 1.6]. Chang e Keisler [1973] contém um tratamento rigoroso do processo de eliminação de quantificadores. Material adicional sobre o Teorema de Herbrand, principalmente as suas consequências para o estudo da decidibilidade de sub-classes de Lógica de Primeira Ordem, pode ser encontrado em Dreben e Goldfarb [1979] e em Lewis [1979]. Experimentos pioneiros com procedimentos de refutação baseados no teorema de Herbrand são descritos, por exemplo, em Gilmore [1960], Davis e Putnam [1960] e Prawitz [1960].

PARTE II - PROVA AUTOMÁTICA DE TEOREMAS

CAPÍTULO 4: RESOLUÇÃO

Este capítulo apresenta o sistema formal da resolução e o processo de unificação, que são a base da maioria dos sistemas para Programação em Lógica. A seção 4.2 é introdutória e explica, através de exemplos, como o sistema formal da resolução opera. A seção 4.3 define as noções de unificador e unificador mais geral. As seções 4.4 e 4.5 descrevem, respectivamente, o algoritmo de unificação e uma prova da sua correção. Finalmente, as seções 4.6 e 4.7 apresentam o sistema formal da resolução e uma prova da sua correção e completude.

As seções recomendadas para cada nível de leitura são:

nível introdutório: 4.2 e 4.6

nível intermediário: 4.2, 4.3, 4.4 e 4.6

4.1 INTRODUÇÃO

Este capítulo apresenta o sistema formal da resolução e o processo de unificação, que são a base da maioria dos sistemas para Programação em Lógica.

A idéia de resolução é devida a J.A. Robinson e originou-se de investigações no início da década de 60 sobre como melhorar o desempenho de procedimentos de refutação baseados no Teorema de

Herbrand. Estes procedimentos seguiam a linha do procedimento básico descrito na seção 3.6 e, embora ainda bastante ineficientes, continham alguns melhoramentos importantes para conter a geração de instâncias básicas. Em particular, os estudos de Prawitz sobre tais procedimentos reviveram a idéia de unificação. Ele observou que bastava gerar as instâncias que contivessem literais complementares que seriam então cancelados. Posto de outra forma, bastava criar instâncias de cláusulas que contivessem literais de sinais contrários e cujos átomos pudessem se tornar idênticos através de substituições. Robinson incorporou o conceito de unificação diretamente ao método de refutação, criando o que veio a ser chamado de resolução. Robinson também observou que não era necessário gerar instâncias básicas, mas bastava obter a substituição mais geral capaz de gerar literais complementares.

Finalmente observamos que a igualdade requer um tratamento especial não apresentado neste capítulo. Portanto, apenas linguagens de primeira ordem sem o símbolo de igualdade serão adotadas.

4.2 O QUE É RESOLUÇÃO

O sistema formal da resolução trabalha exclusivamente com cláusulas e contém apenas uma regra de inferência, chamada de regra da resolução, que gera uma nova cláusula a partir de duas outras. Dado um conjunto **S** de cláusulas e uma cláusula **C**, uma dedução de **C** a partir de **S** neste sistema formal consiste de uma seqüência de cláusulas terminando em **C** e gerada aplicando-se repetidamente a regra da resolução. Uma refutação a partir de **S** é uma dedução da cláusula vazia a partir de **S**. A regra da resolução é definida de tal forma que **S** é insatisfatível se e somente se existe uma refutação a partir de **S**.

Convém recordar alguns pontos antes de prosseguir. Na definição da regra da resolução, trataremos uma cláusula não-vazia " $L_1 \dots L_n$ " como o conjunto finito $\{L_1, \dots, L_n\}$ e a cláusula vazia " \square " como o conjunto vazio. Assim, utilizaremos as operações usuais de teoria dos conjuntos para definir novas cláusulas a partir de outras. Por exemplo, se " $L M N$ " e " $N P$ " são cláusulas, a expressão " $(L M N) \cup (N P)$ " denota a cláusula " $L M N P$ " (a ordem dos literais no resultado é irrelevante em face da semântica das cláusulas).

Uma cláusula **A** é uma instância de **B** se e somente se existir uma substituição $\beta = \{x_1/t_1, \dots, x_n/t_n\}$ de variáveis por termos tal que **A** é obtida

substituindo-se simultaneamente x_i por t_i em B , para $i=1,\dots,n$. Neste capítulo usaremos $B\beta$ para denotar o resultado da substituição.

Letras maiúsculas do meio do alfabeto (L, M, N, \dots) indicarão literais, letras maiúsculas do começo do alfabeto (A, B, C, \dots) indicarão cláusulas, e letras maiúsculas em negrito do meio para o fim do alfabeto ($\mathbf{S}, \mathbf{R}, \dots$) denotarão conjuntos arbitrários de cláusulas, finitos ou não.

A regra da resolução combina:

- uma adaptação para cláusulas da regra Modus Ponens (ver seção 2.6)
- um processo de "unificação" de literais de duas cláusulas
- um processo de "unificação" de literais de uma mesma cláusula

Discutiremos cada um destes pontos em separado para depois reuní-los no enunciado da regra de resolução. A cada passo da explicação corresponderá uma regra de inferência preliminar à regra da resolução, porém o objetivo será sempre o de tentar obter a cláusula vazia a partir do conjunto de cláusulas dado. Todas as regras serão naturalmente corretas no sentido de que os antecedentes de uma regra implicam logicamente o seu consequente.

Para o primeiro passo, recorde que a regra Modus Ponens ditava que de P e de $P \rightarrow Q$ podemos derivar Q ou, equivalentemente, de P e de $\neg P \vee Q$ podemos derivar Q . Uma adaptação imediata de Modus Ponens para cláusulas seria então:

R1: se A' possui um literal L e A'' possui um literal $\neg L$,
derive $A = (A' - L) \cup (A'' - \neg L)$

Ou seja, A é uma lista de literais contendo, em qualquer ordem, os literais em A' e A'' , exceto L e $\neg L$.

Para ilustrar o uso da regra R1, seja R o seguinte conjunto de cláusulas:

1. $chama(a, z)$
2. $\neg chama(a, z) \wedge depende(a, z)$
3. $\neg depende(a, z)$

A partir de R , podemos então derivar as seguintes cláusulas usando R1:

4. *depende(a,z)* . aplicando R1 a (1) e (2)
5. \square . aplicando R1 a (3) e (4)

A derivação da cláusula vazia em (5) é então suficiente para estabelecer que **R** é insatisfatível (por quê?).

Em detalhe, a cláusula em (4) segue das cláusulas em (1) e (2) por R1 identificando-se no enunciado da regra:

- A' com a cláusula "*chama(a,z)*"
- A'' com a cláusula " $\neg chama(a,z)$ *depende(a,z)*"
- L com o literal "*chama(a,z)*"
- $A = (A' - L) \cup (A'' - \neg L) = depende(a,z)$

A dedução da cláusula em (5) segue de forma semelhante. Apenas observe que ela reflete claramente a contradição que resulta da derivação de "*depende(a,z)*" na linha (4), em presença da suposição " $\neg depende(a,z)$ " da linha (3).

Embora a regra R1 só se aplique quando as cláusulas A' e A'' possuem literais complementares, em muitas situações é possível obter instâncias de A' e A'' às quais R1 se aplique. Por exemplo, seja **S** o seguinte conjunto de cláusulas:

1. *chama(a,z)*
2. $\neg chama(x,y)$ *depende(x,y)*
3. $\neg depende(a,z)$

R1 não se aplica a nenhum par de cláusulas em **S**. Por outro lado, aplicando a substituição $\beta = \{x/a, y/z\}$ à cláusula em (2), obtemos:

4. $\neg chama(a,z)$ *depende(a,z)*

Logo, obtemos:

5. *depende(a,z)* . aplicando R1 a (1) e (4)
6. \square . aplicando R1 a (3) e (5)

A regra R2 enunciada abaixo combina, em um só passo, a derivação de cláusulas intermediárias, como a cláusula (4) acima, com a regra R1:

R2: se A' possui um literal L' e A'' possui um literal $\neg L''$ e existe uma substituição β tal que $L'\beta = L''\beta$, derive $A = (A'\beta - L'\beta) \cup (A''\beta - \neg L''\beta)$

Ou seja, A é uma lista de literais formada tomando-se, em qualquer ordem, os literais em A' e A'' , exceto L' e $\neg L''$, e aplicando a substituição β a todos os literais restantes.

No exemplo anterior, podemos então derivar diretamente a cláusula em (5) aplicando R2 a (1) e (2), onde L' é o literal "*chama(a,z)*", $\neg L''$ é o literal " $\neg chama(x,y)$ " e $\beta = \{x/a, y/z\}$.

O processo de tornar idênticos os literais em um conjunto E através de uma substituição de variáveis por termos é chamado de *unificação* e a substituição é chamada de um *unificador* de E . Um *unificador mais geral* é aquele que, intuitivamente, especifica as substituições mais simples possíveis. O processo de unificação deverá então utilizar sempre um unificador mais geral para não bloquear outras unificações. Por exemplo, $\beta = \{x/f(a), y/a\}$ e $\theta = \{x/f(y)\}$ unificam $E = \{p(x), p(f(y))\}$ pois $E\beta = \{p(f(a))\}$ e $E\theta = \{p(f(y))\}$. Porém θ é um unificador mais geral do que β . A substituição θ é então preferível a β pois, por exemplo, o literal " $p(f(b))$ " é unificável com o literal em $E\theta$, mas não é unificável com o literal em $E\beta$.

No enunciado da regra R2 utilizamos unificação para formar um par de literais complementares oriundos de cláusulas diferentes. Porém, também podemos usar unificação para tornar idênticos literais de uma mesma cláusula. Este processo é igualmente importante como ilustra o seguinte exemplo.

Seja T o seguinte conjunto de cláusulas:

1. $\neg p(x) \neg p(a)$
2. $p(y) p(z)$

Note que T é insatisfatível pois $\neg p(a)$ e $p(a)$ são instâncias básicas de (1) e de (2), respectivamente, e estas instâncias levam claramente a uma contradição. Porém, a aplicação repetida da regra R2 falhará em determinar que T é insatisfatível, pois todas as cláusulas deduzidas a partir de T via R2 terão dois literais. Logo, a cláusula vazia nunca será gerada.

Porém, se gerarmos novas cláusulas apenas pela unificação de literais de uma mesma cláusula, então poderemos deduzir a cláusula vazia a partir de \top :

1. $\neg p(x) \neg p(a)$
2. $p(y) p(z)$
3. $\neg p(a)$. gerada de (1) pela substituição $\{x/a\}$
4. $p(z)$. gerada de (2) pela substituição $\{y/z\}$
5. \square . aplicando R2 a (3) e (4)

Em geral, dizemos que uma cláusula B é um *fator* de uma cláusula A se e somente se existe um conjunto L de literais de A e existe um unificador mais geral β para L tal que $B = A\beta$. Note que uma cláusula A é um fator dela mesma. O processo de obter fatores de cláusulas é chamado de *fatoração*.

Em algumas abordagens, o processo de fatoração é enunciado como uma regra de inferência separada. Porém, manteremos aqui uma só regra incorporando fatoração a R2 para obter o enunciado final da regra da resolução:

RE: se B' e B'' são fatores de cláusulas A' e A'' tais que
 B' possui um literal L' e B'' um literal $\neg L''$ e
existe um unificador mais geral β para L' e L'' ,
derive $A = (B'\beta - L'\beta) \cup (B''\beta - \neg L''\beta)$

Neste caso dizemos que a cláusula A é um *resolvente* de A' e A'' .

Dois comentários cabem aqui. Em primeiro lugar, note que B' e B'' naturalmente poderão coincidir com A' e A'' pois uma cláusula é fator dela mesma. Em segundo lugar, observe que β poderá não existir simplesmente porque A' e A'' possuem variáveis em comum. Portanto, deve-se sempre tomar o cuidado de renomear apropriadamente as variáveis de uma das cláusulas para evitar coincidências. Por exemplo, seja A' a cláusula " $p(x)$ " e A'' a cláusula " $p(f(x))$ ". Note que $\{p(x), p(f(x))\}$ não é unificável pois não é possível tornar $p(x)$ igual a $p(f(x))$ substituindo x pelo mesmo termo em ambos os literais. Por exemplo, se substituirmos x por $f(x)$ obteremos $p(f(x))$ e $p(f(f(x)))$. Logo não há resolventes diretamente de A' e A'' . Mas se renomearmos x por y em A'' teremos que \square é um resolvente de A' e $A''\theta$.

Note que, usando apenas RE, podemos deduzir a cláusula vazia diretamente a partir de T:

1. $\neg p(x) \neg p(a)$
2. $p(y) p(z)$
3. \square . aplicando RE a (1) e (2)

Para deduzir a cláusula em (3), identificamos no enunciado da regra RE:

- A' com a cláusula " $p(y) p(z)$ "
- A'' com a cláusula " $\neg p(x) \neg p(a)$ "
- B' com o fator " $p(z)$ " de A' (via a substituição $\{y/z\}$)
- B'' com o fator " $\neg p(a)$ " de A'' (via a substituição $\{x/a\}$)
- β com $\{z/a\}$
- L' β com o literal " $p(a)$ "

Segue-se um exemplo um pouco mais complexo. Seja P o seguinte conjunto de cláusulas:

1. $chama(a,b)$
2. $usa(b,e)$
3. $\neg chama(x,y) depende(x,y)$
4. $\neg usa(x,y) depende(x,y)$
5. $\neg depende(x,z) \neg depende(z,y) depende(x,y)$
6. $\neg depende(a,e)$

Aplicando a regra RE obtemos

7. $depende(a,b)$.de 1, 3 com $\beta = \{x/a, y/b\}$
8. $depende(b,e)$.de 2, 4 com $\beta = \{x/b, y/e\}$
9. $\neg depende(b,y) depende(a,y)$.de 5, 7 com $\beta = \{x/a, z/b\}$
10. $depende(a,e)$.de 8, 9 com $\beta = \{y/e\}$
11. \square .de 6, 10

Em resumo, o sistema formal da resolução trabalha apenas com cláusulas, que são objetos com uma sintaxe bem simples, e possui apenas uma regra de inferência, a regra da resolução. Um procedimento de refutação baseado em resolução é então um procedimento que, dado um conjunto qualquer de cláusulas S, procura sistematicamente derivar a cláusula vazia utilizando apenas a regra da resolução e tendo como ponto de partida as cláusulas em S. A construção de tais procedimentos requer, porém,

métodos especiais para tentar contornar a explosão combinatorial gerada pela liberdade de escolha de cláusulas, fatores e literais. Alguns destes métodos serão explorados no próximo capítulo.

4.3 UNIFICAÇÃO

Esta seção inicia, com as noções de substituição e unificação, a formalização dos conceitos introduzidos na seção 4.2. Todas as definições aqui apresentadas dependem implicitamente da escolha de uma linguagem de primeira ordem, que é deixada em aberto.

A noção de substituição é uma rigorização daquela já introduzida no final da seção 2.2:

Definição 4.1:

- (a) Um par (x,t) é uma *substituição simples* (lê-se "x substituído por t") se e somente se x é uma variável e t é um termo.
- (b) Um conjunto finito β de substituições simples é uma *substituição* se e somente se duas substituições simples em β não coincidem no primeiro elemento.
- (c) Uma substituição β é uma *substituição básica* se e somente se, para todo (x,t) em β , t é um termo sem ocorrências de variáveis.
- (d) β é a *substituição vazia* se e somente se β for o conjunto vazio.
- (e) Uma substituição β é uma *renomeação de variáveis* ou, simplesmente, uma *renomeação*, se e somente se cada par (x,t) em β for tal que t é uma variável e não existirem dois pares (x,u) e (y,v) em β tais que $x \neq y$ e $u = v$.

A expressão " x/t " denotará uma substituição simples (x,t) . Letras gregas denotarão substituições e, em especial, " ε " denotará a substituição vazia.

De acordo com estas definições, $\beta = \{x/a, y/b, z/y\}$ e $\theta = \{x/f(y), y/z\}$ são substituições. Porém, $\phi = \{x/f(x), x/a\}$ não é uma substituição pois possui dois pares com o mesmo primeiro elemento.

Uma *expressão* é qualquer seqüência de símbolos de um alfabeto de primeira ordem, e uma *expressão simples* é qualquer literal ou termo sobre o alfabeto.

Definição 4.2:

- (a) Sejam E uma expressão e $\beta = \{x_1/t_1, \dots, x_n/t_n\}$ uma substituição. A *instanciação* de E por β ou, simplesmente, uma *instância* de E é a expressão $E\beta$ obtida substituindo-se simultaneamente em E cada ocorrência de x_i por t_i , $i = 1, \dots, n$.
- (b) Sejam E um conjunto de expressões e $\beta = \{x_1/t_1, \dots, x_n/t_n\}$ uma substituição. O conjunto de expressões $E\beta = \{E\beta / E \in E\}$ é a *instanciação* de E por β ou, simplesmente, uma *instância* de E .
- (c) Seja C uma cláusula e β uma substituição. A *instanciação* de C por β , denotada por $C\beta$, é a cláusula obtida instanciando-se C por β e eliminando-se as ocorrências repetidas do mesmo literal, exceto a ocorrência mais à esquerda.

Por exemplo, se $E = \{p(x,y), \neg q(f(x))\}$ e $\theta = \{x/f(y), y/z\}$ então $E\theta = \{p(f(y),z), \neg q(f(f(y)))\}$ é a instância de E por θ .

Quando necessário, envolveremos um conjunto de expressões ou uma expressão entre parênteses. Assim, preferiremos $(E_1)\beta$ a $E_1\beta$.

Definição 4.3:

A *composição* de substituições é a função, denotada por “ \circ ”, que mapeia pares de substituições em uma substituição e é definida da seguinte forma:

Para todo par de substituições

$$\begin{aligned}\beta &= \{x_1/t_1, \dots, x_n/t_n, y_1/s_1, \dots, y_k/s_k\} \\ \theta &= \{y_1/r_1, \dots, y_k/r_k, z_1/q_1, \dots, z_m/q_m\}\end{aligned}$$

onde $x_1, \dots, x_n, y_1, \dots, y_k, z_1, \dots, z_m$ são variáveis distintas, a composição de β com θ será a substituição:

$$\beta \circ \theta = \{x_1/(t_1)\theta, \dots, x_n/(t_n)\theta, y_1/(s_1)\theta, \dots, y_k/(s_k)\theta, z_1/q_1, \dots, z_m/q_m\}$$

Por exemplo, a composição de $\beta = \{x/f(y), y/z\}$ com $\theta = \{x/a, y/b, z/y\}$ é a substituição $\beta \circ \theta = \{x/f(b), y/y, z/y\}$ obtida aplicando-se θ aos termos das substituições simples em β , formando o conjunto $\{x/f(b), y/y\}$, e acrescentando-se a este conjunto a única substituição simples de θ , “ z/y ”, cujo primeiro elemento não coincide com o primeiro elemento de uma substituição simples em β .

As substituições possuem as seguintes propriedades:

Proposição 4.1:

Sejam θ , β e φ substituições e ε a substituição vazia.

- (a) $\theta^\circ \varepsilon = \varepsilon^\theta = \theta$
- (b) $(E\theta)\beta = E(\theta^\circ \beta)$, para toda expressão E
- (c) Se $E\theta = E\beta$ para toda variável E , então $\theta = \beta$
- (d) $(\theta^\circ \beta)^\circ \varphi = \theta^\circ (\beta^\circ \varphi)$

Demonstração

Provaremos apenas (b). Sejam as substituições

- (1) $\theta = \{x_1/t_1, \dots, x_n/t_n, y_1/s_1, \dots, y_k/s_k\}$
- (2) $\beta = \{y_1/r_1, \dots, y_k/r_k, z_1/q_1, \dots, z_m/q_m\}$.

Seja a expressão

$$(3) E[x_1, \dots, x_n, y_1, \dots, y_k, z_1, \dots, z_m]$$

Então temos:

- (4) $E\theta = E[t_1, \dots, t_n, s_1, \dots, s_k, z_1, \dots, z_m]$
- (5) $(E\theta)\beta = E[(t_1)\beta, \dots, (t_n)\beta, (s_1)\beta, \dots, (s_k)\beta, q_1, \dots, q_m]$.

Mas o lado direito de (5) é $E(\theta^\circ \beta)$, por definição de composição.

Note que, por (a), a substituição vazia é a identidade à direita e à esquerda da operação de composição e, por (d), a operação de composição é associativa.

Definição 4.4:

Seja $E = \{E_1, \dots, E_n\}$ um conjunto de expressões simples e β uma substituição.

- (a) β é um *unificador* de E se e somente se $(E_1)\beta = \dots = (E_n)\beta$.
- (b) β é um *unificador mais geral* (ou, abreviadamente, um *u.m.g*) de E se e somente se β é um unificador de E e, para todo unificador θ de E , existe uma substituição φ tal que $\theta = \beta^\circ \varphi$.

- (c) O conjunto **E** é *unificável* se e somente se existe um unificador para **E**.

Por exemplo, se $E = \{p(a, f(x)), p(y, z)\}$ então $\theta = \{y/a, x/b, z/f(b)\}$ é um unificador de **E** pois $E\theta = \{p(a, f(b))\}$. Mas $\beta = \{y/a, z/f(x)\}$ também é um unificador de **E** pois $E\beta = \{p(a, f(x))\}$. Porém, β é "mais geral" do que θ pois evita a substituição de x por b . De fato, $\theta = \beta^{\circ}\varphi$, onde $\varphi = \{x/b\}$. Na verdade, β é um u.m.g de **E**.

Já o conjunto $F = \{p(a, x), p(b, y)\}$ não é unificável, essencialmente porque os seus dois literais diferem em duas constantes e não podemos substituir uma constante por outra no processo de unificação.

Por fim, observamos que o unificador mais geral para um dado conjunto de expressões simples **E** não é único. Porém, o seguinte resultado mostra que dois unificadores mais gerais diferem apenas por renomeações de variáveis.

Proposição 4.2:

Se β e β' são unificadores mais gerais de um conjunto de expressões simples **E** então existe uma renomeação θ tal que $\beta = \beta'^{\circ}\theta$.

Demonstração

Sejam β e β' unificadores mais gerais de um conjunto de expressões **E**. Então, por definição, existem substituições φ e φ' tais que $\beta' = \beta^{\circ}\varphi$ e $\beta = \beta'^{\circ}\varphi'$. Logo, $\beta' = (\beta'^{\circ}\varphi')^{\circ}\varphi$, o que é equivalente a $\beta' = \beta'^{\circ}(\varphi'^{\circ}\varphi)$, que por sua vez implica em que $\varphi'^{\circ}\varphi = \lambda$, onde λ é uma coleção de substituições simples da forma x/x . Se φ' for a substituição vazia, então $\beta = \beta'$ e podemos tomar θ também como a substituição vazia. Suponha que φ' não seja a substituição vazia. Seja x/t uma substituição simples de φ' . Então, por definição de composição, $\varphi'^{\circ}\varphi$ possui uma substituição simples da forma $(x/t)\varphi$. Mas, como $\varphi'^{\circ}\varphi = \lambda$, $t\varphi$ deve ser x , pois λ só possui pares da forma x/x . Mas $t\varphi$ só poderá ser igual a x se t for uma variável. Logo, para toda substituição simples x/t em φ' , t é uma variável. Ou seja, φ' é uma renomeação. Logo, podemos tomar $\theta = \varphi'$.

4.4 O ALGORITMO DE UNIFICAÇÃO

Esta seção representa um passo fundamental para a implementação de sistemas de refutação baseados em resolução pois apresenta um algoritmo para determinar um u.m.g, caso exista, para um conjunto de expressões simples. A prova da correção do algoritmo aparecerá na seção 4.5.

O algoritmo generaliza o processo exemplificado a seguir.

Exemplo 4.1:

Neste exemplo determinaremos de forma sistemática um u.m.g para o seguinte conjunto de literais:

$$(1) \mathbf{E} = \{p(a, f(x)), p(y, z)\}$$

O primeiro passo consiste em localizar os termos mais à esquerda em que os literais diferem da seguinte forma:

- Considere cada um dos literais em \mathbf{E} simplesmente como uma cadeia de símbolos. Sob esta perspectiva, compare os literais para localizar a posição i mais à esquerda em que diferem.
- Extraia de cada um dos literais os termos que começam na posição i , formando o *conjunto de discórdia* dos literais.

No caso, os literais em \mathbf{E} diferem na terceira posição. Logo, o conjunto de discórdia de \mathbf{E} é:

$$(2) \mathbf{D} = \{a, y\}$$

Como \mathbf{D} contém uma variável "y" e um termo "a" onde "y" não ocorre, podemos construir a substituição $\theta = \{y/a\}$ que aplicada a \mathbf{E} torna os seus literais "mais semelhantes":

$$(3) \mathbf{E}\theta = \{p(a, f(x)), p(a, z)\}$$

Note que, por força da substituição e por "y" não ocorrer em "a", os dois literais em $\mathbf{E}\theta$ coincidem no primeiro termo.

Repita agora o processo sobre $\mathbf{E}\theta$. O novo conjunto de discordia obtido será:

$$(4) D' = \{f(x), z\}$$

Novamente, como D' contém uma variável "z" e um termo "f(x)" onde "z" não ocorre, podemos construir a substituição $\beta = \{z/f(x)\}$ que aplicada a E^θ torna os seus literais idênticos (por força da substituição e por "z" não ocorrer em "f(x)":)

$$(3) (E^\theta)\beta = \{p(a, f(x))\}$$

O u.m.g de E será então a composição de $\theta = \{y/a\}$ com $\beta = \{z/f(x)\}$:

$$(4) \theta^\circ\beta = \{y/a, z/f(x)\}$$

Dado um conjunto de expressões simples, o processo exemplificado acima consiste essencialmente em tornar idênticas as expressões, caminhando da esquerda para a direita, através de substituições sucessivas. A cada passo, localizam-se os termos mais à esquerda em que as expressões diferem, construindo-se o conjunto de discórdia. Em seguida, extrai-se do conjunto de discórdia D , se for possível, uma variável x e um termo t em que x não ocorre. A substituição simples x/t é então aplicada tanto ao conjunto de expressões quanto à substituição corrente. Se existirem x e t em D tais que x não ocorre em t , diremos que D satisfaz o teste de ocorrência. Este teste é importante pois, se não for respeitado, o algoritmo não terminará corretamente, conforme veremos no final desta seção.

Mais precisamente, temos:

Definição 4.5:

Um conjunto de termos D é o *conjunto de discordia* de um conjunto de expressões simples $E = \{E_1, \dots, E_n\}$ se e somente se

- (i) $D = \emptyset$, se $n = 1$;
- (ii) $D = \{t_1, \dots, t_n\}$, se $n > 1$ e todas as expressões em E são idênticas até o i -ésimo símbolo, exclusivo, e t_j é o termo ocorrendo em E_j que começa no i -ésimo símbolo, para $j = 1, \dots, n$.

Definição 4.6:

- (a) Uma substituição simples x/t *satisfaz o teste de ocorrência* se e somente se x não ocorre em t .

- (b) Um conjunto de discórdia D satisfaz o teste de ocorrência se e somente se existem uma variável x e um termo t em D tais que x não ocorre em t .

Note que um conjunto de discórdia D poderá não satisfazer o teste de ocorrência simplesmente por não conter pelo menos uma variável. Esta condição, apesar de trivial, é parte integrante da nossa definição de teste de ocorrência para conjuntos de discórdia.

Exemplo 4.2:

CONJUNTO DE EXPRESSÕES	CONJUNTO DE DISCÓRDIA
{ $p(a,x,f(g(y)))$, $p(z,h(z,w),f(w))$ }	{ a, z }
{ $p(a,x,f(g(y)))$, $p(a,h(a,w),f(w))$ }	{ $x, h(a,w)$ }
{ $p(a,h(a,w),f(g(y)))$, $p(a,h(a,w),f(w))$ }	{ $g(y), w$ }

Em cada um destes exemplos temos que o conjunto de discórdia satisfaz o teste de ocorrência.

O algoritmo para determinar um u.m.g de um conjunto de expressões simples é o seguinte (se E é um conjunto de expressões e x/t é uma substituição simples, $E\{x/t\}$ denota a instanciação de E por $\{x/t\}$):

Algoritmo 4.1: Algoritmo de Unificação

entrada: um conjunto \mathbf{E} de expressões simples

saída:

- um u.m.g β de \mathbf{E} , se \mathbf{E} for unificável
- 'NÃO', se \mathbf{E} não for unificável

begin

$\beta := \epsilon;$

$W := \mathbf{E};$

$D :=$ conjunto de discórdia de $W;$

while $|W| > 1$ e

D satisfaz o teste de ocorrência **do**

begin

 selecione uma variável x e um termo t em D

 tais que x não ocorre em t ;

$\beta := \beta^o\{x/t\};$

$W := W\{x/t\};$

$D :=$ conjunto de discórdia de $W;$

end;

if $|W| = 1$

then return β

else return 'NÃO'

end

Observe que o algoritmo de unificação apresentado não é determinístico devido ao processo de seleção da variável x e do termo t não ser determinístico.

Os exemplos a seguir ilustram como o algoritmo de unificação opera. Nestes exemplos, para $k \geq 1$, $x(k)$, $t(k)$, $\beta(k)$, $W(k)$ e $D(k)$ denotam os valores assumidos pelas variáveis de programa x , t , β , W e D ao final da k -ésima iteração e $\beta(0)$, $W(0)$ e $D(0)$ denotam os valores iniciais de β , W e D .

Exemplo 4.3:

Aplicaremos, passo a passo, o algoritmo de unificação para determinar um u.m.g do seguinte conjunto de literais:

$$E = \{p(a,x,f(g(y))), p(z,h(z,w),f(w)), p(a,h(a,g(b)),f(g(v)))\}$$

Os valores de β , W e D e a variável x e o termo t selecionados a cada passo serão:

$$\beta(0) = \varepsilon$$

$$\begin{aligned} W(0) &= \{p(a,x,f(g(y))), \\ &\quad p(z,h(z,w),f(w)), \\ &\quad p(a,h(a,g(b)),f(g(v)))\} \\ D(0) &= \{a, z\} \end{aligned}$$

$$x(1) = z$$

$$t(1) = a$$

$$\beta(1) = \{z/a\}$$

$$\begin{aligned} W(1) &= \{p(a,x,f(g(y))), \\ &\quad p(a,h(a,w),f(w)), \\ &\quad p(a,h(a,g(b)),f(g(v)))\} \\ D(1) &= \{x, h(a,w), h(a,g(b))\} \end{aligned}$$

$$x(2) = x$$

$$t(2) = h(a,w)$$

$$\beta(2) = \{z/a, x/h(a,w)\}$$

$$\begin{aligned} W(2) &= \{p(a,h(a,w),f(g(y))), \\ &\quad p(a,h(a,w),f(w)), \\ &\quad p(a,h(a,g(b)),f(g(v)))\} \\ D(2) &= \{w, g(b)\} \end{aligned}$$

$$x(3) = w$$

$$t(3) = g(b)$$

$$\beta(3) = \{z/a, x/h(a,g(b)), w/g(b)\}$$

$$\begin{aligned} W(3) &= \{p(a,h(a,g(b)),f(g(y))), \\ &\quad p(a,h(a,g(b)),f(g(b))), \\ &\quad p(a,h(a,g(b)),f(g(v)))\} \\ D(3) &= \{y, b, v\} \end{aligned}$$

$$\begin{aligned}
 x(4) &= y \\
 t(4) &= b \\
 \beta(4) &= \{z/a, x/h(a,g(b)), w/g(b), y/b\} \\
 W(4) &= \{p(a,h(a,g(b)),f(g(b))), \\
 &\quad p(a,h(a,g(b)),f(g(b))), \\
 &\quad p(a,h(a,g(b)),f(g(v)))\} \\
 D(4) &= \{b, v\}
 \end{aligned}$$

$$\begin{aligned}
 x(5) &= v \\
 t(5) &= b \\
 \beta(5) &= \{z/a, x/h(a,g(b)), w/g(b), y/b, v/b\} \\
 W(5) &= \{p(a,h(a,g(b)),f(g(b)))\} \\
 D(5) &= \emptyset
 \end{aligned}$$

O algoritmo pára então com SIM e retorna a substituição $\beta(5)$.

Exemplo 4.4:

Aplicaremos agora o algoritmo de unificação ao seguinte conjunto de literais:

$$F = \{q(f(a), g(x)), q(y, y)\}$$

Os valores de β , W e D a cada passo serão:

$$\begin{aligned}
 \beta(0) &= \epsilon \\
 W(0) &= \{q(f(a), g(x)), q(y, y)\} \\
 D(0) &= \{f(a), y\}
 \end{aligned}$$

$$\begin{aligned}
 x(1) &= y \\
 t(1) &= f(a) \\
 \beta(1) &= \{y/f(a)\} \\
 W(1) &= \{q(f(a), g(x)), q(f(a), f(a))\} \\
 D(1) &= \{g(x), f(a)\}
 \end{aligned}$$

Como $D(1)$ não satisfaz o teste de ocorrência pois não contém uma variável, o comando de repetição termina e o algoritmo pára com NÃO.

Terminaremos esta seção com algumas observações sobre a complexidade do problema de unificação, ou seja, do problema de determinar

unificadores mais gerais para conjuntos de expressões simples. Comentaremos também o papel do teste de ocorrência.

Inicialmente observe que, em termos da complexidade pessimista de tempo, o algoritmo de unificação apresentado é exponencial sobre o comprimento do conjunto E de entrada. Ou seja, existem conjuntos de expressões simples para os quais o algoritmo leva tempo exponencial para determinar um u.m.g. Tal comportamento está ligado ao teste de ocorrência, conforme ilustra o seguinte exemplo.

Exemplo 4.5:

Seja E o seguinte conjunto de literais:

$$E = \{p(x_1, \dots, x_n), p(f(a,a), f(x_1, x_1), \dots, f(x_{n-1}, x_{n-1}))\}$$

Aplicando o algoritmo de unificação a E temos:

$$\beta(0) = \epsilon$$

$$W(0) = \{p(x_1, x_2, \dots, x_n), \\ p(f(a,a), f(x_1, x_1), \dots, f(x_{n-1}, x_{n-1}))\}$$

$$D(0) = \{x_1, f(a,a)\}$$

$$\beta(1) = \{x_1/f(a,a)\}$$

$$W(1) = \{p(f(a,a), x_2, x_3, \dots, x_n), \\ p(f(a,a), f(f(a,a), f(a,a)), f(x_2, x_2), \dots, f(x_{n-1}, \dots, x_{n-1}))\}$$

$$D(1) = \{x_2, f(f(a,a), f(a,a))\}$$

$$\beta(2) = \{x_1/f(a,a), x_2/f(f(a,a), f(a,a))\}$$

$$W(2) = \{p(f(a,a), f(f(a,a), f(a,a)), x_3, \dots, x_n), \\ p(f(a,a), f(f(a,a), f(a,a)), f(f(f(a,a), f(a,a)), f(f(a,a), f(a,a))), \dots, \\ f(x_{n-1}, \dots, x_{n-1}))\}$$

$$D(2) = \{x_3, f(f(f(a,a), f(a,a)), f(f(a,a), f(a,a))))\}$$

...

Para cada $k \geq 1$, $D(k-1)$ possui então uma variável, x_k , e um termo t_k contendo $2^k - 1$ ocorrências de f . Logo, testar se $D(k-1)$ satisfaz o teste de ocorrência, ou seja, se x_k ocorre em t_k , requer tempo exponencial pois o comprimento de t_k é proporcional a $2^k - 1$.

A discussão do exemplo anterior não implica porém que a cota superior de complexidade (de tempo) para o problema de unificação seja exponencial. Na verdade, algoritmos lineares de unificação podem ser encontrados em Paterson e Wegman [1978] e em Martelli e Montanari [1982]. Dwork, Kanellakis e Mitchell [1985] contém também resultados importantes sobre a complexidade do problema de unificação que implicam, entre outras consequências, que paralelismo não pode melhorar significativamente a performance dos melhores algoritmos seqüenciais de unificação.

Concluiremos esta seção observando que o teste de ocorrência não pode ser omitido sob pena do algoritmo de unificação não terminar, como mostra o exemplo abaixo. Em alguns refinamentos do algoritmo de unificação, a omissão do teste de ocorrência poderá mesmo levar o algoritmo a terminar com uma substituição que não é um unificador do conjunto de expressões dado como entrada.

Exemplo 4.6:

Seja $E = \{p(x), p(f(x))\}$. Aplicando o algoritmo de unificação conforme definido temos:

$$\begin{aligned}\beta(0) &= \epsilon \\ W(0) &= \{p(x), p(f(x))\} \\ D(0) &= \{x, f(x)\}\end{aligned}$$

Como $D(0)$ não satisfaz o teste de ocorrência, o algoritmo pára imediatamente com NÃO.

Suponha agora que o teste de ocorrência seja omitido. O novo algoritmo não parará então para $E = \{p(x), p(f(x))\}$. De fato, temos:

$$\begin{aligned}\beta(0) &= \epsilon \\ W(0) &= \{p(x), p(f(x))\} \\ D(0) &= \{x, f(x)\} \\ \\ \beta(1) &= \{x/f(x)\} \\ W(1) &= \{p(f(x)), p(f(f(x)))\} \\ D(1) &= \{x, f(x)\}\end{aligned}$$

$$\begin{aligned}\beta(2) &= \{x/f(x)\} \\ W(2) &= \{p(f(f(x))), p(f(f(f(x))))\} \\ D(2) &= \{x, f(x)\}\end{aligned}$$

...

A situação tornou-se então interessante. Se, por um lado, conforme visto anteriormente, o teste de ocorrência é responsável pelo comportamento exponencial do algoritmo de unificação apresentado, por outro lado, ele não pode ser omitido sob pena do algoritmo nem sempre terminar.

4.5 CORREÇÃO DO ALGORITMO DE UNIFICAÇÃO

Esta seção estabelece a correção do algoritmo de unificação, provando separadamente que ele sempre termina e que computa corretamente um u.m.g para o conjunto de expressões dado como entrada, se realmente o conjunto for unificável.

Como na seção anterior, para $k \geq 1$, $\beta(k)$, $W(k)$ e $D(k)$ denotam os valores assumidos pelas variáveis de programa β , W e D ao final da k -ésima iteração e $\beta(0)$, $W(0)$ e $D(0)$ denotam os valores iniciais de β , W e D .

Lema 4.1:

O algoritmo de unificação sempre pára em um número finito de iterações.

Demonstração

Provaremos inicialmente que, se $D(k)$ satisfaz o teste de ocorrência, então o número de variáveis ocorrendo em $W(k+1)$ é estritamente menor do que o número de variáveis ocorrendo em $W(k)$. Suponha que $D(k)$ satisfaz o teste de ocorrência. Logo existe uma variável e um termo em $D(k)$ tais que a variável não ocorre no termo. Sejam x e t a variável e o termo de $D(k)$ escolhidos na k -ésima iteração. Logo, pelo critério de seleção de x e t , x não ocorre em t . Como $W(k+1) = W(k)\{x/t\}$ e x não ocorre em t , x não ocorre em $W(k+1)$. Mas, por construção de $D(k)$, x ocorre em $W(k)$. Logo, o número de variáveis ocorrendo em $W(k+1)$ é menor do que em $W(k)$. Portanto, o algoritmo sempre termina em um número de iterações menor ou igual ao número de variáveis ocorrendo em $W(0)$.

Os resultados a seguir estabelecem que o algoritmo de unificação computa um u.m.g para o conjunto de expressões de entrada se e somente se o conjunto realmente for unificável. As duas primeiras proposições são meramente técnicas:

Proposição 4.3:

Se \mathbf{D} é o conjunto de discórdia de um conjunto \mathbf{E} de expressões simples e se θ unifica \mathbf{E} , então θ unifica \mathbf{D} .

Proposição 4.4:

Se \mathbf{D} é o conjunto de discórdia de um conjunto \mathbf{E} de expressões simples tal que \mathbf{E} é unificável e $|\mathbf{E}| > 1$, então \mathbf{D} satisfaz o teste de ocorrência.

Demonstração

Seja \mathbf{E} um conjunto de expressões simples tal que \mathbf{E} é unificável e $|\mathbf{E}| > 1$. Seja θ um unificador de \mathbf{E} . Seja \mathbf{D} o conjunto de discórdia de \mathbf{E} . Mostraremos que \mathbf{D} satisfaz o teste de ocorrência, ou seja, que \mathbf{D} tem pelo menos uma variável e um termo tais que a variável não ocorre no termo. Pelas suposições sobre \mathbf{E} e pela Proposição 4.3, temos que $|\mathbf{D}| > 1$ e θ é um unificador de \mathbf{D} . Logo, como \mathbf{D} é unificável e \mathbf{D} é o conjunto de discórdia de \mathbf{E} , deve haver pelo menos uma variável x em \mathbf{D} pois, de outra forma, o ítem (ii) da Definição 4.5 seria violado. Além disto, como $|\mathbf{D}| > 1$, deve existir um termo t em \mathbf{D} tal que t é diferente de x . Provaremos que x não ocorre em t . Suponha o contrário. Logo $x\theta$ ocorre em $t\theta$. Mas isto é impossível pois x e t são diferentes e $x\theta = t\theta$, pois θ unifica \mathbf{D} e x e t pertencem a \mathbf{D} . Logo, podemos concluir que x não ocorre em t , logo que \mathbf{D} satisfaz o teste de ocorrência.

Lema 4.2: (Lema da Unificação)

Se \mathbf{E} é um conjunto unificável de expressões simples então o algoritmo de unificação retorna um u.m.g para \mathbf{E} .

Demonstração

Seja \mathbf{E} um conjunto unificável de expressões simples. Pelo lema anterior, o algoritmo pára quando \mathbf{E} for a entrada. Suponha que o algoritmo execute K iterações, para $K \geq 0$. Provaremos inicialmente que, para todo unificador θ para \mathbf{E} , para todo $k \in [0, K]$, $I(k)$ vale, onde $I(k)$ é a conjunção das seguintes condições:

- I₁(k): $W(k) = \mathbf{E}\beta(k)$
- I₂(k): existe φ tal que $\theta = \beta(k)^o\varphi$
- I₃(k): $D(k)$ é o conjunto de discórdia de $W(k)$
- I₄(k): $D(k)$ satisfaz o teste de ocorrência ou $|W(k)| = 1$

Seja θ um unificador para \mathbf{E} . Provaremos, por indução sobre $k \in [0, K]$, que I(k) vale.

Base: Mostraremos que I(0) vale. Pela inicialização de β , W e D , temos que $\beta(0) = \varepsilon$, $W(0) = \mathbf{E}$ e $D(0)$ é o conjunto de discórdia de $W(0)$. Tome φ da condição I₂(0) como sendo θ . Logo $W(0) = \mathbf{E}\beta(0)$, $\theta = \beta(0)^o\theta$ e θ é um unificador de $W(0)$, por suposição sobre θ . Resta então provar que $|W(0)| = 1$ ou $D(0)$ satisfaz o teste de ocorrência. Se $|W(0)| = 1$ não há nada a provar, portanto suponha que $|W(0)| > 1$. Logo, $W(0)$ e $D(0)$ satisfazem as condições da Proposição 4.4, o que permite concluir que $D(0)$ satisfaz o teste de ocorrência.

Passo de Indução: Seja $k \in [0, K]$ e suponha que I(k) vale. Provaremos que I($k + 1$) vale. Observe inicialmente que $\beta(k)$, $W(k)$ e $D(k)$ são os valores das variáveis no início da ($k + 1$)-ésima iteração. Observe ainda que o teste de comando de iteração deve ser satisfeito ao início de uma nova iteração. Logo, no início da ($k + 1$)-ésima iteração, temos que $\beta(k)$, $W(k)$ e $D(k)$ satisfazem a I(k) e T(k), onde T(k) é a conjunção de:

- T₁(k): $|W(k)| > 1$
- T₂(k): $D(k)$ satisfaz o teste de ocorrência

Seja $\varphi(k)$ a substituição dada pela condição I₂(k). Logo, temos que:

$$(1) \quad \theta = \beta(k)^o\varphi(k)$$

Como T(k) vale, $D(k)$ satisfaz o teste de ocorrência. Logo existe uma variável e um termo em $D(k)$ tais que a variável não ocorre no termo. Sejam x e t a variável e o termo de $D(k)$ escolhidos no passo $k + 1$. Então, por construção do algoritmo,

- (2) $W(k + 1) = W(k)\{x/t\}$
- (3) $\beta(k + 1) = \beta(k)^o\{x/t\}$
- (4) $D(k + 1)$ é o conjunto de discórdia de $W(k + 1)$

Logo $I_3(k+1)$ é satisfeita. Provaremos separadamente que $I_1(k+1)$, $I_2(k+1)$ e $I_4(k+1)$ valem.

Provaremos inicialmente que $I_1(k+1)$ vale, ou seja, que $W(k+1) = \mathbf{E}\beta(k+1)$. Como $I(k)$ vale, temos $W(k) = \mathbf{E}\beta(k)$. Logo, usando (2) e (3), temos:

$$(5) \quad W(k+1) = W(k)\{x/t\} = (\mathbf{E}\beta(k))\{x/t\} = \mathbf{E}(\beta(k)^o\{x/t\}) = \mathbf{E}\beta(k+1)$$

Mostraremos agora que $I_2(k+1)$ vale. Na verdade provaremos que $\theta = \beta(k+1)^o\varphi(k+1)$ para

$$(6) \quad \varphi(k+1) = \varphi(k) - \{x/t\varphi(k)\}.$$

Como x não ocorre em t , pela escolha de x e t , uma substituição de x por um termo qualquer não afeta t . Logo, usando (6), temos:

$$(7) \quad t\varphi(k+1) = t(\varphi(k) - \{x/t\varphi(k)\}) = t\varphi(k)$$

o que implica em:

$$\begin{aligned} (8) \quad \{x/t\}^o\varphi(k+1) &= \{x/t\varphi(k+1)\} \cup \varphi(k+1) \\ &= \{x/t\varphi(k)\} \cup \varphi(k+1) \\ &= \{x/t\varphi(k)\} \cup (\varphi(k) - \{x/t\varphi(k)\}) \\ &= \varphi(k) \end{aligned}$$

Portanto, usando (1), (8) e (3), temos

$$\begin{aligned} (9) \quad \theta &= \beta(k)^o\varphi(k) = \beta(k)^o(\{x/t\}^o\varphi(k+1)) = (\beta(k)^o\{x/t\})^o\varphi(k+1)) \\ &= \beta(k+1)^o\varphi(k+1) \end{aligned}$$

Mostraremos finalmente que $I_4(k+1)$ vale, ou seja, que $|W(k+1)| = 1$ ou $D(k+1)$ satisfaz o teste de ocorrência. Se $|W(k+1)| = 1$ não há nada a provar, portanto suponha que $|W(k+1)| > 1$. Provaremos inicialmente que $\varphi(k+1)$ unifica $W(k+1)$. Por (9) e (5) temos:

$$(10) \quad \mathbf{E}\theta = \mathbf{E}(\beta(k+1)^o\varphi(k+1)) = (\mathbf{E}\beta(k+1))\varphi(k+1) = W(k+1)\varphi(k+1)$$

Mas, como θ unifica \mathbf{E} por suposição,

$$(11) |\mathbf{E}\theta| = 1 = |W(k+1)\varphi(k+1)|$$

Logo $\varphi(k+1)$ unifica $W(k+1)$. Assim, $W(k+1)$ e $D(k+1)$ satisfazem as condições da Proposição 4.4, o que permite concluir que $D(k+1)$ satisfaz o teste de ocorrência. Com isto concluímos a prova de que $I(k+1)$ vale, logo a indução.

Portanto, $\beta(k)$, $W(k)$ e $D(k)$, o valor das variáveis de programa β , W e D após a k -ésima execução do corpo do comando de repetição, satisfazem a $I(k)$, para todo $k \in [0, K]$ e para todo unificador θ de \mathbf{E} .

Provaremos finalmente que o algoritmo retorna um u.m.g para \mathbf{E} . Como supusemos que o corpo do comando de repetição é executado K vezes, $\beta(K)$, $W(K)$ e $D(K)$, o valor das variáveis de programa β , W e D após a última execução do corpo do comando de repetição, satisfazem a $\neg T(K)$, pois após o término do comando de repetição o seu teste é falso. Logo, $\beta(K)$, $W(K)$ e $D(K)$ satisfazem a $I(K)$ e $\neg T(K)$, para todo unificador θ de \mathbf{E} . Após uma simplificação imediata, temos que $\beta(K)$, $W(K)$ e $D(K)$ satisfazem a $|W(K)| = |\mathbf{E}\beta(K)|$ e existe φ tal que $\theta = \beta(K)^o\varphi$ e $|W(K)| = 1$, para todo unificador θ de \mathbf{E} . Logo, como $|W(K)| = 1$, pelo comando "if...then" após o comando de repetição, o algoritmo retornará a substituição $\beta(K)$. Como $|W(K)| = 1 = |\mathbf{E}\beta(K)|$, $\beta(K)$ é um unificador para \mathbf{E} . Assim, para todo unificador θ de \mathbf{E} , existe φ tal que $\theta = \beta(K)^o\varphi$ e, portanto, $\beta(K)$ é um u.m.g de \mathbf{E} .

Teorema 4.1: (Teorema da Unificação)

- (a) \mathbf{E} é um conjunto unificável de expressões simples se e somente se o algoritmo de unificação pára retornando um u.m.g para \mathbf{E} .
- (b) \mathbf{E} não é um conjunto unificável de expressões simples se e somente se o algoritmo de unificação pára com NÃO.

Demonstração

- (a) Uma direção (\rightarrow) segue do Lema da Unificação. A outra direção (\leftarrow) segue imediatamente da observação de que, se o algoritmo retorna um u.m.g de \mathbf{E} , em particular, \mathbf{E} é unificável.
- (b) Segue de (a) e do fato do algoritmo sempre parar com um u.m.g de \mathbf{E} ou com 'NÃO'.

4.6 O SISTEMA FORMAL DA RESOLUÇÃO

Esta seção define precisamente o sistema formal da resolução e apresenta um exemplo completo da sua aplicação, concluindo assim a formalização dos conceitos introduzidos na seção 4.2.

Uma *renomeação* para B em presença de A é uma renomeação de variáveis β tal que A e $B\beta$ não possuem variáveis em comum. Recorde que se L é um literal da forma P ou da forma $\neg P$ então P é o *átomo* de L , denotado por $|L|$.

Definição 4.7:

Uma cláusula A é um *fator* de uma cláusula A' se e somente se existe um conjunto L' de literais de A' e um u.m.g β de L' tais que $A = A'\beta$.

Note que, por convenção (ver Definição 4.2), a instanciação de A' por β automaticamente elimina ocorrências duplicadas do mesmo literal, mantendo apenas a mais à esquerda.

Exemplo 4.7:

Sejam as cláusulas

$$\begin{aligned} A' &: \neg p(z_1, a) \neg p(z_1, x_1) \neg p(x_1, z_1) \\ A &: \neg p(z_1, a) \neg p(a, z_1) \end{aligned}$$

Então, A é um fator de A' pois podemos identificar na Definição 4.7:

- $L' = \{\{\neg p(z_1, a), \neg p(z_1, x_1)\}\}$
- $\beta = \{x_1/a\}$

Definição 4.8:

Uma cláusula A é um *resolvente binário* de cláusulas A' e A'' se e somente se existem literais L' e L'' de A' e A'' , respectivamente, e uma substituição θ tais que

- (i) L' e L'' tem sinais opostos e θ é um u.m.g de $\{|L'|, |L''|\}$
- (ii) $A = (A'\theta - L'\theta) \cup (A''\theta - L''\theta)$

Dizemos ainda que L' e L'' são os *literais resolvidos* e que θ é a *substituição de resolução*.

Portanto, por convenção, o resolvente A é formado:

- eliminando $L'\theta$ de $A'\theta$ e $L''\theta$ de $A''\theta$;
- listando os literais restantes de $A'\theta$ e $A''\theta$ em qualquer ordem;
- eliminando os literais repetidos.

Exemplo 4.8:

Sejam as cláusulas

$$\begin{aligned} A' &: \neg p(a,a) \\ A'' &: p(z_2, f(z_2)) \quad p(z_2, a) \\ A &: p(a, f(a)) \end{aligned}$$

Então, A é um resolvente binário de A' e A'' pois podemos identificar na Definição 4.8:

- L' com " $\neg p(a,a)$ "
- L'' com " $p(z_2,a)$ "
- $\theta = \{z_2/a\}$
- $A = (A'\theta - L'\theta) \cup (A''\theta - L''\theta) = p(a,f(a))$

Definição 4.9:

Sejam A' e A'' cláusulas e β uma renomeação para A'' em presença de A' . Uma cláusula A é um *resolvente* de A' e A'' se e somente se A é um resolvente binário de fatores de A' e $A''\beta$.

Exemplo 4.9:

Este exemplo ilustra a importância de renomeação e fatoração, reforçando observações já feitas na seção 4.2. Admitindo a definição correta de resolvente, temos que a cláusula (3) é um resolvente das cláusulas (1) e (2), onde

1. $p(x)$
2. $\neg p(f(x))$
3. \square

Para se convencer de que (3) realmente é um resolvente de (1) e (2), basta renomear x por y em (2) e observar que o conjunto $\{p(x), p(f(y))\}$ é unificável.

Porém, não admitindo renomeação prévia na definição de resolvente, não teríamos nenhum resolvente de (1) e (2), essencialmente porque o conjunto $\{p(x), p(f(x))\}$ não é unificável, já que o seu conjunto de discórdia $\{x, f(x)\}$ não satisfaz o teste de ocorrência.

De forma semelhante, permitindo fatoração, temos que (3) é um resolvente de (1) e (2), onde:

1. $p(a) \ p(x)$
2. $\neg p(a) \ \neg p(x)$
3. \square

De fato, (3) é um resolvente binário de (4) e (5), onde:

4. $p(a)$
5. $\neg p(a)$

e (4) e (5) são fatores de (1) e (2), respectivamente.

No entanto, não admitindo fatoração, todas as possíveis cláusulas geradas a partir de (1) e (2) por resolução teriam dois literais. Portanto, a cláusula vazia nunca seria gerada.

De posse destas noções preliminares, podemos então enunciar o sistema formal da resolução da seguinte forma:

Definição 4.10:

O sistema formal da resolução, **RE**, consiste de:

Classe de Linguagens: linguagens de cláusulas

Axiomas: nenhum

Regra de Inferência: Regra da Resolução (RE)

RE: se A' e A'' são cláusulas e
 A é um resolvente de A' e A'' ,
então derive A de A' e A''

A noção de uma dedução em **RE** é formalizada de maneira semelhante à noção de dedução em um sistema axiomático.

Definição 4.11:

Seja **S** um conjunto de cláusulas e **C** uma cláusula.

- (a) Uma *dedução* de **C** a partir de **S** no sistema formal da resolução ou, simplesmente, uma *R-dedução* de **C** a partir de **S**, é uma seqüência $\mathbf{D} = (\mathbf{D}_1, \dots, \mathbf{D}_n)$ de cláusulas tal que:

- (i) $\mathbf{D}_n = \mathbf{C}$
- (ii) para todo $i \in [1, n]$, \mathbf{D}_i pertence a **S** ou \mathbf{D}_i é um resolvente de \mathbf{D}_j e \mathbf{D}_k , para algum $j, k < i$.

Para cada $i \in [1, n]$, \mathbf{D}_i é uma *cláusula de entrada* em **D** se e somente se \mathbf{D}_i pertence a **S**; caso contrário, \mathbf{D}_i é uma *cláusula derivada*.

- (b) Uma *refutação* a partir de **S** no sistema formal da resolução ou, simplesmente, uma *R-refutação* a partir de **S**, é uma R-dedução de \square a partir de **S**.

Na seção 4.7 provaremos que um conjunto de cláusulas **S** é insatisfatível se e somente se existir uma R-refutação a partir de **S** em **RE**.

Note que a regra da resolução, conforme definida no sistema **RE**, incorpora fatoração. Alternativamente, poderíamos incluir uma regra separada para fatoração, limitando a regra da resolução à derivação de resolventes binários. Este segundo enfoque não foi adotado pois dificulta a definição do método da resolução linear, a ser apresentado no próximo capítulo. Por outro lado, ele tornaria mais transparente a descrição de R-deduções por explicitar os fatores usados. Esta vantagem pode ser incorporada ao sistema **RE** introduzindo fatores explicitamente na descrição das R-deduções através da seguinte convenção:

1. cada cláusula aparece em uma linha separada;
2. os fatores da cláusula na linha i usados em aplicações da regra da resolução aparecem abaixo da cláusula em linhas numeradas " $i.1$ ", " $i.2$ ", etc.
3. " $i.u$ " indica o literal de ordem u da cláusula na linha i , onde u é " a " para o primeiro literal, " b " para o segundo, etc.

4. um par "*l.u, m.v*" após cada resolvente indica que ele foi obtido resolvendo-se o literal de ordem *u* da cláusula (ou fator) na linha *l* contra o literal de ordem *v* da cláusula (ou fator) na linha *m*.

Terminaremos esta seção com um exemplo detalhado de uma R-refutação.

Exemplo 4.10:

Considere o seguinte conjunto de cláusulas:

$$\begin{array}{l} \neg p(z_1, a) \quad \neg p(z_1, x_1) \quad \neg p(x_1, z_1) \\ p(z_2, f(z_2)) \quad p(z_2, a) \\ p(f(z_3), z_3) \quad p(z_3, a) \end{array}$$

A seguinte seqüência é uma R-refutação a partir destas cláusulas:

- | | | |
|-----|--|------------|
| 1. | $\neg p(z_1, a) \quad \neg p(z_1, x_1) \quad \neg p(x_1, z_1)$ | |
| 1.1 | $\neg p(a, a)$ | |
| 1.2 | $\neg p(z_1, a) \quad \neg p(a, z_1)$ | |
| 2. | $p(z_2, f(z_2)) \quad p(z_2, a)$ | |
| 3. | $p(f(z_3), z_3) \quad p(z_3, a)$ | |
| 4. | $p(a, f(a))$ | . 1.1a, 2b |
| 5. | $p(f(a), a)$ | . 1.1a, 3b |
| 6. | $\neg p(f(a), a)$ | . 1.2b, 4a |
| 7. | \square | . 5a, 6a |

Explicaremos em detalhe apenas como obtivemos a cláusula em (4), já que as demais seguem de forma semelhante. A cláusula em (4) é um resolvente de (1) e (2) pois podemos identificar na Definição 4.9:

- A' com a cláusula em (1)
- A'' com a cláusula em (2)
- β com a substituição vazia
- fator de A' : a cláusula em (1.1)
- fator de A'' : a própria cláusula em (2)

A cláusula em (1.1) é um fator de (1) pois podemos identificar na Definição 4.7:

- L' com a própria cláusula em (1)
- $\beta = \{z_1/a, x_1/a\}$

Finalmente, a cláusula em (4) é um resolvente binário de (1.1) e (2) pois podemos identificar na Definição 4.8:

- L' com " $\neg p(a,a)$ "
- L'' com " $p(z_2,a)$ "
- $\theta = \{z_2/a\}$
- $A = (A'\theta - L'\theta) \cup (A''\theta - L''\theta) = p(a,f(a))$

4.7 CORREÇÃO E COMPLETUDADE DO SISTEMA FORMAL DA RESOLUÇÃO

Esta seção apresenta uma demonstração de que o sistema formal da resolução é refutacionalmente consistente e completo. Ou seja, um conjunto de cláusulas S é insatisfatível se e somente se existe uma R-refutação a partir de S .

Denote o fecho universal da disjunção dos literais de uma cláusula C por $\forall C$. Então, C e $\forall C$ são equivalentes, no sentido de que I satisfaz C se, por definição, I satisfaz $\forall C$.

A correção do sistema da resolução baseia-se no seguinte lema auxiliar.

Lema 4.3: (Lema da Correção para Resolução)

Se C é um resolvente de duas cláusulas C' e C'' , então C' e C'' implicam logicamente C .

Demonstração

Dividiremos a prova em três partes:

Provaremos inicialmente que toda estrutura que satisfaz uma cláusula C também satisfaz qualquer fator $C\beta$ de C . De fato, seja $C\beta$ um fator de C . Seja I uma estrutura que satisfaz C . Logo, I satisfaz $\forall C$. Observe que $(\forall C \rightarrow \forall C\beta)$ é uma fórmula válida. Logo, I satisfaz $\forall C\beta$ e, portanto, I satisfaz $C\beta$.

Provaremos agora que toda estrutura que satisfaz C' e C'' também satisfaz qualquer resolvente binário C de C' e C'' . Seja C um resolvente binário de C' e C'' , L' e L'' os literais de C' e C'' resolvidos para obter C e β a substituição usada. Seja I uma estrutura que satisfaz C' e C'' , logo $\forall C'$ e $\forall C''$. Temos que $(\forall C' \rightarrow \forall C'\beta)$ e $(\forall C'' \rightarrow \forall C''\beta)$ são fórmulas válidas.

Logo, $(\forall C' \rightarrow \forall C'') \rightarrow \forall C$ também é uma fórmula válida pois $C = (C'\beta - L'\beta) \cup (C''\beta - L''\beta)$ e, como $L'\beta$ e $L''\beta$ são literais complementares, não podem ser simultaneamente satisfeitos. Logo, como I satisfaz $\forall C'$ e $\forall C''$, I satisfaz $\forall C$, logo C .

Finalmente, provaremos o resultado desejado. Seja C um resolvente de C' e C'' , β a renomeação de C'' em presença de C' e B' e B'' os fatores de C' e $C''\beta$, respectivamente, usados na derivação de C . Seja I uma estrutura que satisfaz C' e C'' . Como β é uma renomeação, I satisfaz $C''\beta$. Por (a), I satisfaz B' e B'' . Logo, por (b), I satisfaz C .

Deste lema obtemos outro que será útil também no Capítulo 8.

Lema 4.4:

Se existe uma R-dedução a partir de um conjunto de cláusulas S terminando em uma cláusula C , então S implica logicamente C .

Demonstração

Suponha que exista uma R-dedução D a partir de S e terminando em uma cláusula C . Suponha que D tenha comprimento n . Provaremos, por indução sobre i , para $i \in [1, n]$, que S implica logicamente D_i .

Base: Por definição de R-dedução, D_1 pertence a S . Logo, S trivialmente implica D_1 .

Passo de Indução: Seja $i \in (1, n]$ e suponha que S implique D_j , para todo j tal que $j < i$. Provaremos que S implica D_i . Se D_i pertence S , então S trivialmente implica D_i . Suponha então que D_i é um resolvente de D_p e D_q , para $p, q < i$. Pela hipótese de indução, S implica D_p e D_q . Logo, pelo Lema da Correção, S implica D_i .

Isto conclui a indução. Logo, para todo $i \in [1, n]$, S implica D_i . Portanto, S implica C .

O Teorema da Correção segue imediatamente deste lema.

Teorema 4.2: (Correção da Resolução)

Para todo conjunto **S** de cláusulas, se existe uma R-refutação a partir de **S**, então **S** é insatisfatível.

Demonstração

Seja **S** um conjunto de cláusulas e suponha que exista uma R-refutação **D** a partir de **S**. Logo, **S** implica logicamente a cláusula vazia, pelo lema anterior. Assim, **S** é insatisfatível.

A prova da completude do sistema **RE** baseia-se nas seguintes observações. Suponha que **S** seja um conjunto insatisfatível de cláusulas. Desejamos provar que existe uma R-refutação a partir de **S**. Como supusemos que **S** é insatisfatível, pelo Lema 3.4, existe então uma árvore semântica completa e fechada para **S**. Esta árvore semântica induz um conjunto finito e insatisfatível de instâncias básicas de cláusulas em **S** (isto nada mais é do que o Teorema de Herbrand). Porém a estrutura da árvore indica também como combinar estas instâncias em uma R-refutação. Esta ainda não é uma R-refutação a partir de **S**. Porém, através do Lema da Promoção enunciado abaixo, a R-refutação utilizando instâncias básicas pode ser transformada em uma R-refutação a partir de **S**. Portanto, se **S** é insatisfatível, existe uma R-refutação a partir de **S**.

Lema 4.5: (Lema da Promoção)

Sejam A' e A'' cláusulas sem variáveis em comum. Se B' e B'' são instâncias de A' e A'' e B é um resolvente binário de B' e B'' , então existe um resolvente A de A' e A'' tal que B é uma instância de A .

Demonstração

Sejam A' e A'' cláusulas sem variáveis em comum. Sejam B' e B'' instâncias de A' e A'' . Suponha que B seja um resolvente binário de B' e B'' . Sejam L' e L'' os literais de B' e B'' resolvidos para gerar B e λ a substituição usada. Logo:

$$(1) B = (B'\lambda - L'\lambda) \cup (B''\lambda - L''\lambda)$$

Como B' e B'' são instâncias de A' e A'' , $B'\lambda$ e $B''\lambda$ também o são. Ou seja, existem substituições θ' e θ'' tais que $B'\lambda = A'\theta'$ e $B''\lambda = A''\theta''$. Mas,

por suposição, A' e A'' não possuem variáveis em comum. Logo, nenhuma substituição simples em θ' possui o mesmo primeiro elemento que uma substituição simples em θ'' . Logo, $\theta = \theta' \cup \theta''$ é uma substituição válida. Por construção de θ , temos então que:

$$(2) B'\lambda = A'\theta \text{ e } B''\lambda = A''\theta$$

Além disto, existem conjuntos de literais $L' = \{L_1', \dots, L_m'\}$ de A e $L'' = \{L_1'', \dots, L_n''\}$ de A'' tais que:

$$(3) L'\lambda = L_1'\theta = \dots = L_m'\theta \text{ e } L''\lambda = L_1''\theta = \dots = L_n''\theta$$

Se $m = 1$, seja φ' a substituição vazia, senão seja φ' um u.m.g de L' . Seja

$$(4) M' = L_1'\varphi' = \dots = L_m'\varphi'$$

Logo, como φ' é um u.m.g de L' e como, por (3), θ unifica L' , temos:

$$(5) \theta = (\varphi'^o\gamma'), \text{ para alguma substituição } \gamma'$$

Da mesma forma, se $n = 1$, seja φ'' a substituição vazia, senão seja φ'' um u.m.g de L'' . Seja

$$(6) M'' = L_1''\varphi'' = \dots = L_n''\varphi''$$

Logo, como φ'' é um u.m.g de L'' e como, por (3), θ unifica L'' , temos:

$$(7) \theta = (\varphi''^o\gamma''), \text{ para alguma substituição } \gamma''$$

Como A' e A'' não possuem variáveis em comum, L' e L'' não possuem variáveis em comum. Logo, nenhuma substituição simples em γ' possui o mesmo primeiro elemento que uma substituição simples em γ'' . Logo, $\varphi = \gamma' \cup \gamma''$ é uma substituição válida. Além disto, por construção de φ , por (5) e por (7), temos:

$$(8) (\varphi'^o\varphi) = \theta = (\varphi''^o\varphi)$$

Provaremos agora que $\{|M'|, |M''|\}$ é unificável por φ . De fato, por (4), (8) e (3) temos:

$$(9) M'\varphi = L_i'(\varphi'^o\varphi) = L_i'\theta = L'\lambda, \text{ para } i = 1, \dots, n$$

e por (3), (8) e (6) temos:

$$(10) L''\lambda = L_i''\theta = L_i''(\varphi''^o\varphi) = M''\varphi, \text{ para } i=1,\dots,n$$

Mas $|L'\lambda| = |L''\lambda|$ pois λ unifica $\{|L'|, |L''|\}$. Assim, $|M'\varphi| = |M''\varphi|$, por (9) e (10). Ou seja, $\{|M'|, |M''|\}$ é unificável por φ . Seja β um u.m.g de $\{|M'|, |M''|\}$. Como φ unifica este conjunto, temos:

$$(11) \varphi = (\beta^o\beta'), \text{ para alguma substituição } \beta'$$

Sejam A , C' e C'' as seguintes cláusulas:

$$(12) C' = A'\varphi' \text{ e } C'' = A''\varphi''$$

$$(13) A = (C'\beta - M'\beta) \cup (C''\beta - M''\beta)$$

Então C' e C'' são fatores de A' e A'' pois φ' é um u.m.g do conjunto de literais L' de A' e φ'' é um u.m.g do conjunto de literais L'' de A'' . Além disto, M' e M'' são literais de sinais opostos de C' e C'' , e β é um u.m.g de $\{|M'|, |M''|\}$. Logo, A é um resolvente binário de C' e C'' , ou seja, A é um resolvente de A' e A'' .

Resta mostrar que B é uma instância de A . Provaremos, na verdade, que $A\beta' = B$.

Por (11) e (13) temos:

$$(14) A\beta' = (C'\varphi - M'\varphi) \cup (C''\varphi - M''\varphi)$$

Por (12), (4), (12) novamente, e (6):

$$(15) A\beta' = (A'(\varphi'^o\varphi) - L_i'(\varphi'^o\varphi)) \cup (A''(\varphi''^o\varphi) - L_i''(\varphi''^o\varphi))$$

Por (8):

$$(16) A\beta' = (A'\theta - L_i'\theta) \cup (A''\theta - L_i''\theta)$$

Finalmente, por (1), (2) e (3):

$$(17) A\beta' = (B'\lambda - L'\lambda) \cup (B''\lambda - L''\lambda)$$

Teorema 4.3: (Completude da Resolução)

Para todo conjunto \mathbf{S} de cláusulas, se \mathbf{S} é insatisfável, então existe uma R-refutação a partir de \mathbf{S} .

Demonstração

Pelo Lema 3.4, se um conjunto de cláusulas é insatisfável, então existe uma árvore semântica completa e fechada para o conjunto de cláusulas. Provaremos então que, para todo conjunto \mathbf{R} de cláusulas, se \mathbf{R} possui uma árvore semântica completa e fechada então há uma R-refutação a partir de \mathbf{R} . Logo, para todo conjunto insatisfável de cláusulas há uma R-refutação a partir do conjunto.

A prova será por indução sobre n , com o seguinte predicado de indução: para todo conjunto \mathbf{R} de cláusulas, se \mathbf{R} possui uma árvore semântica completa e fechada com n nós de falha então há uma R-refutação a partir de \mathbf{R} .

Base: Seja \mathbf{S} um conjunto de cláusulas e suponha que \mathbf{S} tenha uma árvore semântica completa e fechada com 1 nó de falha. Mas isto só é possível se a raiz for um nó de falha, o que implica que \square pertence a \mathbf{S} . Logo trivialmente há uma R-refutação a partir de \mathbf{S} .

Passo de Indução: Seja $n > 1$ e suponha que, para todo conjunto \mathbf{R} de cláusulas, se \mathbf{R} possui uma árvore semântica completa e fechada com i nós de falha, $i < n$, então há uma R-refutação a partir de \mathbf{R} . Seja \mathbf{S} um conjunto de cláusulas e suponha que \mathbf{S} tenha uma árvore semântica A completa e fechada com n nós. Mostraremos que há uma R-refutação a partir de \mathbf{S} .

Note inicialmente que A possui pelo menos um nó de inferência pois, de outra forma, todo nó de A teria um filho que não é um nó de falha, violando a suposição de que A é fechada. Seja N um nó de inferência de A e N' e N'' os filhos de N . Logo, N' e N'' são nós de falha. Sejam L e $\neg L$ os literais básicos rotulando as arestas (N,N') e (N,N'') . Como N' e N'' são nós de falha, mas não N , existem instâncias básicas B' e B'' de cláusulas A' e A'' em \mathbf{S} tais que $comp(B')$ é um subconjunto de $rot(N')$, mas não de $rot(N)$, e $comp(B'')$ é um subconjunto de $rot(N'')$, mas não de $rot(N)$. Logo, B' deverá conter $\neg L$ e B'' deverá conter L . Podemos então obter um resolvente binário B de B' e B'' tal que (a substituição é vazia, pois B' e B'' são cláusulas básicas):

$$(1) B = (B' - \neg L) \cup (B'' - L)$$

Seja β uma renomeação de A'' em presença de A' . Logo B'' também é uma instância básica de $A''\beta$. Pelo Lema da Promoção, existe então um resolvente A de A' e $A''\beta$, logo de A' e A'' , tal que B é uma instância de A .

Mostraremos que A é uma árvore semântica completa e fechada para $S' = S \cup \{A\}$ com $n-1$ nós de falha. Como A é uma árvore semântica completa e fechada com n nós de falha para S , é suficiente mostrar que A tem $n-1$ nós de falha para S' . Provaremos inicialmente que N é um nó de falha de A para S' . Observe que, como $comp(B') \subset rot(N')$ e $comp(B'') \subset rot(N'')$ e como L e $\neg L$ rotulam as arestas (N, N') e (N, N'') de A , temos que

$$(2) comp(B' - \neg L) \subset rot(N) \text{ e } comp(B'' - L) \subset rot(N).$$

Logo,

$$(3) comp((B' - \neg L) \cup (B'' - L)) \subset rot(N).$$

Ou seja, $comp(B) \subset rot(N)$. Como A pertence a S' e B é uma instância de A , N é um nó de falha de A' para S' .

Mas, se N é um nó de falha de A para S' , os seus filhos N' e N'' não o são. Portanto, como A tem n nós de falha para S , incluindo N' e N'' , A tem $n-1$ nós de falha para S' . Logo, pela hipótese de indução, há uma R-refutação D' a partir de S' . Finalmente, como A é um resolvente de A' e A'' e A pertence a S' , podemos estender D' para uma R-refutação D a partir de S .

NOTAS BIBLIOGRÁFICAS

O princípio da resolução foi introduzido em Robinson [1965]. Desde então inúmeras variações e extensões de resolução foram publicadas. Loveland [1978] contém uma excelente exposição de grande parte destas variações. Robinson [1983] contém uma excelente introdução histórica, usada como fonte para a seção 4.1. Algoritmos lineares de unificação podem ser encontrados em Paterson e Wegman [1978] e em Martelli e Montanari [1982]. Dwork, Kanellakis e Mitchell [1985] contém resultados importantes sobre a complexidade do problema de unificação.

CAPÍTULO 5: MÉTODOS DE REFUTAÇÃO POR RESOLUÇÃO

Este capítulo apresenta alguns métodos de refutação baseados em resolução. A seção 5.2 contém uma discussão bastante geral que estabelece de forma clara as diferenças entre um método e um procedimento de dedução. A seção 5.3 descreve os métodos de resolução por saturação, resolução por saturação com eliminação de tautologias e de cláusulas subjugadas, e resolução com conjunto de suporte. A seção 5.4 define o método de resolução linear, deixando para a seção 5.5 a prova da completude deste método. Finalmente, a seção 5.6 discute brevemente, através do conceito de árvore de refutação, como construir procedimentos de refutação baseados em resolução linear.

As seções recomendadas para cada nível de leitura são:

nível introdutório: 5.2, 5.4 e 5.6

nível intermediário: 5.2, 5.3, 5.4 e 5.6

5.1 INTRODUÇÃO

O sistema formal da resolução serve de base para uma série bastante extensa de métodos de refutação para conjuntos de cláusulas, ou seja, métodos para testar se um conjunto de cláusulas é insatisfatível ou não.

Este capítulo descreve inicialmente alguns destes métodos, escolhidos dentre os mais simples. O método de resolução por saturação é essencialmente uma enumeração exaustiva de todas as R-deduções a partir de um dado conjunto de cláusulas. Este método não tem interesse prático pois a liberdade de escolha de cláusulas, fatores e literais permitida pela regra da resolução resulta em uma explosão combinatorial. Os outros dois métodos elementares discutidos neste capítulo, resolução por saturação com eliminação de tautologias e de cláusulas subjugadas, e resolução com conjunto de suporte, são refinamentos imediatos da resolução por saturação, no sentido de evitarem a geração de certos tipos de cláusulas que necessariamente não contribuem para a geração da cláusula vazia. Provas formais da completude destes métodos serão omitidas, devendo o leitor consultar Chang e Lee [1973] ou Loveland [1978].

A parte central do capítulo aborda o método de resolução linear, discutindo como construir procedimentos de refutação baseados neste método. Um dedução é linear se cada cláusula derivada for obtida resolvendo-se a cláusula imediatamente anterior contra uma cláusula de entrada ou uma cláusula anteriormente derivada. Esta restrição simples permite estruturar o conjunto de todas as deduções lineares a partir de uma dado conjunto de cláusulas de forma relativamente simples, oferecendo uma oportunidade excepcional para a aplicação de heurísticas.

5.2 MÉTODOS E PROCEDIMENTOS DE DEDUÇÃO

Esta seção contém uma discussão bastante geral sobre sistemas formais, métodos e procedimentos de dedução que permite colocar estes conceitos em uma perspectiva mais ampla. Esta discussão torna-se importante neste ponto pois este capítulo e o seguinte apresentarão vários exemplos destes conceitos.

Um *sistema formal F* é um par (L, D) , onde L é uma família de linguagens formais e D é um conjunto de seqüências de sentenças em uma mesma linguagem. Dada uma sentença C de uma linguagem de F , uma *dedução* de C em F é uma seqüência D em D terminando em C .

Um *sistema axiomático* é um sistema formal tal que o conjunto das deduções é especificado por um conjunto de sentenças, chamadas de *axiomas lógicos*, e um conjunto de funções mapeando sentenças em uma sentença, chamadas de *regras de inferência*. Neste caso, dado um conjunto de sentenças S e uma sentença C de uma linguagem na classe de linguagens

de F , uma dedução de C em F , a partir de S , consiste de uma seqüência de sentenças terminando em C e gerada a partir de S e dos axiomas de F através das regras de inferência de F , da forma usual.

O sistema AX da seção 2.6 e o sistema RE da seção 4.6 são exemplos de sistemas axiomáticos. Recorde que AX possui um número infinito de axiomas e uma regra de inferência, chamada de Modus Ponens, e que RE não possui axiomas e apenas uma regra de inferência, a regra da resolução.

Neste e no próximo capítulo estaremos interessados em simplificar o processo de dedução em certos sistemas formais impondo restrições sobre as deduções. Convém então introduzir a noção de um *método de dedução* como consistindo de um par $M = (F, A)$, onde F é um sistema formal e A é um conjunto de deduções em F , chamadas de *deduções admissíveis* em M .

Dado um método de dedução M , o *espaço de busca* de um conjunto de sentenças S é o conjunto de todas as deduções a partir de S admissíveis em M . Um método de dedução M é um *refinamento* de um método de dedução M' se e somente se

- os sistemas formais de ambos admitem a mesma classe de linguagens;
- para todo conjunto S de sentenças, o espaço de busca de S em M está contido no espaço de busca de S em M' ;
- existe pelo menos um conjunto S de sentenças tal que o espaço de busca de S em M não é igual ao espaço de busca de S em M' ;

Um *procedimento de dedução* baseado em um método $M = (F, D)$ é um algoritmo que recebe como entrada um conjunto S de sentenças e uma sentença C de uma das linguagens de F e gera sistematicamente todas as deduções no espaço de busca de S em M até encontrar uma dedução de C a partir de S .

Suponha no resto desta seção que $M = (F, D)$ seja um método de dedução tal que F admite apenas as linguagens de cláusulas.

Neste contexto, uma *refutação* em F é uma dedução da cláusula vazia em F . O método M é *refutacionalmente correto* se e somente se, para todo conjunto de cláusulas S , se existe uma refutação no espaço de busca de S em M então S é insatisfatível. O método M é *refutacionalmente completo* se e somente se, para todo conjunto de cláusulas S , se S é insatisfatível

então existe uma refutação no espaço de busca de \mathbf{S} em M . Note que, se F é correto, então M é refutacionalmente correto, mas se F é completo M pode ou não ser refutacionalmente completo, pois uma dedução em F não é necessariamente admissível em M . Semelhantemente, se M' é um refinamento de M e M' é refutacionalmente correto, então M é refutacionalmente correto, mas M' pode ser refutacionalmente completo e M ser incompleto.

Finalmente, *procedimentos de refutação* são procedimentos de dedução especializados para detetar refutações.

5.3 MÉTODOS ELEMENTARES DE RESOLUÇÃO

5.3.1 Resolução por Saturação

O sistema formal da resolução imediatamente induz o método de dedução definido abaixo:

Definição 5.1:

O *método de resolução por saturação* consiste do par $RS = (RE, DS)$, onde RE é o sistema formal da resolução e DS é o conjunto de todas as R-deduções.

Este método é refutacionalmente correto e completo pois o sistema RE é correto e completo e toda R-dedução é admissível pelo método. Note, porém, que a completude de um método baseado no sistema RE pode não ser imediata pois, em geral, nem toda R-dedução será admissível.

O método de resolução por saturação sugere o seguinte procedimento de refutação:

Procedimento de Resolução por Saturação

1. dado um conjunto finito de cláusulas \mathbf{S} , construa uma seqüência $S(0), S(1), \dots$ de conjuntos de cláusulas da seguinte forma:

$$S(0) = \mathbf{S}$$

$$S(n+1) = S(n) \cup \{A / A \text{ é resolvente de cláusulas em } S(n)\}$$

2. pare com SIM quando \square for gerada
 3. pare com NÃO quando não houver novos resolventes a derivar
-

Note que este procedimento:

- sempre pára com SIM quando S for realmente insatisfatível, pois o método de resolução por saturação é completo;
- nunca pára quando S for satisfatível e existir um conjunto infinito de resolventes obtidos a partir de S ;
- sempre pára com NÃO quando S for satisfatível mas o conjunto de resolventes obtidos a partir de S é finito;

Este procedimento é então um procedimento de decisão parcial para o problema da insatisfatibilidade de conjuntos de cláusulas. Mais ainda, pela observação acima, ele é um procedimento de decisão para o problema da insatisfatibilidade de conjuntos de cláusulas cujo conjunto de resolventes é finito.

5.3.2 Resolução por Saturação com Filtragem

O conjunto das deduções admissíveis pelo método de resolução por saturação contém muitas deduções com passos que definitivamente não contribuem para a obtenção da cláusula vazia. Definiremos então um outro método de dedução, também baseado no sistema RE , mas cujo conjunto de deduções admissíveis elimina certas redundâncias, capturadas nas três definições abaixo.

Definição 5.2:

Um literal L é *puro* em presença de um conjunto S de cláusulas se e somente se não existe um outro literal L' ocorrendo em S tal que L e L' possuem sinais opostos e $\{|L|, |L'|\}$ é unificável.

Se uma cláusula A possui um literal L puro em S , qualquer resolvente de A ou de um resolvente de A conterá necessariamente L . Portanto, A não poderá participar da derivação da cláusula vazia. Como consequência, se uma cláusula em S contiver um literal puro, esta cláusula não deve ser usada para construir uma R-refutação a partir de S .

Definição 5.3:

Uma cláusula A é uma *tautologia* se e somente se A contém dois literais complementares.

Se um conjunto de cláusulas contiver uma tautologia A e for insatisfatível, a remoção de A manterá a insatisfatibilidade do conjunto. Assim, durante a construção de uma refutação, deve-se evitar a geração de cláusulas que sejam tautologias.

Recorde que $\forall A$ denota o fecho universal da disjunção dos literais de uma cláusula A. Recorde ainda que, por definição, A é satisfatível se e somente se $\forall A$ for satisfatível.

Definição 5.4:

Uma cláusula A *subjuga* uma cláusula B se e somente se $\forall A \rightarrow \forall B$ for uma fórmula válida.

Se existirem cláusulas A e B em um conjunto insatisfatível de cláusulas tais que A subjuga B, então B poderá ser removida do conjunto sem destruir a sua insatisfatibilidade. Assim, durante a construção de uma refutação, deve-se evitar a geração de uma cláusula B se existir uma cláusula A anteriormente gerada tal que A subjuga B.

A combinação destas observações leva à definição do seguinte conjunto de deduções:

Definição 5.5:

Uma R-dedução $D = (D_1, \dots, D_n)$ a partir de S é *filtrada* se e somente se, para todo i em $[1, n]$,

- (i) se D_i pertence a S , então nenhum literal de D_i é puro em S , e
- (ii) D_i não é uma tautologia, e
- (iii) não existe j , com $j < i$, tal que D_j subjuga D_i .

Definição 5.6:

O método de resolução por saturação com filtragem consiste do par $R = (RE, DF)$, onde RE é o sistema formal da resolução e DF é o conjunto das R-deduções filtradas.

É possível provar que este método é refutacionalmente correto e completo, observando apenas que o fato de uma cláusula subjugar seus fatores não afeta a completude do método pois a regra da resolução, conforme definida, incorpora fatoração.

O método de resolução com filtragem sugere o seguinte procedimento de refutação, que é uma modificação imediata do procedimento apresentado na seção anterior:

Procedimento de Resolução por Saturação com Filtragem

1. dado um conjunto finito de cláusulas S , construa uma seqüência $S(0), S(1), \dots$ de conjuntos de cláusulas da seguinte forma:

$$S(0) = \{A \in S / A \text{ não possui um literal puro em } S\}$$

$$S(n+1) = S(n) \cup \{A / A \text{ é resolvente de cláusulas em } S(n) \text{ e} \\ A \text{ não é uma tautologia, e} \\ \text{não existe } B \in S(n) \text{ tal que } B \text{ subjuga } A\}$$

2. pare com SIM quando \square for gerada
 3. pare com NÃO quando não houver novos resolventes a derivar
-

Este procedimento é também um procedimento de decisão parcial para o problema da insatisfatibilidade de conjuntos de cláusulas, já que o método de resolução por saturação com filtragem é refutacionalmente completo.

Os testes para eliminação de cláusulas, contidos neste procedimento, requerem alguma discussão adicional:

- A eliminação das cláusulas com literais puros do conjunto S pode ser feita com o auxílio do Algoritmo de Unificação apresentado anteriormente. Esta filtragem, apesar de demorada, deve ser mantida no procedimento pois evita a geração de muitos resolventes inúteis.
- A eliminação de resolventes que são tautologias é imediata pois basta determinar a presença de literais complementares no resolvente.

- Já a eliminação de cláusulas subjugadas é um processo caro em termos de tempo e deve ser mantida apenas em casos especiais, como quando a cláusula que subjuga possui apenas um literal.

O resto desta seção contém uma discussão mais detalhada sobre este último ponto, incluindo um critério mais forte de subjugamento, mas computacionalmente mais simples.

Observemos inicialmente que resolução pode ser usada para testar se A subjuga B, como sugere a seguinte proposição:

Proposição 5.1:

Sejam A e B cláusulas. Suponha que B seja da forma " $M_1 \dots M_n$ ". Seja β uma substituição das variáveis de B por constantes distintas que não ocorrem em A ou B. Seja $C = \{A, B_1, \dots, B_n\}$, onde B_i é a cláusula " $\neg M_i \beta$ ". Então, A subjuga B se e somente se existe uma R-refutação a partir de C.

Demonstração

Observe inicialmente que " $\forall A \rightarrow \forall B$ " é uma fórmula válida se e somente se " $\forall A \wedge \neg \forall B$ " for insatisfatível. Mas, transformando-se " $\forall A \wedge \neg \forall B$ " para a forma clausal, obtemos o conjunto de cláusulas $C = \{A, B_1, \dots, B_n\}$, onde B_i é a cláusula " $\neg M_i \beta$ " e β é uma substituição das variáveis de B por constantes distintas que não ocorrem em A ou B (β reflete o resultado da Skolemização de $\neg \forall B$). Logo, " $\forall A \wedge \neg \forall B$ " é insatisfatível se e somente se o conjunto C for insatisfatível se e somente se existir uma R-refutação a partir de C.

Portanto, determinar se A subjuga B reduz-se a determinar se existe uma R-refutação a partir de C. Porém, como esta forma de testar subjugamento tem um custo elevado em termos de tempo, em geral adota-se uma condição mais forte definida da seguinte forma.

Definição 5.7:

Uma cláusula A subjuga B por substituição se e somente se existe uma substituição θ tal que todo literal de $A\theta$ ocorre em B.

Note que se A subjuga B por substituição então A subjuga B, mas não vice-versa. O teste para determinar se A subjuga B por substituição é uma

versão simplificada do teste de subjugamento através de resolução e baseia-se na seguinte observação:

Proposição 5.2:

Sejam A e B cláusulas. Suponha que B seja da forma " $M_1 \dots M_n$ " e que B não seja uma tautologia. Seja β uma substituição das variáveis de B por constantes distintas que não ocorrem em A ou B . Seja $C = \{A, B_1, \dots, B_n\}$, onde B_i é a cláusula " $\neg M_i \beta$ ". Então, A subjuga B por substituição se e somente se existe uma R-refutação a partir de C tal que, em cada aplicação da regra da resolução, um dos resolventes é B_i , para algum i em $[1, n]$.

Demonstração

(\rightarrow) Suponha que A subjuga B por substituição, ou seja, que existe uma substituição θ tal que todo literal de $A\theta$ ocorre em B . Logo, todo literal de $A(\theta^\circ\beta)$ ocorre em $B\beta$. Portanto, os literais da cláusula $A(\theta^\circ\beta)$ podem ser todos resolvidos contra literais da forma $\neg M_i \beta$, para i em $[1, n]$. Mas $A(\theta^\circ\beta)$ é uma instância de A . Logo, os literais de A também podem ser todos resolvidos contra literais da forma $\neg M_i \beta$, para i em $[1, n]$. O resultado é uma derivação de \square a partir de C .

(\leftarrow) (Esta parte é deixada como exercício e depende da suposição de que B não é uma tautologia).

O algoritmo para determinar se A subjuga B por substituição será então o seguinte:

Algoritmo 5.1: Algoritmo de Subjugamento por Substituição

- entrada:**
- cláusula A
 - cláusula B tal que B é da forma " $M_1 \dots M_n$ " e B não é tautologia
- sída:**
- 'SIM', se A subjuga B por substituição
 - 'NÃO', em caso contrário

begin

seja β uma substituição das variáveis de B por constantes distintas que não ocorrem em A ou B;
 $W := \{\neg M_1\beta, \dots, \neg M_n\beta\}$;
 $U := \{A\}$;
while \square não pertence a U e $U \neq \emptyset$ **do**
 $U := \{D / D$ é um resolvente de D' e D'' , para $D' \in U$ e $D'' \in W\}$;
if $\square \in U$
then return 'SIM'
else return 'NÃO'

end

Seja $U(k)$ o valor de U no k -ésimo passo. Note que, para qualquer $k \geq 0$, cada cláusula em $U(k)$ possui um literal a menos que a cláusula correspondente de $U(k-1)$. Logo, eventualmente ou $U(k) = \emptyset$ ou $\square \in U(k)$. Em ambos os casos, o algoritmo pára. Pela proposição anterior, o algoritmo retorna corretamente SIM, se A subjuga B por substituição, e NÃO em caso contrário.

Por fim, observamos que subjugamento por substituição ainda é um processo caro em termos de tempo, devendo ser adotado também apenas em casos especiais.

5.3.3 Resolução com Conjunto de Suporte

Esta seção define um outro refinamento do método de resolução por saturação baseado na idéia de conjunto de suporte.

Definição 5.8:

Um subconjunto T de um conjunto de cláusulas S é um *conjunto de suporte* de S se e somente se $S - T$ é satisfatível.

Se T é um conjunto de suporte de S então deve-se evitar resolver duas cláusulas tais que ambas pertençam a $S - T$ pois, como este conjunto é satisfatível, todas as cláusulas direta ou indiretamente dele derivadas pela regra da resolução serão satisfatóveis. Esta afirmação é justificada pelo Lema da Correção para Resolução da seção 4.7. Logo, a cláusula vazia nunca poderá ser derivada partindo apenas de cláusulas em $S - T$. Mais precisamente, temos:

Definição 5.9:

Seja T um conjunto de suporte de S . Uma R-dedução D a partir de S tem *conjunto de suporte* T se e somente se nenhuma cláusula derivada em D foi obtida resolvendo-se duas cláusulas em $S - T$.

Exemplo 5.1: (Chang e Lee [1973])

Seja S o seguinte conjunto de cláusulas:

1. $p(f(x_1, y_1), x_1, y_1)$
2. $\neg p(x_2, g(x_2, y_2), y_2)$
3. $\neg p(x_3, y_3, u_3) \ p(y_3, z_3, v_3) \ \neg p(x_3, v_3, w_3) \ p(u_3, z_3, w_3)$
4. $\neg p(h(x_4), x_4, h(x_4))$

Seja T o conjunto consistindo apenas da cláusula (4). Então a R-refutação abaixo tem conjunto de suporte T pois nenhuma cláusula é obtida resolvendo-se as cláusulas (1), (2) e (3) entre si.

5. $\neg p(x_3, y_3, h(z_3)) \ p(y_3, z_3, v_3) \ \neg p(x_3, v_3, h(z_3)) \quad . 3d, 4a$
6. $\neg p(x_3, y_3, h(g(y_3, v_3))) \ \neg p(x_3, v_3, h(g(y_3, v_3))) \quad . 2a, 5b$
- 6.1 $\neg p(x_3, y_3, h(g(y_3, y_3))) \quad . \text{fator de } 6$
7. $\square \quad . 1a, 6.1a$

Definição 5.10:

O *método de resolução com conjunto de suporte* consiste do par $RC = (RE, DC)$, onde RE é o sistema formal da resolução e DC é o conjunto das R-deduções com conjunto de suporte.

Da mesma forma que os refinamentos anteriores, é possível provar que este método é refutacionalmente completo, no seguinte sentido:

Teorema 5.1:

Se \mathbf{S} é insatisfatível e \mathbf{T} é um conjunto de suporte para \mathbf{S} , então existe uma R-refutação a partir de \mathbf{S} com conjunto de suporte \mathbf{T} .

Para completar esta seção, observamos que o método de resolução com conjunto de suporte é especialmente interessante no contexto de teorias de primeira ordem. De fato, seja \mathbf{P} o conjunto de axiomas de uma teoria, suposto satisfatível, e \mathbf{Q} uma sentença satisfatível. Então \mathbf{Q} é um teorema de \mathbf{P} se e somente se $\mathbf{S} = \mathbf{S}' \cup \mathbf{S}''$ é insatisfatível, onde \mathbf{S}' é um conjunto de cláusulas que representam \mathbf{P} e \mathbf{S}'' é um conjunto de cláusulas que representam $\neg\mathbf{Q}$. Neste caso, \mathbf{S}' e \mathbf{S}'' são ambos conjuntos de suporte de \mathbf{S} . Portanto, para provar que \mathbf{S} é insatisfatível basta, por exemplo, enumerar as R-deduções a partir de \mathbf{S} que tenham \mathbf{S}'' como conjunto de suporte e determinar se uma destas R-deduções é uma R-refutação.

5.4 O MÉTODO DA RESOLUÇÃO LINEAR

Nas definições a seguir, todas as deduções são no sistema formal da resolução, designado por RE , como anteriormente. Recorde que uma cláusula C é de entrada em uma R-dedução a partir de um conjunto \mathbf{S} se e somente se C pertencer a \mathbf{S} .

Definição 5.11:

Seja \mathbf{S} um conjunto de cláusulas, B uma cláusula em \mathbf{S} e C uma cláusula qualquer.

- (a) Uma *R-dedução linear* ou, simplesmente, uma *RL-dedução* de C a partir de \mathbf{S} e iniciando-se em B é uma seqüência $\mathbf{D} = (D_1, \dots, D_n)$ de cláusulas tal que:

- (i) $D_n = C$
- (ii) existe $r \leq n$ tal que D_1, \dots, D_r são cláusulas em \mathbf{S} e $D_r = B$
- (iii) para todo $i \in [r+1, n]$, D_i é um resolvente de D_{i-1} e de D_j , para algum $j < i$.

A cláusula **B** é a *cláusula inicial* e a subseqüência D_1, \dots, D_r é o *prefixo* de **D**. Para cada $i \in [r+1, n]$, D_{i-1} é a *cláusula-pai* e D_i é a *cláusula auxiliar* de D_i .

- (b) Uma RL-dedução é *linear de entrada* se e somente se toda cláusula auxiliar for uma cláusula de entrada.
- (c) Uma *refutação por resolução linear (de entrada)*, ou, simplesmente, uma *RL-refutação (de entrada)* a partir de **S** e iniciando-se em **B** é uma RL-dedução (de entrada) da cláusula vazia a partir de **S** e iniciando-se em **B**.

O prefixo de uma RL-dedução é apenas uma conveniência no sentido de listar as cláusulas de entrada usadas na RL-dedução logo no seu início. Também por conveniência, a última cláusula do prefixo é sempre a cláusula inicial da RL-dedução.

Exemplo 5.2:

Seja **S** o seguinte conjunto de entrada:

1. *chama(a,b)*
2. *usa(b,e)*
3. $\neg chama(x,y) \ depende(x,y)$
4. $\neg usa(x,y) \ depende(x,y)$
5. $\neg chama(x,z) \ \neg depende(z,y) \ depende(x,y)$
6. $\neg depende(a,e)$

A seguinte seqüência de cláusulas é uma RL-refutação a partir de **S** tendo (6) como cláusula inicial:

1. *chama(a,b)*
2. *usa(b,e)*
3. $\neg chama(x,y) \ depende(x,y)$
4. $\neg usa(x,y) \ depende(x,y)$
5. $\neg chama(x,z) \ \neg depende(z,y) \ depende(x,y)$
6. $\neg depende(a,e)$
7. $\neg chama(a,z) \ \neg depende(z,e)$. 6a, 5c
8. $\neg depende(b,e)$. 7a, 1a
9. $\neg usa(b,e)$. 8a, 4b
11. \square . 9a, 2a

Observe que as anotações após cada cláusula derivada são em parte redundantes pois a cláusula-pai é sempre a anterior.

Definição 5.12:

O *método da resolução linear* consiste do par $RL = (RE, DL)$, onde RE é sistema formal da resolução e DL é o conjunto das RL-deduções.

Por ser um refinamento de resolução, este método é refutacionalmente correto. A prova de que o método é refutacionalmente completo aparecerá na seção seguinte. O exemplo seguinte mostra que, se o método da resolução linear admitisse apenas R-deduções lineares de entrada, seria incompleto, mesmo para a versão proposicional.

Exemplo 5.3:

Seja S o seguinte conjunto de entrada:

1. $\neg P \neg Q$
2. $\neg P Q$
3. $P \neg Q$
4. $P Q$

A seguinte seqüência de cláusulas é uma RL-refutação a partir de S :

1. $\neg P \neg Q$
2. $\neg P Q$
3. $P \neg Q$
4. $P Q$
5. P .4b, 3b
6. Q .5a, 2a
7. $\neg P$.6a, 1b
8. \square .7a, 5a

Note que esta RL-refutação não é de entrada pois a cláusula auxiliar de (8), a cláusula (5), não é uma cláusula de entrada. O leitor deve se convencer agora de que não há nenhuma RL-refutação de entrada a partir de S .

Terminaremos esta seção enunciando explicitamente um refinamento importante que combina a restrição envolvendo conjuntos de suporte com resolução linear.

Observe inicialmente que, pela proposição abaixo, testar se uma dedução linear tem conjunto de suporte torna-se bastante simples.

Proposição 5.3:

Seja \mathbf{S} um conjunto de cláusulas e \mathbf{T} um conjunto de suporte para \mathbf{S} . Uma RL-dedução \mathbf{D} a partir de \mathbf{S} tem conjunto de suporte \mathbf{T} se e somente se a primeira cláusula derivada não foi obtida resolvendo-se duas cláusulas em $\mathbf{S} - \mathbf{T}$.

Demonstração

Seja \mathbf{D} uma RL-dedução a partir de \mathbf{S} . Observe que, por \mathbf{D} ser linear, toda cláusula derivada em \mathbf{D} tem como cláusula-pai a cláusula imediatamente anterior. Logo, nenhuma cláusula derivada, exceto possivelmente a primeira, pode resultar da resolução de duas cláusulas em $\mathbf{S} - \mathbf{T}$. Portanto, \mathbf{D} terá conjunto de suporte \mathbf{T} se e somente se a primeira cláusula derivada não foi obtida resolvendo-se duas cláusulas em $\mathbf{S} - \mathbf{T}$.

Seja \mathbf{D} uma RL-dedução a partir de \mathbf{S} e iniciando em \mathbf{B} . Logo, a primeira cláusula derivada de \mathbf{D} resulta de \mathbf{B} e de uma cláusula auxiliar em \mathbf{S} . Suponha que \mathbf{B}' seja esta cláusula. Pela proposição anterior, \mathbf{D} terá \mathbf{T} como conjunto de suporte se e somente se \mathbf{B} ou \mathbf{B}' pertencerem a \mathbf{T} . Mas se \mathbf{B} não pertencer a \mathbf{T} , podemos criar outra RL-dedução \mathbf{D}' , idêntica a \mathbf{D} , exceto que \mathbf{D}' inicia-se em \mathbf{B}' . Este argumento leva então ao seguinte refinamento da noção de conjunto de suporte para deduções lineares:

Definição 5.13:

- (a) Seja \mathbf{S} um conjunto de cláusulas e \mathbf{T} um conjunto de suporte para \mathbf{S} . Uma RL-dedução \mathbf{D} a partir de \mathbf{S} e iniciando-se em \mathbf{B} tem *suporte inicial* de \mathbf{T} se e somente se \mathbf{B} pertence a \mathbf{T} .
- (b) O *método de resolução linear com conjunto de suporte inicial*, consiste do par $RLS = (RE, DLS)$, onde RE é o sistema formal da resolução e DLS é o conjunto das RL-deduções que tem suporte inicial.

A seção seguinte apresentará uma demonstração da completude deste refinamento.

Finalmente, observamos que é possível refinar ainda o método da resolução linear, sem perder completude, restringindo as RL-deduções àquelas que não contêm tautologias ou cláusulas subjugadas, sob vários critérios de subjugamento (ver referências bibliográficas).

5.5 COMPLETITUDE DO MÉTODO DA RESOLUÇÃO LINEAR

Esta seção estabelece a completude do método de resolução linear, incluindo o refinamento com conjunto de suporte.

Lema 5.1:

Seja C uma cláusula básica pertencente a um conjunto S de cláusulas básicas. Se S é insatisfatível e $S - \{C\}$ é satisfatível, então existe uma RL-refutação a partir de S iniciando-se em C .

Demonstração

Provaremos o resultado por indução sobre o número de literais de S (ou seja, a soma total do número de literais das cláusulas de S), denotado por $|S|$.

Base: Se $|S|=0$ então $S=\{\square\}$ e o resultado segue trivialmente.

Passo de Indução: Seja $n > 0$ e suponha que, para todo conjunto S' de cláusulas básicas e toda cláusula básica C' tais que $C' \in S'$ e $|S'| < n$, se S' é insatisfatível e $S' - \{C'\}$ é satisfatível, então existe uma RL-refutação a partir de S' iniciando-se em C' . Sejam S e C tais que $|S|=n$, S é insatisfatível e $S - \{C\}$ é satisfatível.

Caso 1: Suponha que C seja uma cláusula unitária básica.

Suponha que L seja o único literal de C . Seja S' o conjunto obtido removendo-se de S todas as cláusulas onde L ocorre e removendo-se $\neg L$ das cláusulas restantes. Então, S' é insatisfatível. De fato, suponha que exista uma interpretação de Herbrand I' que satisfaça S' . Construa, a partir de I' , uma interpretação de Herbrand I para S tal que o literal L seja verdadeiro. Então, I satisfaz a todas as cláusulas em S , contradizendo a suposição de que S é insatisfatível.

Seja T' um subconjunto insatisfatível de S' tal que todo subconjunto próprio T'' de T' é satisfatível. Então existe pelo menos uma cláusula E' em T' tal que E' foi obtida de uma cláusula E de S removendo-se $\neg L$, pois de outra forma T' seria um subconjunto de $S - \{C\}$ e, portanto, satisfatível. Temos então que T' é insatisfatível, $T' - \{E'\}$ é satisfatível e $|T'| < n$. Pela hipótese de indução, há então uma RL-refutação D' a partir de T' iniciando-se em E' . Reintegrando-se $\neg L$ nas cláusulas de D , exceto em E' , e obtendo E' resolvendo C (que é a cláusula unitária " L ") contra E , obtemos uma RL-dedução D de \square , ou de $\neg L$, a partir de S e iniciando-se em C . Se ocorrer o primeiro caso, D é a RL-refutação desejada. Se ocorrer o segundo caso, basta acrescentar um passo a D , resolvendo-se $\neg L$ contra a cláusula inicial C para obter \square , construindo assim a RL-refutação desejada.

Caso 2: Suponha que C não seja uma cláusula unitária.

Seja L um literal de C e C' a cláusula obtida removendo-se L de C . Seja S' construída como no caso 1. Então, pelo mesmo argumento do caso 1, S' é insatisfatível.

Provaremos agora que $S' - \{C'\}$ é satisfatível. Como $S - \{C\}$ é satisfatível por hipótese, seja I uma interpretação de Herbrand que satisfaz este conjunto. Como S é insatisfatível por hipótese, I não pode satisfazer C . Como L é um literal de C e I não satisfaz C , I não satisfaz L . Seja E' uma cláusula em $S' - \{C'\}$. Então existe uma cláusula E em $S - \{C\}$ tal que E' é a cláusula E sem o literal L . Como I satisfaz E , mas não satisfaz L , I satisfaz E' . Logo, I satisfaz toda cláusula em $S' - \{C'\}$. Note que este argumento também estabelece que $\{L\} \cup (S - \{C\})$ é insatisfatível.

Temos então que $|S'| < n$, S' é insatisfatível e $S' - \{C'\}$ é satisfatível. Logo, pela hipótese de indução, há uma RL-refutação D' a partir de S' iniciando-se em C' . Recolocando-se L em todas as cláusulas de onde foi retirado, obtemos uma nova RL-dedução D'' de L a partir de S iniciando-se em C .

Por argumento anterior, $S''' = \{L\} \cup (S - \{C\})$ é insatisfatível. Mas $S - \{C\}$ é satisfatível por suposição. Logo, pelo caso 1, há uma RL-refutação D''' a partir de S''' iniciando-se na cláusula cujo único literal é L . Concatenando-se D'' com D''' obtemos finalmente uma RL-refutação a partir de S iniciando-se em C .

Teorema 5.2:

Seja C uma cláusula pertencente a um conjunto S de cláusulas. Se S é insatisfatível e $S - \{C\}$ é satisfatível então existe uma RL-refutação a partir de S iniciando-se em C .

Demonstração

Como S é insatisfatível, pelo Teorema de Herbrand, há um conjunto finito S' de instâncias básicas de cláusulas em S e uma instância básica C' de C tais que S' é insatisfatível, C' pertence a S' e $S' - \{C'\}$ é satisfatível. Pelo lema anterior, há uma RL-refutação D' a partir de S' iniciando-se em C' . Usando o Lema da Promoção da seção 4.7, podemos então construir uma RL-refutação D a partir de S iniciando-se em C .

A completude do método da resolução linear e a do refinamento envolvendo conjuntos de suporte seguem como corolários deste teorema.

Corolário 5.1: (Completude da Resolução Linear)

Se S é insatisfatível então existe uma RL-refutação a partir de S iniciando-se em alguma cláusula C pertencente a S .

Demonstração

Como S é insatisfatível, há um subconjunto R de S e uma cláusula C em R tais que R é insatisfatível e $R - \{C\}$ é satisfatível. Pelo teorema anterior, há uma RL-refutação a partir de R , logo de S , iniciando-se em C .

Corolário 5.2: (Completude da Resolução Linear com Conjunto de Suporte Inicial)

Se S é insatisfatível e $S - T$ é satisfatível, onde T é um subconjunto de S , então existe uma RL-refutação a partir de S iniciando-se em alguma cláusula C pertencente a T .

Demonstração

(Semelhante à anterior).

5.6 PROCEDIMENTOS DE REFUTAÇÃO POR RESOLUÇÃO LINEAR

De acordo com os conceitos introduzidos na seção 5.2, um *procedimento de refutação linear* é um algoritmo que recebe como entrada um conjunto **S** de cláusulas e gera sistematicamente todas as RL-deduções a partir de **S**, iniciando-se em alguma cláusula de **S**, até encontrar uma RL-refutação. Mostraremos nesta seção que um procedimento de refutação linear reduz-se a um algoritmo para construir um conjunto de árvores especiais, chamadas de *árvores de RL-refutação*. Ou, equivalentemente, mostraremos que o espaço de busca de cada conjunto de cláusulas **S** no método de resolução linear pode ser descrito por uma coleção de árvores de RL-refutação.

Antes de definir a noção de árvore de RL-refutação, convém introduzir certos conceitos auxiliares. Recorde que uma renomeação é uma substituição de variáveis por variáveis. Pela própria definição, duas cláusulas **A** e **A'** tem em geral resolventes que diferem apenas por uma renomeação de variáveis e pela ordem dos literais (note porém que, por definição, um resolvente não possui literais repetidos). Estes fatos geram grande redundância no espaço de busca de um conjunto de cláusulas, um problema que pode ser facilmente evitado forçando-se uma certa disciplina no uso de variáveis e na ordem dos literais.

Definição 5.14:

Sejam **E** e **E'** duas seqüências de símbolos de um alfabeto de primeira ordem **A**. Suponha que as variáveis de **A** sejam ordenadas x_1, x_2, \dots .

- (a) **E'** é uma *variante* de **E** se e somente se existe uma renomeação β tal que $E' = E\beta$.
- (b) **E'** é a *variante canônica* de **E** se e somente se
 - (i) **E'** é uma variante de **E**
 - (ii) existe $n \geq 0$ tal que as variáveis ocorrendo em **E'** são x_1, \dots, x_n
 - (iii) para todo $i, j \in [1, n]$, a ocorrência de x_i mais à esquerda em **E'** precede a ocorrência de x_j mais à esquerda em **E'** se e somente se $i < j$.

No que se segue suporemos que as cláusulas são sobre um alfabeto de primeira ordem **A** dotado de uma ordenação para as suas variáveis.

Definição 5.15:

Uma cláusula A é uma *canonização* de uma cláusula A' se e somente se A é obtida tomando-se a variante canônica de A' e colocando-se os literais em ordem alfabética, ignorando a negação (se houver empate, o literal positivo precede o negativo).

Definição 5.16:

- (a) Uma cláusula A é um *resolvente canônico* de duas cláusulas A' e A'' se e somente se A é a canonização de um resolvente de A' e A''.
- (b) Uma cláusula A é um *resolvente canônico* de uma cláusula A' em presença de um conjunto de cláusulas S se e somente se A é um resolvente canônico de A' por uma cláusula em S.

Exemplo 5.4:

Sejam A' e A'' as cláusulas

1. $\neg p(x) \neg p(y)$
2. $p(a) q(z)$

O conjunto dos resolventes canônicos de A' e A'' é obtido da seguinte forma. Primeiro construa o conjunto de fatores de A' (A'' não tem fatores):

3. $\neg p(x)$
4. $\neg p(y)$

Construa agora o conjunto dos resolventes de A' e A'':

5. $\neg p(y) q(z)$. 1a, 2a
6. $q(z) \neg p(y)$. 1a, 2a
7. $\neg p(x) q(z)$. 1b, 2a
8. $q(z) \neg p(x)$. 1b, 2a
9. $q(z)$. 3 (ou 4), 2a

Como (5), (6), (7) e (8) possuem a mesma canonização

10. $\neg p(x_1) q(x_2)$

o conjunto dos resolventes canônicos de A' e A'' se reduz às duas cláusulas abaixo:

11. $\neg p(x_1) q(x_2)$
12. $q(x_1)$

Note que os literais em (11) aparecem em ordem alfabética, ignorando-se a negação.

De posse destas noções complementares, podemos definir então o conceito de árvore de refutação.

Definição 5.17:

Sejam S um conjunto de cláusulas e C uma cláusula. Uma árvore, com os nós rotulados por cláusulas, é uma *árvore de refutação por resolução linear* ou, simplesmente, uma *árvore de RL-refutação* para S com cláusula inicial C se e somente se:

- (i) o rótulo da raiz é a canonização de C ;
- (ii) para cada nó N , o conjunto dos filhos de N é definido da seguinte forma. Seja A o rótulo de N e R o conjunto das cláusulas rotulando os ancestrais de N . Então:
 - para cada resolvente canônico A' de A em presença de $R \cup S$, existe exatamente um filho de N rotulado com A' ;
 - todos os filhos de N são rotulados com resolventes canônicos de A em presença de $R \cup S$.

Definição 5.18:

A *floresta de refutação por resolução linear* ou, simplesmente, a *floresta de RL-refutação* para um conjunto de cláusulas S é o conjunto de todas as árvores de RL-refutação para S tendo como cláusula inicial alguma cláusula em S .

Definição 5.19:

Seja S um conjunto de cláusulas e C uma cláusula em S . Seja A uma árvore de RL-refutação para S com cláusula inicial C .

- (a) uma folha de A é um *nó de sucesso* de A se e somente se é rotulada pela cláusula vazia.

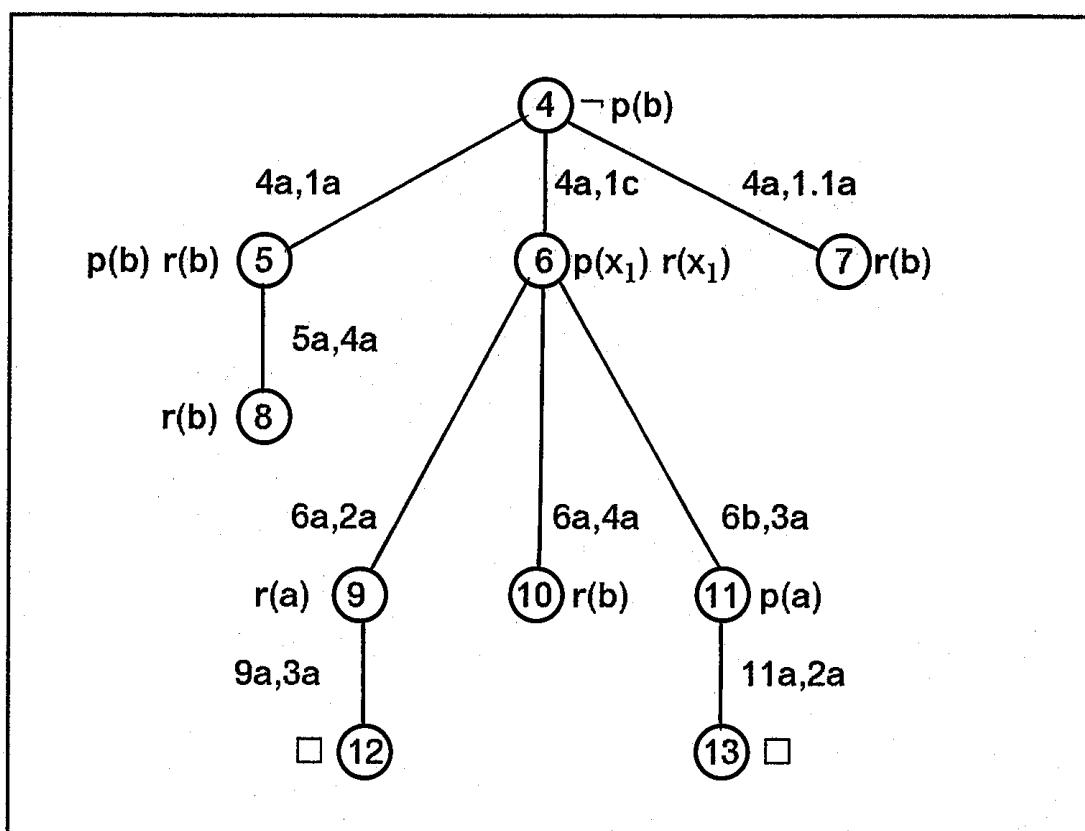
- (b) uma folha de A é um *nó de fracasso* de A se e somente se não é um nó de sucesso.
- (c) um *ramo de sucesso* de A é um ramo finito de A terminando em um nó de sucesso.
- (d) um *ramo de fracasso* de A é um ramo finito de A terminando em um nó de fracasso.

Exemplo 5.5:

Seja \mathbf{S} o seguinte conjunto de cláusulas (já incluindo o fator da primeira cláusula na linha 1.1 para efeitos de referência futura):

1. $p(x) \ r(x) \ p(b)$
- 1.1 $p(b) \ r(b)$
2. $\neg p(a)$
3. $\neg r(a)$
4. $\neg p(b)$

Uma árvore de RL-refutação para \mathbf{S} começando em (4) seria então:



Para facilitar o entendimento da árvore, numeramos os nós sequencialmente a partir de 4 e rotulamos as arestas de acordo com a mesma convenção usada para anotar as R-deduções. Por exemplo, a cláusula que rotula o nó 5 é a canonização do resolvente de (4) por (1) resultante da escolha do primeiro literal de (1).

Os ramos terminando em 12 e 13 são ramos de sucesso, enquanto que todos os outros ramos são de fracasso. Note que, ao ramo terminando em 12, corresponde a seguinte RL-refutação (mantendo a mesma numeração dos nós):

1. $p(x) \ r(x) \ p(b)$
2. $\neg p(a)$
3. $\neg r(a)$
4. $\neg p(b)$
6. $p(x_1) \ r(x_1)$. 4a, 1c
9. $r(a)$. 6a, 2a
12. \square . 9a, 3a

O mapeamento entre ramos de uma árvore de RL-refutação e RL-deduções pode ser explorado para se obter o seguinte resultado:

Teorema 5.3:

Um conjunto S de cláusulas é insatisfatível se e somente se existe uma árvore de RL-refutação A na floresta de RL-refutação para S tal que A tem um ramo de sucesso.

Demonstração

(\leftarrow) Suponha que exista uma cláusula C em S e uma árvore de RL-refutação A para S com cláusula inicial C tal que A tem um ramo de sucesso r . Seja D a seqüência de cláusulas rotulando o ramo r (da raiz para a folha de R). Então D induz uma RL-refutação a partir de S iniciando-se em C . Portanto, S é insatisfatível.

(\rightarrow) Suponha que S seja insatisfatível. Como o método da resolução linear é refutacionalmente completo, existe uma RL-refutação D a partir de S iniciando-se em uma cláusula C em S . Seja A a árvore de RL-refutação para S com cláusula inicial C . Seja D_1, \dots, D_p o prefixo de D . Logo, A possui por construção um ramo r rotulado com as canonizações das cláusulas

D_p, \dots, D_n , para todo i em $[p, n]$. Logo a folha de r é rotulada com $D_n = \square$. Assim, r é um ramo de sucesso.

Este resultado tem consequências importantes para a construção de procedimentos de refutação linear, que serão exploradas no resto desta seção.

Recorde da seção 5.2 que, dado um método de dedução M , para cada conjunto de sentenças S , o espaço de busca de S é o conjunto de todas as deduções a partir de S admissíveis em M . A demonstração do Teorema 5.3 sugere então que a floresta de RL-refutação para S , ou seja, o conjunto das árvores de RL-refutação começando em cada cláusula de S , é uma representação compacta e organizada do espaço de busca de S no método de resolução linear. Um procedimento de refutação linear reduz-se então a um algoritmo que recebe como entrada um conjunto S de cláusulas e progressivamente constrói a floresta de RL-refutação para S em busca de um nó de sucesso.

Em geral, um algoritmo de refutação linear combina uma estratégia básica para expansão das árvores da floresta de RL-refutação com uma heurística para seleção de cláusulas, fatores e literais. Há duas estratégias básicas para expansão de árvores, expansão em profundidade e expansão em amplitude, que são adaptações das formas básicas de pesquisa em árvores. Uma terceira estratégia de expansão pode ser imediatamente idealizada construindo a floresta de RL-refutação em profundidade até que cada árvore possua todos os nós até um nível pré-determinado, k , depois todos os nós até o nível $2k$ e assim por diante.

Uma estratégia de expansão das árvores da floresta, independentemente de ser em amplitude ou profundidade, depende da ordem com que são selecionadas:

- as cláusulas que rotularão as raízes das árvores;
- as cláusulas, fatores e literais que gerarão os filhos de cada nó.

É possível então definir heurísticas para governar estes dois processos de seleção e combiná-las com as estratégias básicas de expansão.

Por fim, recorde que o método da resolução linear mantém a completude se, dado um conjunto S de cláusulas e um conjunto de suporte T para S , nos limitarmos a considerar apenas as RL-refutações a partir de S cuja

cláusula inicial pertence a T . Como consequência, podemos eliminar imediatamente da floresta de RL-refutação todas as árvores cuja cláusula inicial não pertence a T . Portanto, este refinamento potencialmente evita a construção de um número considerável de árvores de RL-refutação e deve ser sempre considerado. Na verdade, se T possuir apenas uma cláusula, a floresta de RL-refutação se reduzirá a apenas uma árvore.

NOTAS BIBLIOGRÁFICAS

A noção de procedimento de dedução pode ser encontrada em Kowalski e Kuehner [1971]. O procedimento básico de resolução é chamado de procedimento de resolução por saturação em Loveland [1978] e em Chang e Lee [1973], que discutem também a noção de eliminação de tautologias e de cláusulas subjugadas. O método de resolução com conjunto de suporte foi proposto em Wos, Robinson e Carson [1965]. O método de refutação por resolução linear foi descoberto independentemente por Loveland [1970] e Luckham [1970]. Posteriormente Reiter [1971], Loveland [1972] e Kowalski e Kuehner [1971] desenvolveram novos resultados importantes. Chang e Lee [1973, Cap 7] e, principalmente, Loveland [1978, Cap. 3.5] contêm excelentes resumos sobre resolução linear. Um tratamento geral de conjuntos de suporte pode ser encontrado em Loveland [1978, Sec. 3.5].

CAPÍTULO 6: ELIMINAÇÃO DE MODELOS

Este capítulo introduz o sistema formal da eliminação de modelos e um método de refutação baseado nesta técnica. A seção 6.2 introduz a idéia básica de eliminação de modelos através de um exemplo. A seção 6.3 descreve o sistema formal da eliminação de modelos em detalhe, enquanto que a seção 6.4 define o método da eliminação de modelos fraca. Finalmente, a seção 6.5 discute brevemente como construir procedimentos de refutação por eliminação de modelos fraca.

Este capítulo constitui apenas leitura avançada.

6.1 INTRODUÇÃO

A técnica de eliminação de modelos, embora bastante semelhante a resolução, oferece uma alternativa interessante para a construção de sistemas para Programação em Lógica pois reduz consideravelmente o espaço de busca dos procedimentos de refutação. As seções 6.2 e 6.3 apresentam as idéias básicas desta técnica sob forma de um novo sistema formal.

Há vários métodos de refutação baseados no sistema formal da eliminação de modelos. As seções 6.4 e 6.5 apresentam o método de eliminação de modelos fraca. Este método é linear de entrada, não utiliza fatoração e, apesar destas restrições, mantém completude. Para isto usa um artifício: as cláusulas derivadas mantém os literais resolvidos, que são

posteriormente tratados por regras especiais. O método incorpora ainda uma forma restritiva de seleção de literais e um filtro para eliminar cláusulas derivadas que não são necessárias para a obtenção da cláusula vazia. O resultado são árvores de refutação mais compactas e, consequentemente, procedimentos de refutação mais eficientes.

A idéia genérica de eliminação de modelos é devida a Loveland, bem como vários dos métodos de refutação nela baseados. O método de resolução-SL de Kowalski e Kuener também é uma variação de eliminação de modelos.

6.2 UM EXEMPLO INICIAL

O método de refutação discutido neste capítulo, chamado de eliminação de modelos fraca, segue um formato semelhante ao método da resolução linear, porém com certas diferenças fundamentais. Primeiro, não trabalha com resolução propriamente dita, mas baseia-se no sistema formal da eliminação de modelos, *EM*, que possui as seguintes características:

- trabalha com seqüências de literais, chamadas de *cadeias*, em lugar de cláusulas;
- contém três regras de inferência - extensão, redução e contração - das quais apenas extensão é semelhante à regra da resolução;
- ao aplicar as regras, não é necessário fatorar as cadeias e a seleção de literais segue uma política rígida.

As deduções admissíveis pelo método da eliminação de modelos fraca possuem duas características principais:

- são sempre lineares de entrada;
- incorporam um processo de eliminação de resolventes bastante restritivo, mas que em alguns casos admite a geração de tautologias.

As três regras do sistema formal *EM* operam da seguinte forma.

A regra da extensão cria uma nova cadeia A a partir de uma cadeia A' e de uma cadeia auxiliar A'' da seguinte forma. Seja L' o literal mais à esquerda de A' . Seja L'' um literal de A'' tal que L' e L'' tem sinais opostos e os átomos de L' e L'' são unificáveis (se não existir um literal L'' em A'' com estas propriedades, a regra não é aplicável). Seja θ um u.m.g dos átomos de L' e L'' . A cadeia A é construída colocando-se os literais de

$A''\theta$ à esquerda dos literais de $A'\theta$, removendo $L''\theta$ do resultado, mantendo $L'\theta$ em A , mas marcando este literal como tendo sido resolvido. Assim, as cadeias derivadas passam a ter dois tipos de literais: normais e resolvidos (a regra também não se aplica se o primeiro literal de A' for um literal resolvido).

A regra da redução cria uma nova cadeia A a partir de uma cadeia A' da seguinte forma. Seja L' o literal mais à esquerda de A' . Seja M' um literal resolvido de A' tal que L' e M' tem sinais opostos e os átomos de L' e M' são unificáveis (se não existir um literal resolvido M' em A' com esta propriedade, ou se o primeiro literal de A' for um literal resolvido, a regra não se aplica). Seja θ um u.m.g dos átomos de L' e M' . A cadeia A será $A'\theta$ com o literal mais à esquerda removido.

A regra da contração cria uma nova cadeia A a partir de uma cadeia A' removendo repetidamente o elemento mais à esquerda de A' até que este seja um literal normal (se o elemento mais à esquerda de A' for um literal normal, a regra não se aplica).

As deduções admissíveis pelo método de eliminação de modelos fraca seguem um formato semelhante às deduções lineares de entrada. Dado um conjunto S de cadeias de entrada e uma cadeia $C \in S$, uma dedução a partir de S e iniciando-se em C tem a seguinte forma:

1. a dedução contém um prefixo, terminando na cadeia C , onde são listadas todas as cadeias de S usadas na dedução;
2. cada cadeia D_i da dedução, exceto as do prefixo, é obtida da cadeia D_{i-1} através da regra de
 - a. extensão, usando como cadeia auxiliar uma das cadeias do prefixo, ou
 - b. redução, ou
 - c. contração

Uma refutação é uma dedução terminando na cadeia vazia.

Terminaremos esta seção com um exemplo de uma refutação por eliminação de modelos.

Exemplo 6.1:

Considere o seguinte conjunto de cadeias de entrada:

- S1. $\neg p(u) \neg p(v)$
- S2. $p(x)$

Construiremos uma refutação a partir de S com cadeia inicial S2 da seguinte forma. Começaremos a refutação listando:

- 1. $\neg p(u) \neg p(v)$
- 2. $p(x)$

Aplicaremos extensão a (2) com (1) no papel de cadeia auxiliar, " $\neg p(u)$ " no papel de literal selecionado de (1) e $\theta = \{u/x\}$. O resultado será:

- 3. $\neg p(v) [p(x)]$

Os colchetes envolvendo o literal " $p(x)$ " indicam que ele é um literal resolvido de (3).

Aplicaremos agora redução a (3). De fato podemos fazê-lo pois esta cadeia possui um literal resolvido, " $p(x)$ ", cujo átomo é unificável com o átomo do literal mais à esquerda, " $\neg p(v)$ ". O resultado será:

- 4. $[p(x)]$

Finalmente, aplicando contração a (4), teremos:

- 5. \square

6.3 O SISTEMA FORMAL DA ELIMINAÇÃO DE MODELOS

O sistema formal da eliminação de modelos trabalha com seqüências de literais, resolvidos ou não, chamadas de cadeias para não confundir com as cláusulas usadas até o momento. Os colchetes esquerdo e direito, "[" e "]", agora incluídos nos alfabetos de primeira ordem, marcarão os literais resolvidos.

Definição 6.1:

Seja A um alfabeto de primeira ordem.

- (a) Um *literal resolvido* (ou um *R-literal*) sobre A é uma expressão da forma $[L]$, onde L é um literal.
- (b) Um *elemento* é um literal ou um R-literal.

Definição 6.2:

- (a) Uma *cadeia* sobre um alfabeto de primeira ordem A é ou uma seqüência não vazia de elementos sobre A ou a *cadeia vazia*, denotada por " \square ".
- (b) Uma cadeia C sobre A é *elementar* se e somente se C é a cadeia vazia ou é uma cadeia sem ocorrências de R-literais.

Assim " $p(x) \ q(x,y) \ r(y)$ " é uma cadeia elementar e " $p(f(z)) \ [q(f(z),a)] \ r(a)$ " é uma cadeia onde o segundo elemento é um R-literal.

Dada uma cadeia $C = E_1 \dots E_n$, E_1 é o elemento *mais à esquerda* de C , E_n é o elemento *mais à direita* de C , E_i está à *direita* de E_j em C se e somente se $i > j$, e E_i está à *esquerda* de E_j em C se e somente se $i < j$.

Definição 6.3:

A *linguagem das cadeias* sobre um alfabeto de primeira ordem A é o conjunto de todas as cadeias sobre A .

A semântica das linguagens de cadeias segue diretamente da semântica das linguagens de cláusulas estendendo a noção de satisfatibilidade a cadeias da seguinte forma:

Definição 6.4:

Uma estrutura I de um alfabeto de primeira ordem A *satisfaz* uma cadeia não-vazia C sobre A (denotado $I \models C$) se e somente se I satisfaz a cláusula D definida como " $L_1 \dots L_n$ ", onde L_1, \dots, L_n são os literais de C . Diz-se ainda que C e D são *equivalentes*.

Por convenção, a cadeia vazia é sempre insatisfatível.

Portanto, os R-literais de uma cadeia não influenciam a sua satisfatibilidade e todos os resultados do Capítulo 3 sobre a semântica de cláusulas valem também para cadeias. Por exemplo, a cadeia

" $p(f(z)) \ [q(f(z),a)] \ r(a)$ " é equivalente à cláusula " $p(f(z)) \ r(a)$ " e ambas são satisfatíveis se e somente se a sentença " $\forall z(p(f(z) \vee r(a))$ " o for.

As definições de substituição, unificador, unificador mais geral, etc... transferem-se sem modificações para cadeias. Em particular, a aplicação de uma substituição a uma cadeia afeta tanto literais quanto R-literais. Porém, por convenção, a aplicação de uma substituição a uma cadeia não eliminará elementos duplicados, diferentemente do caso das cláusulas. Por exemplo, se A for a cadeia " $p(y) \ [q(y)] \ \neg r(x,y) \ p(a)$ " e θ a substituição $\{y/a\}$, $A\theta$ será a cadeia " $p(a) \ [q(a)] \ \neg r(x,a) \ p(a)$ ".

As regras de inferência do sistema formal da eliminação de modelos baseiam-se nas definições auxiliares a seguir, onde $B'B''$ denota a concatenação de duas cadeias B' e B'' e $|L|$ indica a fórmula atômica F se L for o literal F ou o literal $\neg F$.

Definição 6.5:

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

- (i) L' e L'' tem sinais opostos e θ é um u.m.g de $\{|L'|, |L''\beta|\}$;
- (ii) $A = B''B'$, onde B'' é a cadeia $A''\beta\theta$ com o literal $L''\beta\theta$ removido e B' é a cadeia $A'\theta$ com o literal $L'\theta$ transformado em um R-literal.

Definição 6.6:

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

- (i) L' e M' tem sinais opostos e θ é um u.m.g de $\{|L'|, |M'|\}$;
- (ii) A é $A'\theta$ com o literal $L'\theta$ removido.

Definição 6.7:

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.

Exemplo 6.2:

Exemplos de extensão são (A é a extensão de A' por A''):

$$A': p(a) [q(x)]$$

$$A'': r(y) \neg p(y)$$

$$A : r(a) [p(a)] [q(x)]$$

$$A': p(x) [q(x)] r(x)$$

$$A'': \neg p(a)$$

$$A : [p(a)] [q(a)] r(a)$$

Exemplos de redução são (A é a redução de A'):

$$A': p(x) r(y) [\neg p(a)]$$

$$A : r(y) [\neg p(a)]$$

$$A': p(x) [q(x)] r(x) [\neg p(a)]$$

$$A : [q(a)] r(a) [\neg p(a)]$$

Exemplos de contração são (A é a contração de A'):

$$A': [q(x)] r(x) [\neg p(a)]$$

$$A : r(x) [\neg p(a)]$$

$$A': [p(x)] [r(y)] [\neg p(a)]$$

$$A : \square$$

Note que a extensão de A' por A'' não é equivalente à extensão de A'' por A' . Portanto, a ordem dos antecedentes na extensão é importante. Além disto, a extensão de A' por A'' não é única, pois há liberdade na escolha do literal de A'' e das substituições. Da mesma forma, a redução de A' não é única, pois há liberdade na escolha do R-literal de A' e das substituições.

Definição 6.8:

O sistema formal de eliminação de modelos, *EM*, consiste de:

Classe de Linguagens: linguagens das cadeias

Axiomas: nenhum

Regras de Inferência:

Extensão (EX)

EX: se A' e A'' são cadeias tais que
 o elemento mais à esquerda de A' é um literal e
 A é uma extensão de A' por A''
 derive A de A' e A''

Redução (RD)

RD: se A' é uma cadeia
 tal que o elemento mais à esquerda é um literal e
 A é uma redução de A'
 derive A de A'

Contração (CN)

CN: se A' é uma cadeia
 tal que o elemento mais à esquerda é um R-literal e
 A é a contração de A'
 derive A de A'

As definições de EM-dedução e de EM-refutação são semelhantes às definições correspondentes para o sistema formal da resolução, exceto que há três regras.

Exemplo 6.3:

A seguinte seqüência de cadeias é uma EM-refutação a partir das cadeias em (1) e (2):

1. $p(x) p(y)$
2. $\neg p(u) \neg p(v)$
3. $p(x) [\neg p(u)] \neg p(v)$. extensão de 2 por 1 usando o literal 1b
4. $[\neg p(u)] \neg p(v)$. redução de 3

- | | |
|-----------------------|--|
| 5. $\neg p(v)$ | . contração de 4 |
| 6. $p(x) [\neg p(v)]$ | . extensão de 5 de 1 usando literal o 1b |
| 7. $[\neg p(v)]$ | . redução de 6 |
| 8. \square | . contração de 7 |

Note que esta EM-refutação não utiliza fatores. Porém, toda refutação no sistema formal da resolução a partir de (1) e (2) exige o uso de fatores de (1) ou (2).

Os resultados abaixo mostram que o sistema formal da eliminação de modelos, como o sistema da resolução, é refutacionalmente correto e completo.

Lema 6.1:

Para todo conjunto S de cadeias, se existe uma EM-dedução a partir de S terminando em uma cadeia D então S implica logicamente D .

Demonstração

Seja S um conjunto de cadeias e suponha que existe uma EM-dedução $D = (D_1, \dots, D_n)$ a partir de S . Seja I uma estrutura satisfazendo S . Provaremos, por indução sobre $i \in [1, n]$, que I satisfaz D_i .

Base: Por definição de EM-dedução, D_1 pertence a S . Logo, I satisfaz D_1 .

Passo de Indução: Seja $i \in (1, n]$ e suponha que I satisfaça D_j , para todo $j < i$. Provaremos que I satisfaz D_i .

Caso 1: D_i é a contração de D_j , para $j < i$. Neste caso, D_i é idêntica a D_j sem os R-literais iniciais. Mas os R-literais não afetam a satisfatibilidade de uma cadeia. Logo, como I satisfaz D_j , I também satisfaz D_i .

Caso 2: D_i é uma extensão de D_j por D_k , para $j, k < i$. Pela hipótese de indução, I satisfaz D_j e D_k . Logo, de forma semelhante ao Lema da Correção para Resolução, I satisfaz D_i .

Caso 3: D_i é uma redução de D_j , para $j < i$. Suponha que D_j seja da forma $LM_1 \dots [M_p] \dots M_n$ e que $[M_p]$ foi o R-literal e θ o u.m.g usados na redução. Logo, θ é um u.m.g de $\{|L|, |M_p|\}$. Seja B a cadeia $M_p \dots M_n$.

Como $[M_p]$ é um R-literal de D_j , existe D_k , com $k < j$, tal que B é uma instância de D_k . Note agora que é possível estender D_j por B , selecionando o literal M_p de B para cancelar com o literal L de D e escolhendo θ como u.m.g.. A cadeia resultante será

$$B' = (M_{p+1} \dots M_n [L] M_1 \dots [M_p] M_{p+1} \dots M_n)\theta$$

Pela hipótese de indução, I satisfaz D_j e D_k . Logo, I também satisfaz B , pois B é uma instância de D_k . De forma semelhante ao caso 2, temos então que I satisfaz B' . Logo, I satisfaz $(M_1 \dots [M_p] \dots M_n)\theta$, que é a cadeia D_i . Isto conclui a indução.

Logo, para toda estrutura I satisfazendo S , para todo $i \in [1, n]$, I satisfaz D_i . Portanto, S implica logicamente D_i , para todo $i \in [1, n]$.

A partir deste lema, a correção do sistema formal da eliminação de modelos segue exatamente como para resolução.

Teorema 6.1: (Correção da Eliminação de Modelos)

Para todo conjunto S de cadeias, se existe uma EM-refutação a partir de S então S é insatisfatível.

Teorema 6.2: (Completude para Eliminação de Modelos)

Para todo conjunto S de cadeias, se S é insatisfatível, então existe uma EM-refutação a partir de S

(Para uma demonstração, veja Loveland [1969a]).

Portanto, um conjunto de cadeias S é insatisfatível se e somente se existir uma refutação a partir de S em EM .

6.4 O MÉTODO DA ELIMINAÇÃO DE MODELOS FRACA

6.4.1 Definição do Método

Definiremos o método da eliminação de modelos fraca como um método de dedução baseado no sistema formal EM cujo conjunto de deduções admissíveis incorpora quatro tipos de restrições:

- restrições sobre a seleção de cláusulas impondo o formato linear de entrada;
- restrições sobre a seleção de literais, já incorporadas na definição das regras de extensão, redução e contração;
- restrições sobre as cadeias que poderão servir de antecedentes das regras;
- restrições sobre as cadeias que poderão ser geradas como consequentes das regras.

A definição seguinte introduz as classes de cadeias admitidas como antecedentes e consequentes das regras (recorda que dois literais são complementares se e somente se forem da forma P e $\neg P$, para alguma fórmula atômica P).

Definição 6.9:

- (a) Uma cadeia é *preadmissível* se e somente se
- (i) literais complementares estão separados por um R-literal;
 - (ii) se um literal é idêntico a um R-literal, então o literal está à direita do R-literal;
 - (iii) dois R-literais não são complementares;
 - (iv) dois R-literais não são iguais.
- (b) Uma cadeia é *admissível* se e somente se é preadmissível e o elemento mais à esquerda é um literal.

Embora seja simples decidir se uma cadeia é preadmissível ou admissível, justificar estas restrições não é imediato pois elas estão intimamente ligadas à prova da completude do método. Voltaremos a este ponto na seção 6.4.2.

Exemplo 6.4:

TIPO DE CADEIA	EXEMPLOS
preadmissível	$p(x)$ $[q(x,y)]$ $\neg p(x)$ $[q(x,y)]$ $r(y)$

não preadmissível	$p(x) \neg p(x) [q(x,y)]$ $q(x,y) [q(x,y)]$
admissível	$p(x) [q(x,y)] \neg p(x)$ $r(y)$
não admissível	$p(x) \neg p(x) [q(x,y)]$ $[q(x,y)] r(y)$

Como indica o primeiro exemplo da lista acima, uma tautologia pode ser uma cadeia admissível desde que os literais complementares estejam separados por um R-literal.

Definição 6.10:

Seja **S** um conjunto de cadeias elementares, **B** uma cadeia em **S** e **C** uma cadeia qualquer. Uma *EM-dedução linear fraca* ou, simplesmente, uma *EMF-dedução* de **C** a partir de **S** e iniciando-se em **B** é uma seqüência **D** = (**D**₁, ..., **D**_n) de cadeias tal que:

- (i) $D_n = C$
- (ii) existe $r \leq n$ tal que D_1, \dots, D_r são cadeias em **S** e $D_r = B$. A subseqüência D_1, \dots, D_r é o *prefixo* de **D**.
- (iii) para todo $i \in [r+1, n]$, D_i é obtida de D_{i-1} através de uma das regras de inferência do sistema *EM*, se forem satisfeitas as seguintes condições adicionais:

Extensão: o primeiro antecedente, D_{i-1} , deve ser admissível;
o segundo antecedente deve ser da lista D_1, \dots, D_r ;
o resolvente, D_i , deve ser preadmissível.

Redução: o antecedente, D_{i-1} , deve ser admissível;
o resolvente, D_i , deve ser preadmissível.

Contração: o antecedente, D_{i-1} , deve ser preadmissível.

A cadeia D_{i-1} é chamada de *cadeia-pai* de D_i e, no caso de extensão, o segundo antecedente é chamado de *cadeia auxiliar* de D_i .

Note que as restrições sobre os antecedentes da regra de extensão ou da regra de redução podem naturalmente bloquear a aplicação da regra. Da mesma forma, a restrição sobre o consequente A da regra de extensão ou

da regra de redução pode bloquear a aplicação da regra. De fato, se A é uma extensão de uma cadeia admissível por uma cadeia elementar, então A pode não ser uma cadeia preadmissível. O mesmo acontece no caso em que A é uma redução de uma cadeia admissível.

Exemplo 6.5:

(a) Exemplos de bloqueio de extensão (A é a extensão de A' por A''):

$$A': p(x) \neg p(x) [q(x,y)]$$

$$A'': \neg p(a)$$

(a regra não se aplica pois A' não é admissível)

$$A': p(x) [q(x)]$$

$$A'': \neg p(a) q(a)$$

$$A : q(a) [p(a)] [q(a)]$$

(a regra não se aplica pois A não é preadmissível)

(b) Exemplos de bloqueio de redução (A é a redução de A'):

$$A': p(x) \neg p(x) [\neg p(y)]$$

(a regra não se aplica pois A' não é admissível)

$$A': p(x) q(x) [q(a)] [\neg p(a)]$$

$$A : q(a) [q(a)] [\neg p(a)]$$

(a regra não se aplica pois A não é preadmissível)

Definição 6.11:

Seja S um conjunto de cadeias elementares e B uma cadeia em S . Uma *EM-refutação linear fraca* ou, simplesmente, uma EMF-refutação a partir de S e iniciando-se em B é uma EMF-dedução da cadeia vazia a partir de S e iniciando-se em B .

As anotações após cada cadeia D_i de uma EMF-dedução (D_1, \dots, D_n) seguirão a seguinte convenção:

- a abreviação CN indica que D_i foi obtida por contração de D_{i-1} ;
- a abreviação RD, seguida de uma letra l , indica que D_i foi obtida por redução de D_{i-1} e que o literal resolvido escolhido é o de ordem l em D_{i-1} ("a" para o primeiro, "b" para o segundo, etc.);

- um número j , seguido de uma letra l , indica que D_i for obtida estendendo-se D_{i-1} pela cadeia D_j e que o literal escolhido é o de ordem l em D_j .

Exemplo 6.6:

Considere o seguinte conjunto de cadeias elementares de entrada:

$$\{p(a) \ q(a), p(x) \neg q(a), \neg p(x) \ r(a,x), \neg p(x) \neg r(x,a)\}$$

A seguinte seqüência é uma EMF-refutação a partir destas cadeias iniciando-se em "p(a) q(a)"

1. $p(x) \neg q(a)$
2. $\neg p(x) \ r(a,x)$
3. $\neg p(x) \ \neg r(x,a)$
4. $p(a) \ q(a)$
5. $\neg r(a,a) [p(a)] q(a)$. 3a
6. $\neg p(a) [\neg r(a,a)] [p(a)] q(a)$. 2b
7. $[\neg r(a,a)] [p(a)] q(a)$. RD c
8. $q(a)$. CN
9. $p(x) [q(a)]$. 1b
10. $r(a,x) [p(x)] [q(a)]$. 2a
11. $\neg p(a) [r(a,a)] [p(a)] [q(a)]$. 3b
12. $[r(a,a)] [p(a)] [q(a)]$. RD c
13. \square . CN

Definição 6.12:

O método da eliminação de modelos fraca consiste do par $EMb = (EM, EMFD)$, onde EM é o sistema formal da eliminação de modelos e $EMFD$ é o conjunto das EMF-deduções.

Como o método da eliminação de modelos fraca baseia-se no sistema formal da eliminação de modelos, ele é refutacionalmente correto. O teorema a seguir afirma que, apesar das severas restrições sobre as deduções admissíveis, o método também é refutacionalmente completo.

Teorema 6.3: (Completude da Eliminação de Modelos Fraca)

Para todo conjunto \mathbf{S} de cadeias, se \mathbf{S} é insatisfável, então existe uma cadeia C em \mathbf{S} e existe uma EMF-refutação a partir de \mathbf{S} e iniciando-se em C .

(Para uma demonstração, veja Loveland [1969a]).

Podemos formular um resultado um pouco mais preciso da seguinte forma.

Teorema 6.4:

Se \mathbf{S} é um conjunto insatisfável de cadeias e C é uma cadeia em \mathbf{S} tal que $\mathbf{S} - \{C\}$ é satisfável, então existe uma EMF-refutação a partir de \mathbf{S} e iniciando-se em C .

Para completar esta seção, mencionaremos um outro método de eliminação de modelos que resume a essência do método de resolução-SL de Kowalski e Kuehner [1971].

Seja D uma EMF-dedução e D_i uma cadeia admissível derivada em D . A *célula inicial* de D_i consiste da seqüência de literais à esquerda do R-literal mais à esquerda em D_i . Pela forma com que EMF-deduções são construídas, todos os literais da célula inicial de D_i originam-se de uma mesma cadeia de entrada E e a sua ordem apenas reflete a ordem dos literais em E . Mas a ordem dos literais em uma cadeia de entrada é escolhida arbitrariamente. Logo, uma reordenação da célula inicial de uma dada cadeia derivada não afetará a completude do método. Por outro lado, uma tal reordenação determinará um novo literal para ocupar a posição mais à esquerda da cadeia e, portanto, determinará um novo literal a ser selecionado pelas regras de inferência.

Assim, o método da eliminação de modelos pode ser dotado de uma *função de seleção* que, para cada cadeia admissível C , reordena a célula inicial de C selecionando um novo literal para a posição mais à esquerda e influenciando assim o desenvolvimento das EMF-deduções. A função de seleção não precisa ser definida a priori, podendo ser determinada à medida que a EMF-dedução se desenvolver (veja o exemplo no final da seção 6.5).

6.4.2 Resumo das Principais Características do Método

Esta seção explica e organiza as características mais importantes do método, que até agora permaneceram embutidas na descrição das regras e na definição de EMF-dedução (ver a figura no final desta seção para um resumo das características).

(A) CONJUNTO DE ENTRADA

O conjunto de entrada conterá sempre cadeias elementares obtidas ordenando-se os literais das cláusulas que representam o domínio de discurso. A ordenação é livre e pode ser usada como um instrumento para guiar o procedimento de refutação.

(B) ESTRATÉGIA DE SELEÇÃO DE CADEIAS

A estratégia é linear de entrada, ou seja, cada resolvente é obtido do resolvente imediatamente anterior por redução, contração ou extensão, usando neste último caso uma cadeia auxiliar tirada do conjunto de entrada.

(C) ESTRATÉGIA DE FATORAÇÃO

A manutenção dos literais resolvidos na cadeia derivada torna desnecessário utilizar fatores tanto da cadeia-pai quanto da cadeia auxiliar, se houver (veja o Exemplo 6.3).

(D) ESTRATÉGIA DE SELEÇÃO DE LITERAIS

A estratégia de seleção de literais dita que:

- a escolha do literal da cadeia-pai deve sempre recair sobre o elemento mais à esquerda;
- a escolha do literal da cadeia auxiliar (para extensão) é livre.

(E) RESTRIÇÕES SOBRE A APLICAÇÃO DAS REGRAS

(E.1) Restrições sobre os Antecedentes

A definição de EMF-dedução impõe as seguintes restrições aos antecedentes:

1. os antecedentes da regra de extensão são uma cadeia admissível e uma cadeia elementar;
2. o antecedente da regra de redução é uma cadeia admissível;
3. o antecedente da regra de contração é uma cadeia preadmissível.

Portanto, sempre que um resolvente recém-gerado for uma cadeia preadmissível, mas não admissível, apenas contração pode ser aplicada. Como consequência, contração pode e deve ser combinada com extensão e redução.

(E.2) Restrições sobre os Consequentes

Conforme observado anteriormente, a restrição de que o consequente das regras de extensão e redução deve ser uma cadeia preadmissível pode também bloquear a aplicação da regra. Esta restrição age como um "filtro" eliminando cadeias que não contribuem para a obtenção da cadeia vazia no método de eliminação de modelos.

Em detalhe, o efeito de cada uma das condições da definição de preadmissibilidade é o seguinte.

A condição (i), "literais complementares estão separados por um R-literal", evita que tautologias sejam usadas como a cadeia auxiliar em uma aplicação da regra de extensão. De fato, suponha que A'' seja uma tautologia usada como cadeia auxiliar na derivação de A pela regra da extensão. Então, como A'' contém dois literais complementares, A conterá dois literais complementares à esquerda do seu R-literal mais à esquerda, por definição de extensão. Portanto, A não será admissível e nunca será realmente gerada por extensão. Mas isto é exatamente desejado pois A'' , por ser uma tautologia, nunca contribuirá para obter uma contradição.

As condições (ii) e (iii) são puramente técnicas e só podem ser entendida estudando-se a prova da completude do método. Ambas bloqueiam deduções com certas características especiais.

A condição (ii), "um literal idêntico a um R-literal está à direita do R-literal", bloqueia a extensão de uma cadeia-pai A' por uma cadeia elementar A'' sempre que A' contiver um R-literal L' e A'' contiver um literal L'' tais que L' e L'' se tornem iguais após a unificação usada na extensão.

A condição (iii), "dois R-literais não são complementares", força a redução da última cadeia derivada, A, se o elemento mais à esquerda de A for um literal complementar a um R-literal de A. De fato, contração não pode ser aplicada por definição e, se extensão fosse aplicada a A, a cadeia resultante conteria dois R-literais complementares e, portanto, violaria a condição (iii). Porém, tirando este caso, a redução da última cadeia derivada não é obrigatória. A próxima seção contém, na verdade, um exemplo em que a aplicação obrigatória de redução tornaria o método incompleto.

A condição (iv), "dois R-literais não são iguais", é consequência direta da condição (ii) e da forma como literais são transformados em R-literais. De fato, por (ii) não há um literal L e um R-literal L' em uma cadeia derivada A tais que L e L' são idênticos e L está à esquerda de L'. Logo não há como gerar a partir de A uma cadeia onde L e L' são ambos R-literais.

6.5 PROCEDIMENTOS DE REFUTAÇÃO POR ELIMINAÇÃO DE MODELOS FRACA

Da mesma forma que os procedimentos de refutação baseados no método de resolução linear, os procedimentos baseados em eliminação de modelos devem ser entendidos como algoritmos para construção de florestas de refutação. Esta seção se restringe então a definir a classe das árvores de refutação para eliminação de modelos fraca.

Convém inicialmente definir duas novas operações, extensão plena e redução plena, que combinam contração com extensão e redução, conforme sugerido no item D.1 da seção 6.4.2, e incorporam também certas restrições impostas pelo método.

Definição 6.13:

- (a) Uma cadeia A é a *extensão plena* de uma cadeia admissível A' por uma cadeia elementar A'' se e somente se A é a contração da extensão de A' por A'' e A é admissível.
- (b) Uma cadeia A é a *redução plena* de uma cadeia admissível A' se e somente se A é a contração da redução de A' e A é admissível.

Definição 6.14:

Uma cadeia A é uma *canonização* de uma cadeia A' se e somente se A é a variante canônica de A'.

Principais Características do Método de Eliminação de Modelos Fraca

cadeia-pai:

- sempre a cadeia anteriormente derivada (linear)
- sem fatoração
- literal escolhido é sempre o mais à esquerda

cadeia auxiliar (para extensão):

- sempre uma cadeia de entrada (linear de entrada)
- sem fatoração
- literal escolhido não é fixado à priori

restrições sobre os antecedentes das regras:

- os antecedentes da regra de extensão são uma cadeia admissível e uma cadeia elementar;
- o antecedente da regra de redução é uma cadeia admissível;
- o antecedente da regra de contração é uma cadeia preadmissível;

Logo, durante uma EMF-dedução, se uma cadeia preadmissível D_i for gerada por uma aplicação da regra de extensão ou redução, a EMF-dedução só poderá continuar aplicando-se a regra de contração a D_i .

restrições sobre os consequentes das regras:

O consequente das regras de extensão e redução deverá ser uma cadeia preadmissível. Entre outras, esta restrição tem as seguintes consequências:

- tautologias não são usadas como cadeias auxiliares na aplicação da regra de extensão;
- se um resolvente tiver um R-literal complementar ao seu primeiro literal, a regra de redução deve ser a ele aplicada.

Definição 6.15:

Uma cadeia A é um *conseqüente canônico* de uma cadeia admissível A' *em presença de* um conjunto de cadeias elementares S se e somente se A é a canonização de uma redução plena de A' , ou A é a canonização de uma extensão plena de A' por uma cadeia em S .

Assim, a definição de conseqüente canônico, via a própria definição de extensão e redução plena, já incorpora a eliminação de cadeias não admissíveis.

De posse destas noções complementares podemos definir então a noção de uma árvore de refutação para eliminação de modelos fraca, ou árvore de EMF-refutação, de forma semelhante às árvores para resolução linear, mas levando em consideração as restrições adicionais inerentes ao método de eliminação de modelos, principalmente o fato do método ser linear de entrada e filtrar cadeias que não sejam admissíveis.

Definição 6.16:

Sejam S um conjunto de cadeias elementares e C uma cadeia em S . Uma árvore, com os nós rotulados por cadeias, é uma *árvore de refutação por eliminação de modelos fraca* ou, simplesmente, uma *árvore de EMF-refutação* para S com cadeia inicial C se e somente se:

- (i) o rótulo da raiz é a canonização de C ;
- (ii) para cada nó N , rotulado com uma cadeia A ,
 - para cada conseqüente canônico A' de A em presença de S , existe exatamente um filho de N rotulado com A' , e
 - todos os filhos de N são rotulados com conseqüentes canônicos de A em presença de S .

As noções de floresta de EMF-refutação, nó de sucesso, nó de fracasso, ramo de sucesso, ramo de fracasso seguem da mesma forma que para resolução linear.

À semelhança do método de resolução linear, podemos provar o seguinte resultado:

Teorema 6.5:

Um conjunto \mathbf{S} de cadeias elementares é insatisfatível se e somente se existe uma árvore A na floresta de EMF-refutação para \mathbf{S} tal que A tem um ramo de sucesso.

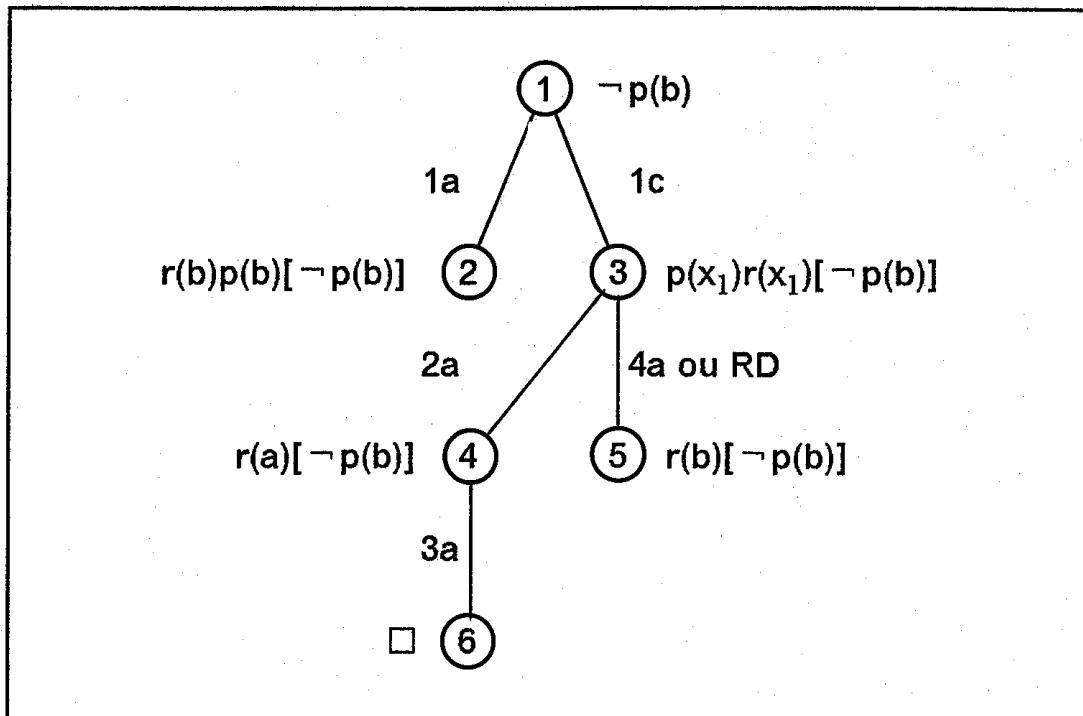
Portanto, como no caso do método da resolução linear, um procedimento de refutação baseado no método de eliminação de modelos fraca reduz-se a um algoritmo que recebe como entrada um conjunto \mathbf{S} de cadeias elementares e progressivamente constrói a floresta de EMF-refutação para \mathbf{S} em busca de um nó de sucesso. Porém, procedimentos de refutação baseados no método de eliminação de modelos serão em geral mais eficientes pois as florestas de EMF-refutação tenderão a ser menores do que as florestas para resolução linear. De fato, as restrições impostas aos antecedentes e consequentes das regras, a seleção do mesmo literal em todas as deduções a partir da mesma cadeia, a inexistência de fatoração e o uso de cadeias auxiliares retiradas apenas do conjunto de entrada reduzem o tamanho da árvore de EMF-refutação, em comparação com o tamanho da árvore equivalente para resolução linear. O exemplo a seguir, que deve ser comparado com o Exemplo 5.5, ilustra este ponto.

Exemplo 6.7:

Seja \mathbf{S} o seguinte conjunto de cadeias elementares:

1. $p(x) \ r(x) \ p(b)$
2. $\neg p(a)$
3. $\neg r(a)$
4. $\neg p(b)$

Uma árvore de EMF-refutação para \mathbf{S} começando em (4) seria então:



Para facilitar o entendimento da árvore, rotulamos as arestas de acordo com a mesma convenção usada para anotar as derivações. Em detalhe temos, onde R_i denota a cadeia que rotula o nó i :

1. R_1 é a cadeia inicial (4).
2. R_2 é a extensão plena de (4) por (1), escolhendo o primeiro literal de (1).
3. R_3 é a canonização da extensão plena de (4) também por (1), mas escolhendo o terceiro literal de (1).
4. R_4 é a extensão plena de R_3 por (2).
5. R_5 é a extensão plena de R_3 por (4), ou a redução plena de R_3 .
6. R_6 é a extensão plena de R_4 por (3).

Note que a redução de R_3 não leva à derivação da cadeia vazia. Portanto não podemos forçar a aplicação única e exclusiva da regra da redução sempre que possível. De fato, a cadeia R_3 possui uma extensão que leva à cadeia vazia.

Mais precisamente, os ramos terminando em 2 e 5 são ramos de fracasso e o ramo terminando em 6 é um ramo de sucesso, pois este é

o único ramo terminando em um nó rotulado com a cadeia vazia. A este ramo corresponde a seguinte EMF-refutação

1. $p(x) \ r(x) \ p(b)$
2. $\neg p(a)$
3. $\neg r(a)$
4. $\neg p(b)$
5. $p(x_1) \ r(x_1) [\neg p(b)]$. 1c
6. $r(a) [\neg p(b)]$. 2a
7. \square . 3a

Note que estamos usando extensão plena aqui. Logo, as derivações nas linhas (6) e (7) contêm embutidas uma aplicação da regra de contração.

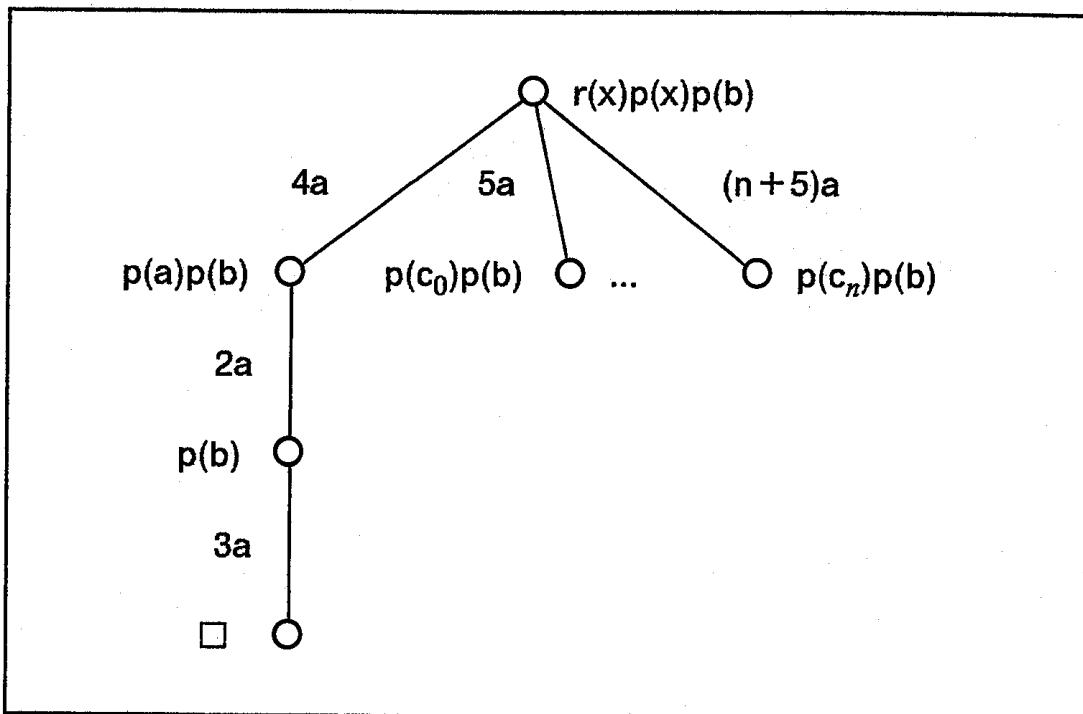
Exemplo 6.8:

Este exemplo ilustra, de forma simples, como o uso de funções de seleção para reordenar os literais da célula inicial de uma cadeia pode reduzir o tamanho de uma árvore de EMF-refutação.

Seja **S** o seguinte conjunto de cadeias elementares:

1. $r(x) \ p(x) \ p(b)$
2. $\neg p(a)$
3. $\neg p(b)$
4. $\neg r(a)$
5. $\neg r(c_0)$
6. $\neg r(c_1)$
- ...
- $n+5. \ \neg r(c_n)$

Uma árvore de EMF-refutação para **S** começando em (1) seria:



Considere agora a função de seleção f definida informalmente como:

$$f(C) = C' \text{ se e somente se}$$

C' é idêntica a C , exceto que a posição mais à esquerda de C' é ocupada pelo literal pertencente à célula inicial de C que pode ser unificado contra o menor número de literais pertencentes a cadeias de entrada (se houver mais de um literal com esta propriedade, escolhe-se o mais à esquerda em C).

Note que esta função, a rigor, deveria ter como parâmetro também o conjunto de cadeias de entrada. Note ainda que, em uma definição mais realista da função, a escolha do literal para ocupar a posição mais à esquerda seria feita por estimativa, baseada em estatísticas sobre o conjunto de entrada.

Por exemplo, para o conjunto de entrada S definido acima, temos que:

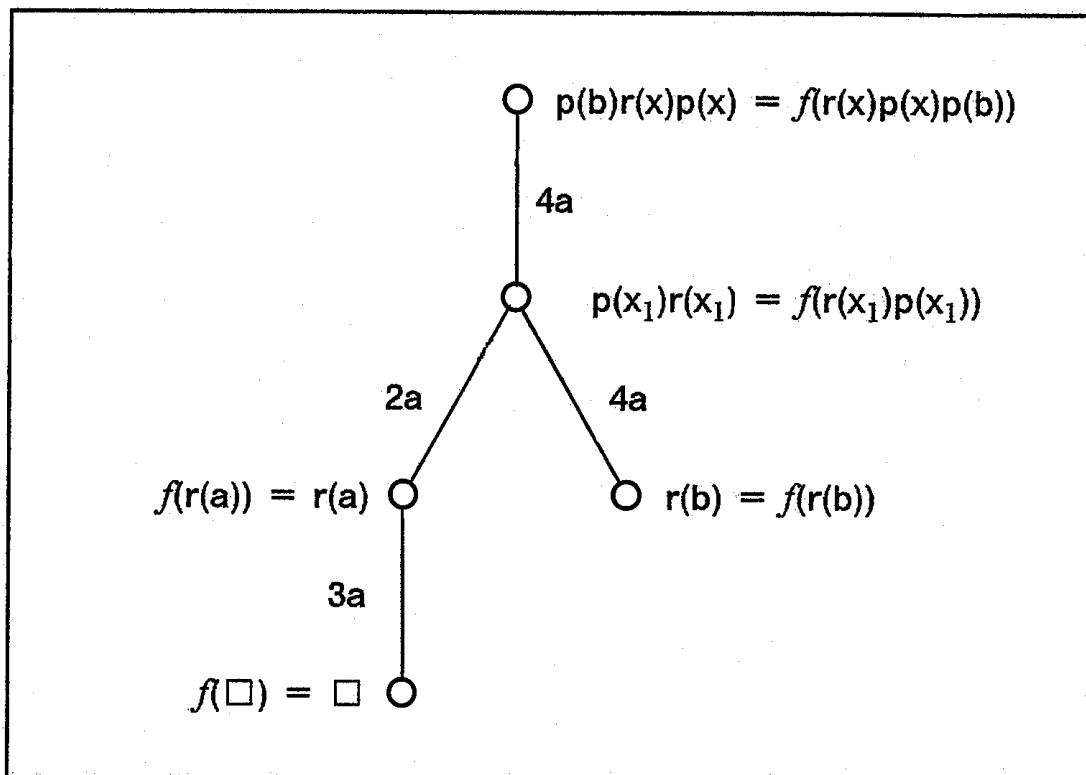
$$f(r(x)p(x)p(b)) = p(b)p(x)r(x)$$

$$f(r(x)p(x)) = p(x)r(x)$$

O primeiro exemplo segue observando-se que a célula inicial da cadeia "r(x) p(x) p(b)" é ela própria e que "p(b)" só pode ser unificado contra um único literal, o literal da cadeia (3), enquanto que "p(x)" pode ser

unificado contra dois literais, os literais das cadeias (2) e (3), e " $r(x)$ " pode ser unificado contra $n+2$ literais, os literais das cadeias (4) em diante. O segundo exemplo segue de forma semelhante.

Assuma agora que, antes de expandir os filhos de cada nó da árvore, inclusive a raiz, a cadeia que o rotula é transformada pela função de seleção f . A árvore de EMF-refutação começando em (1) seria então a seguinte:



Note que a função f altera os rótulos da raiz e do segundo nó, levando à construção de uma árvore mais compacta.

NOTAS BIBLIOGRÁFICAS

O método de eliminação de modelos fraca foi introduzido em Loveland [1968] e posteriormente elaborado em Loveland [1969a,1969b,1978 Sec.3.6]. Uma descrição integrada com outros métodos de refutação lineares encontra-se em Loveland [1972,1978]. Uma implementação é descrita em Fleisig [1974]. Uma proposta para implementar uma extensão de Prolog, baseada em eliminação de modelos, encontra-se em Stickel [1985].

CAPÍTULO 7: RESOLUÇÃO-LSD E NEGAÇÃO POR FALHA FINITA

Este capítulo apresenta dois novos métodos de refutação que formam a base da linguagem Prolog. As seções 7.2 a 7.4 discutem o método de resolução linear com função de seleção para cláusulas definidas ou, brevemente, resolução-LSD. As seções 7.5 e 7.6 apresentam uma extensão deste método que incorpora a regra da negação por falha finita.

As seções recomendadas para cada nível de leitura são:

nível introdutório: 7.2 e 7.4

nível intermediário: 7.2, 7.4 e 7.5

7.1 INTRODUÇÃO

Este capítulo apresenta inicialmente um novo método de refutação, chamado de resolução linear com função de seleção para cláusulas definidas, ou resolução-LSD. Este método muito contribuiu para o sucesso de Prolog por oferecer para esta linguagem tanto uma base teórica quanto condições para a implementação de interpretadores extremamente eficientes. Apesar do nome, resolução-LSD deve ser visto como um método de refutação por eliminação de modelos trabalhando com uma determinada classe de cadeias. De fato, o principal resultado da seção 7.3,

a completude do método, segue como corolário da completude de eliminação de modelos fraca.

Porém, resolução-LSD impõe limitações importantes no conjunto de entrada. Por exemplo, uma simples cláusula da forma " $p(a) \ p(b)$ " não pode pertencer a um conjunto de entrada para resolução-LSD. Este capítulo explora então muito brevemente uma extensão de resolução-LSD que admite conjuntos de entrada mais gerais, mas mantém parte da simplicidade do método original. A extensão considerada trata a negação como "impossibilidade de prova", ou seja, ela assume $\neg P$, em presença de um dado conjunto de cadeias S , se não for possível provar P a partir de S de uma forma finitária. Este tratamento é chamado de negação por falha finita.

O método da resolução-LSD é um caso particular de resolução linear com função de seleção, ou resolução-LS, devida a Kowalski e Kuehner [1971]. Ele foi introduzido com este nome em Apt [1982] e investigado com o nome de resolução-LUSH em Hill [1974]. O tratamento da negação por falha finita deve-se a Clark [1978].

7.2 O MÉTODO DA RESOLUÇÃO-LSD

Resolução-LSD trabalha com conjuntos consistindo de uma ou mais cláusulas com exatamente um literal positivo e de uma cláusula apenas com literais negativos. Embora o método pudesse ser formulado em termos de cláusulas na notação usual, tornou-se comum na literatura introduzir uma notação especial, capturada através de uma nova linguagem formal.

Novamente modificaremos a noção de alfabeto de primeira ordem para incluir também o símbolo " \leftarrow " no papel de um novo conectivo.

Definição 7.1:

Seja A um alfabeto de primeira ordem.

- (a) Uma *cláusula definida* sobre A é uma expressão da forma " $L \leftarrow M_1 \dots M_n$ " ou da forma " $L \leftarrow$ ", onde L, M_1, \dots, M_n são literais positivos sobre A . O literal L é a *cabeça* e os literais M_1, \dots, M_n formam o *corpo* da cláusula.

- (b) Uma *cláusula-objetivo* sobre A é ou a cláusula vazia ou é uma expressão da forma " $\leftarrow M_1 \dots M_n$ ", onde M_1, \dots, M_n são literais positivos sobre A . Estes literais formam o *corpo* da cláusula.
- (c) Uma *cláusula de Horn* sobre A é ou uma cláusula definida ou uma cláusula-objetivo sobre A .
- (d) Um *conjunto quase-definido* é um conjunto de cláusulas definidas acrescido de exatamente uma cláusula-objetivo.

Por exemplo, " $p(x,a) \leftarrow$ ", " $p(x) \leftarrow q(y)$ " e " $p(x) \leftarrow r(x,y)q(x,y)$ " são cláusulas definidas e " \square " e " $\leftarrow r(x,y)p(x)$ " são cláusulas-objetivo.

Definição 7.2:

A *linguagem das cláusulas de Horn* sobre um alfabeto de primeira ordem A é o conjunto de todas as cláusulas de Horn sobre A .

A Definição 7.1 traduz a notação favorecida na literatura mais recente, embora tradicionalmente uma cláusula de Horn seja definida como qualquer cláusula com no máximo um literal positivo. Estas duas definições são inteiramente equivalentes pois a semântica das linguagens das cláusulas de Horn é definida da seguinte forma:

Definição 7.3:

Seja A um alfabeto de primeira ordem e seja I uma estrutura para A .

- (a) I satisfaçõa uma cláusula definida " $L \leftarrow M_1 \dots M_n$ " sobre A se e somente se I satisfaçõa a cláusula " $L \neg M_1 \dots \neg M_n$ ". Diz-se ainda que as duas cláusulas são *equivalentes*.
- (b) I satisfaçõa uma cláusula-objetivo não-vazia " $\leftarrow M_1 \dots M_n$ " sobre A se e somente se I satisfaçõa a cláusula " $\neg M_1 \dots \neg M_n$ ". Diz-se ainda que as duas cláusulas são *equivalentes*.
- (c) I não satisfaçõa a cláusula vazia.

A própria sintaxe sugere porém uma segunda leitura para a semântica das cláusulas de Horn em que uma cláusula definida representa uma implicação e uma cláusula-objetivo define a negação de uma conjunção. De fato, uma cláusula definida " $L \leftarrow M_1 \dots M_n$ " representa a fórmula $\forall(L \vee \neg M_1 \vee \dots \vee \neg M_n)$, que é equivalente à implicação $\forall(M_1 \wedge \dots \wedge M_n \rightarrow L)$. Semelhantemente, uma cláusula-objetivo

" $\leftarrow M_1 \dots M_n$ " representa a fórmula $\forall(\neg M_1 \vee \dots \vee \neg M_n)$, que é equivalente a $\neg\exists(M_1 \wedge \dots \wedge M_n)$.

A restrição a conjuntos de entrada quase-definidos simplifica consideravelmente a aplicação do método de eliminação de modelos pois os literais resolvidos não são mais necessários para atingir completude. Consequentemente, as regras de redução e contração, que operam sobre tais literais, podem ser descartadas e a regra de extensão não mais precisa manter no resolvente o literal resolvido. Estas propriedades permitem ainda generalizar, de forma bem simples, a política de seleção do literal da cláusula-pai na aplicação da regra da extensão. Esta generalização leva ao conceito de função de seleção para cláusulas-objetivo definido da seguinte forma:

Definição 7.4:

- (a) Uma função f é uma *função de seleção* para cláusulas-objetivo, ou simplesmente uma *função de seleção*, se e somente se f mapeia cada cláusula-objetivo C em um literal de C .
- (b) Uma função f é a *função de seleção padrão* se e somente se f mapeia cada cláusula-objetivo C no literal mais à esquerda de C .

As simplificações mencionadas acima, junto com a noção de função de seleção, se traduzem em um novo sistema formal baseado em apenas uma regra de inferência. Esta regra captura a noção de f -extensão, introduzida da seguinte forma:

Definição 7.5:

Seja f uma função de seleção. Sejam A' uma cláusula-objetivo e A'' uma cláusula definida. Seja β uma renomeação de A'' em presença de A' . Suponha que A' seja da forma " $\leftarrow N_1 \dots N_m$ ", que $A''\beta$ seja da forma " $L \leftarrow M_1 \dots M_k$ " e que $f(\leftarrow N_1 \dots N_m) = N_i$. Uma cláusula-objetivo A é uma *extensão governada por f* ou, simplesmente, uma *f -extensão* de A' por A'' se e somente se existe um u.m.g θ do conjunto de literais $\{L, N_i\}$ tal que A é da forma " $\leftarrow(N_1 \dots N_{i-1} M_1 \dots M_k N_{i+1} \dots N_m)\theta$ ".

Note que

- Não há liberdade na escolha dos literais pois o literal escolhido de A' é fixado pela função f e o literal escolhido de A'' é sempre a cabeça.

- A cláusula resultante é uma cláusula-objetivo. Portanto, qualquer dedução utilizando apenas esta regra de inferência terá apenas cláusulas-objetivo como cláusulas derivadas.
- Se A e B são f-extensões de A' por A'' , então A e B são variantes entre si. Portanto, a menos de renomeação de variáveis, a f-extensão de A' por A'' é única.
- A noção de f-extensão é de fato uma especialização da noção de extensão pois A' é equivalente à cadeia " $\neg N_1 \dots \neg N_m$ " e A'' é equivalente à cadeia " $L \neg M_1 \dots \neg M_m$ ".

Definição 7.6:

Seja f uma função de seleção. O sistema formal da *resolução com função de seleção f para conjuntos quase-definidos*, $RSD(f)$, consiste de:

Classe de Linguagens: linguagens das cláusulas de Horn

Axiomas: nenhum

Regra de Inferência: f-Extensão (EXS):

EXS: se A' é uma cláusula-objetivo,
 A'' é uma cláusula definida e
 A é uma f-extensão de A' por A''
 derive A de A' e A''

A noção de EMF-refutação especializa-se por sua vez da seguinte forma.

Definição 7.7:

Seja f uma função de seleção. Sejam S um conjunto quase-definido e C uma cláusula-objetivo. Uma *LSD(f)-dedução* de C a partir de S é uma seqüência $D = (D_1, \dots, D_n)$ de cláusulas tal que $D_n = C$ e existe $r \leq n$ tal que

- (i) para todo $i < r$, D_i é uma cláusula definida em S ;
- (ii) D_r é a cláusula-objetivo de S ;

(iii) para todo $i > r$, D_i é uma f-extensão de D_{i-1} por cláusula definida D_j , para algum $j < r$. A cláusula D_{i-1} é chamada de *cláusula-pai* de D_i e D_j é chamada de *cláusula auxiliar* de D_i .

A subseqüência D_1, \dots, D_r é o *prefixo* de D .

Definição 7.8:

Sejam f uma função de seleção e S um conjunto quase-definido. Uma *LSD(f)-refutação* a partir de S é uma LSD(f)-dedução da cláusula vazia a partir de S .

Exemplo 7.1:

Seja f a função de seleção que mapeia cada cláusula no seu primeiro literal. A seguinte seqüência de cláusulas é uma LSD(f)-refutação a partir das cláusulas (1) a (6):

1. *chama(a,b)* \leftarrow
2. *usa(b,e)* \leftarrow
3. *depende(x,y)* \leftarrow *chama(x,y)*
4. *depende(x,y)* \leftarrow *usa(x,y)*
5. *depende(x,y)* \leftarrow *chama(x,z)* *depende(z,y)*
6. \leftarrow *depende(a,e)*
7. \leftarrow *chama(a,z)* *depende(z,e)* . 5
8. \leftarrow *depende(b,e)* . 1
9. \leftarrow *usa(b,e)* . 4
10. \square . 2

Note que "*chama(a,z)*" foi o literal selecionado da cláusula em (7). Note ainda que basta indicar em cada linha a cláusula definida D usada na f-extensão, pois o literal selecionado é sempre a cabeça de D .

Seja g a função de seleção que mapeia cada cláusula no seu último literal. A seguinte seqüência de cláusulas é então uma LSD(g)-refutação a partir das cláusulas (1) a (6):

1. $chama(a,b) \leftarrow$
2. $usa(b,e) \leftarrow$
3. $depende(x,y) \leftarrow chama(x,y)$
4. $depende(x,y) \leftarrow usa(x,y)$
5. $depende(x,y) \leftarrow chama(x,z) depende(z,y)$
6. $\leftarrow depende(a,e)$
7. $\leftarrow chama(a,z) depende(z,e)$. 5
8. $\leftarrow chama(a,z) usa(z,e)$. 4
9. $\leftarrow chama(a,b)$. 2
10. \square . 1

Note que " $depende(z,e)$ " foi o literal selecionado da cláusula em (7) e " $usa(z,e)$ " o da cláusula em (8).

A noção de $LSD(f)$ -dedução induz uma família de métodos de refutação, indexada pela função de seleção e definida da seguinte forma:

Definição 7.9:

Seja f uma função de seleção. O *método da resolução linear com função de seleção f para conjuntos quase-definidos* ou, abreviadamente, *resolução-LSD(f)*, consiste do par $(RSD(f), DD(f))$, onde $RSD(f)$ é o sistema formal da resolução com função de seleção f para conjuntos quase-definidos e $DD(f)$ é o conjunto das $LSD(f)$ -deduções.

Quando não for relevante mencionar a função de seleção, nos referiremos ao método como resolução-LSD simplesmente.

Como consequência das restrições sobre a aplicação da regra de f -extensão e sobre a construção das $LSD(f)$ -deduções, o método da resolução-LSD(f) possui as seguintes características básicas:

- O conjunto de entrada deverá ser um conjunto de cláusulas definidas, acrescido de uma cláusula-objetivo.
- As refutações são lineares de entrada e têm o seguinte formato:
 - uma lista de cláusulas definidas de entrada, seguida de
 - a cláusula-objetivo de entrada, seguida de
 - uma lista de cláusulas-objetivo
- Fatoração é desnecessária.

- A cláusula auxiliar em uma aplicação da regra de f-extensão é sempre uma cláusula definida do conjunto de entrada e o resolvente é sempre uma cláusula-objetivo.
- A estratégia de seleção de literais, incorporada na definição da regra de f-extensão, dita que
 - o literal da cláusula-pai selecionado é indicado pela função de seleção f ;
 - o literal da cláusula auxiliar selecionado é sempre a cabeça da cláusula.

As árvores de refutação por resolução-LSD(f) ou, simplesmente, árvores de LSD(f)-refutação, também são consideravelmente mais simples já que os rótulos dos nós serão sempre cláusulas-objetivo e os filhos de um nó serão obtidos apenas por f-extensão, usando como cláusula auxiliar cada uma das cláusulas definidas de entrada cuja cabeça é unificável com o literal selecionado por f da cláusula que rotula o nó. Note que nenhum outro literal além do selecionado é considerado, o que não é o caso na construção das árvores de refutação por resolução linear.

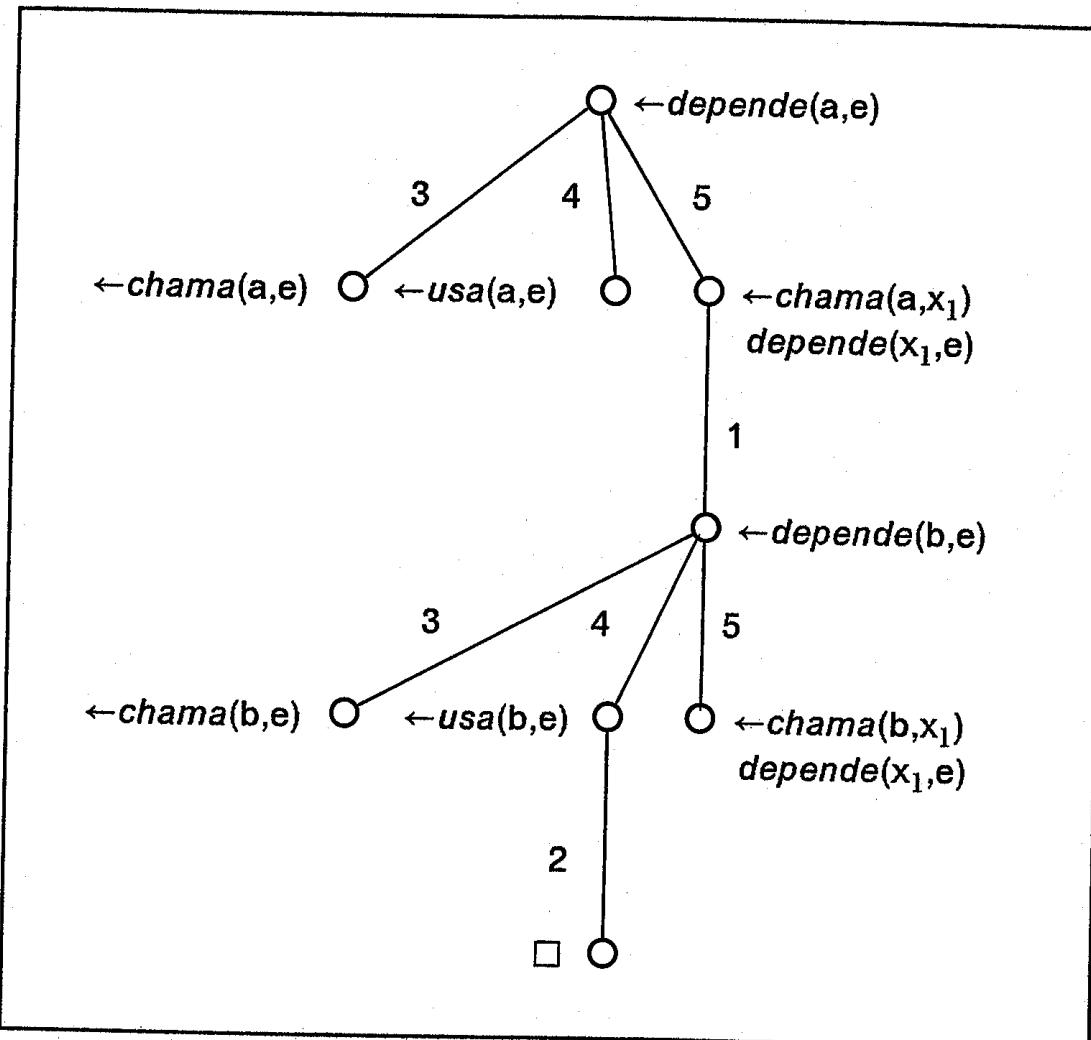
Mais importante ainda, para determinar se um conjunto quase-definido **S** é insatisfatível, basta construir a árvore de LSD(f)-refutação começando na cláusula-objetivo do conjunto de entrada e determinar se esta árvore possui um nó de sucesso. Portanto não é necessário considerar a floresta de refutação completa, o que reduz consideravelmente o problema de construir procedimentos de refutação para conjuntos quase-definidos.

Exemplo 7.2:

Seja **S** o seguinte conjunto de entrada:

1. *chama(a,b) ←*
2. *usa(b,e) ←*
3. *depende(x,y) ← chama(x,y)*
4. *depende(x,y) ← usa(x,y)*
5. *depende(x,y) ← chama(x,z) depende(z,y)*
6. *←depende(a,e)*

Seja f a função de seleção padrão. Para determinar se **S** é insatisfatível basta construir a seguinte árvore de LSD(f)-refutação:



Para cada nó N , com rótulo A' , construímos o conjunto dos seus filhos da seguinte forma. Para cada cláusula A'' de entrada, na ordem apresentada, se existe uma f-extensão A de A' por A'' , adicionamos um novo filho de N rotulado com a variante canônica de A . Como esta árvore possui um nó de sucesso, S é insatisfatível.

7.3 CORREÇÃO E COMPLETUDADE DO MÉTODO DA RESOLUÇÃO-LSD

Como resolução-LSD pode ser vista como um refinamento de resolução, a sua correção segue imediatamente de resultados anteriores. Porém, a completude requer um tratamento cuidadoso, em parte pelo uso de funções de seleção para governar a aplicação da regra de extensão.

A demonstração da completude de resolução-LSD(f) divide-se em duas etapas. Inicialmente obtém-se a completude do método, quando a função

de seleção é a função padrão f , mapeando-se EMF-refutações em LSD(f)-refutações conforme ilustrado pelo seguinte exemplo.

Exemplo 7.3:

Considere a seguinte EMF-refutação (as cláusulas de entrada são as seis primeiras):

1. $chama(a,b)$
2. $usa(b,e)$
3. $depende(x,y) \neg chama(x,y)$
4. $depende(x,y) \neg usa(x,y)$
5. $depende(x,y) \neg chama(x,z) \neg depende(z,y)$
6. $\neg depende(a,e)$
7. $\neg chama(a,z) \neg depende(z,e) [\neg depende(a,e)] . 5a$
8. $[\neg chama(a,b)] \neg depende(b,e) [\neg depende(a,e)] . 1a$
9. $\neg depende(b,e) [\neg depende(a,e)] . CN$
10. $\neg usa(b,e) [\neg depende(b,e)] [\neg depende(a,e)] . 4a$
11. $[\neg usa(b,e)] [\neg depende(b,e)] [\neg depende(a,e)] . 2a$
12. $\square . CN$

Esta EMF-refutação pode ser diretamente mapeada em uma LSD(f)-refutação, eliminando as cadeias resultantes de contração, removendo de cada cadeia os R-literais e, em seguida, mapeando cada cadeia em uma cláusula-objetivo. O resultado será (não renumeramos os passos para facilitar a comparação):

1. $chama(a,b) \leftarrow$
2. $usa(b,e) \leftarrow$
3. $depende(x,y) \leftarrow chama(x,y)$
4. $depende(x,y) \leftarrow usa(x,y)$
5. $depende(x,y) \leftarrow chama(x,z) depende(z,y)$
6. $\leftarrow depende(a,e)$
7. $\leftarrow chama(a,z) \neg depende(z,e) . 5$
8. $\leftarrow depende(b,e) . 1$
10. $\leftarrow usa(b,e) . 4$
11. $\square . 2$

Note que a cadeia (11) da primeira refutação passou a ser a cláusula vazia após a eliminação dos R-literais.

O lema a seguir formaliza as propriedades importantes das EMF-refutações a partir de um conjunto quase-definido e iniciando-se na cadeia-objetivo do conjunto.

Lema 7.1:

Sejam \mathbf{R} um conjunto contendo apenas cadeias elementares com exatamente um literal positivo e \mathbf{C} uma cadeia elementar com todos os literais negativos. Seja $\mathbf{S} = \mathbf{R} \cup \{\mathbf{C}\}$. Seja $\mathbf{D} = (\mathbf{D}_1, \dots, \mathbf{D}_n)$ uma EMF-refutação a partir de \mathbf{S} e iniciando-se em \mathbf{C} . Seja $\mathbf{D}_1, \dots, \mathbf{D}_r$ o prefixo de \mathbf{D} . Então, para todo $i \in [r, n]$

- (i) \mathbf{D}_i é uma cadeia com todos os elementos negativos.
- (ii) \mathbf{D}_i é a cadeia inicial ou \mathbf{D}_i é a contração de \mathbf{D}_{i-1} ou \mathbf{D}_i é a extensão de \mathbf{D}_{i-1} por uma cadeia definida \mathbf{D}_j , $j < r$, selecionando o único literal positivo de \mathbf{D}_j .

Demonstração

Provaremos que (i) e (ii) valem para \mathbf{D}_i , por indução sobre $i \in [r, n]$.

Base: \mathbf{D}_r é a cadeia inicial da refutação que, por suposição, é uma cadeia com todos os elementos negativos. Logo, (i) e (ii) valem trivialmente.

Passo de Indução: Seja $i \in (r, n]$ e suponha que (i) e (ii) valham para \mathbf{D}_{i-1} . Então, \mathbf{D}_{i-1} é uma cadeia com apenas elementos negativos. Logo, redução não se aplica a \mathbf{D}_{i-1} . Portanto, \mathbf{D}_i só pode ser obtida de \mathbf{D}_{i-1} por contração ou por extensão usando uma cadeia auxiliar \mathbf{D}_j do prefixo de \mathbf{D} . No primeiro caso, \mathbf{D}_i possuirá apenas elementos negativos, pois \mathbf{D}_{i-1} possui apenas elementos negativos. No segundo caso, como \mathbf{D}_{i-1} e a cadeia inicial \mathbf{C} são cadeias com apenas elementos negativos, \mathbf{D}_i não pode ser uma extensão de \mathbf{D}_{i-1} por \mathbf{C} . Logo, \mathbf{D}_j e \mathbf{C} são diferentes, o que significa que $j < r$, ou seja, \mathbf{D}_j pertence a \mathbf{R} . Como \mathbf{D}_j só tem então um literal positivo e \mathbf{D}_{i-1} só tem elementos negativos, apenas o literal positivo de \mathbf{D}_j poderá ser selecionado na aplicação da regra de extensão e o resolvente \mathbf{D}_i só terá elementos negativos.

De posse deste lema, podemos provar o principal resultado para resolução-LSD(f):

Teorema 7.1: (Correção e Completude de Resolução-LSD Padrão)

Seja f a função de seleção padrão.

- (a) Para todo conjunto quase-definido \mathbf{S} , se existe uma $LSD(f)$ -refutação a partir de \mathbf{S} então \mathbf{S} é insatisfatível.
- (b) Para todo conjunto quase-definido \mathbf{S} , se \mathbf{S} é insatisfatível, então existe uma $LSD(f)$ -refutação a partir de \mathbf{S} .

Demonstração

(a) Pela Definição 7.3, e pela própria definição de extensão e de f-extensão, uma $LSD(f)$ -dedução \mathbf{R} pode ser mapeada em uma R -dedução \mathbf{S} tal que todas as cláusulas de Horn em \mathbf{R} são satisfatóveis se e somente as cláusulas em \mathbf{S} o forem. Logo, a correção da resolução- $LSD(f)$ segue do Teorema 4.2.

(b) Seja f a função de seleção padrão. Seja \mathbf{S} um conjunto quase-definido e suponha que \mathbf{S} seja insatisfatível. Provaremos que existe uma $LSD(f)$ -refutação a partir de \mathbf{S} . Seja \mathbf{S}' o conjunto das cláusulas equivalentes às cláusulas de Horn em \mathbf{S} . Como \mathbf{S} é insatisfatível, \mathbf{S}' também será insatisfatível. Note que \mathbf{S}' pode ser visto também como um conjunto de cadeias elementares. Seja \mathbf{B} a cadeia de \mathbf{S}' com apenas literais negativos. Como $\mathbf{S}' - \{\mathbf{B}\}$ só possui cadeias com exatamente um literal positivo, este conjunto é satisfatório. Mas \mathbf{S}' é insatisfatível. Logo, pelo Teorema 6.4, existe uma EMF-refutação, \mathbf{D} , a partir de \mathbf{S}' e iniciando-se em \mathbf{B} . Construa a seqüência \mathbf{E} de cláusulas de Horn mapeando as cadeias do prefixo de \mathbf{D} nas cláusulas de Horn correspondentes de \mathbf{S} , eliminando de \mathbf{D} as cadeias resultantes de contração, removendo de cada cadeia restante os R-literais e mapeando cada uma destas cadeias na cláusula-objetivo equivalente. Como \mathbf{D} possui as propriedades enunciadas no Lema 7.1, e pela própria definição de extensão e de f-extensão, este mapeamento é possível e \mathbf{E} será então uma $LSD(f)$ -refutação a partir de \mathbf{S} .

A completude do método para qualquer função de seleção segue como corolário do Teorema 7.1e do seguinte lema (para uma demonstração veja Lloyd [1984]).

Lema 7.2:

Sejam f e g funções de seleção. Seja \mathbf{S} um conjunto quase-definido. Para toda $LSD(f)$ -refutação \mathbf{D} a partir de \mathbf{S} , existe uma $LSD(g)$ -refutação \mathbf{E} a partir de \mathbf{S} tal que \mathbf{D} e \mathbf{E} possuem o mesmo comprimento e, se β_1, \dots, β_k é a seqüência de $u.m.g's$ usada em \mathbf{D} e $\theta_1, \dots, \theta_k$ é a seqüência de $u.m.g's$ usados em \mathbf{E} , então existe uma renomeação φ tal que $\theta_1 \circ \dots \circ \theta_k = \beta_1 \circ \dots \circ \beta_k \circ \varphi$.

Corolário 7.1: (Correção e Completude de Resolução-LSD)

- (a) Para toda função de seleção f , para todo conjunto quase-definido \mathbf{S} , se existe uma $LSD(f)$ -refutação a partir de \mathbf{S} então \mathbf{S} é insatisfatível.
- (b) Para toda função de seleção f , para todo conjunto quase-definido \mathbf{S} , se \mathbf{S} é insatisfatível, então existe uma $LSD(f)$ -refutação a partir de \mathbf{S} .

O último resultado indica que realmente só é necessário construir uma árvore de refutação no contexto de resolução-LSD.

Teorema 7.2:

Para todo conjunto quase-definido \mathbf{S} , se \mathbf{S} é insatisfatível então existe um nó de sucesso na árvore de $LSD(f)$ -refutação para \mathbf{S} começando na cláusula-objetivo de \mathbf{S} .

Demonstração

Seja \mathbf{S} um conjunto quase-definido e suponha que \mathbf{S} seja insatisfatível. Seja A a árvore de $LSD(f)$ -refutação para \mathbf{S} começando na cláusula-objetivo de \mathbf{S} . Então pelo Corolário 7.1, existe uma $LSD(f)$ -refutação a partir de \mathbf{S} que começa por definição na cláusula-objetivo de \mathbf{S} . A esta refutação corresponderá, por construção, um ramo de A . Logo, este ramo terminará em um nó rotulado com a cláusula vazia.

7.4 UMA INTERPRETAÇÃO INTUITIVA PARA RESOLUÇÃO-LSD

O processo de refutação por resolução-LSD admite uma interpretação intuitiva, introduzida em Kowalski [1974], que teve grande influência sobre a popularização de Programação em Lógica. Nesta interpretação intuitiva, uma cláusula definida da forma " $A \leftarrow B_1 \dots B_n$ " é entendida como um *procedimento*. O *nome* do procedimento é o literal A , que identifica os

problemas que o procedimento pode resolver. O *corpo* do procedimento é o conjunto de *chamadas* B_1, \dots, B_n . Uma cláusula-objetivo da forma " $\leftarrow C_1 \dots C_m$ " consistindo apenas de chamadas (ou problemas a resolver) se comporta como uma *lista de objetivos*.

Um procedimento da forma " $A \leftarrow B_1 \dots B_n$ " é *chamado* por uma chamada C_i da lista de objetivos:

1. unificando a chamada C_i com o nome A do procedimento, gerando assim um unificador mais geral θ tal que $C_i\theta = A\theta$;
2. substituindo a chamada C_i pelo corpo do procedimento, obtendo uma nova lista de objetivos:

$$\leftarrow C_1 \dots C_{i-1} B_1 \dots B_n C_{i+1} \dots C_m$$

3. aplicando a substituição θ gerada pelo processo de unificação:

$$\leftarrow (C_1 \dots C_{i-1} B_1 \dots B_n C_{i+1} \dots C_m)\theta$$

A parte da substituição θ que afeta as variáveis de C_1, \dots, C_m transmite *valores de saída*, já a parte que afeta variáveis de B_1, \dots, B_n transmite *valores de entrada* para o procedimento chamado. Se o passo (1) for bem sucedido, a chamada terá *sucesso*, caso contrário a chamada *fracassará*.

Dado um conjunto de cláusulas definidas P e uma cláusula-objetivo O da forma " $\leftarrow C_1 \dots C_m$ " o processo de refutação se dará da seguinte forma:

1. Faça a lista de objetivos corrente, OC , inicialmente igual a O ;
2. Selecione uma chamada C de OC , de acordo com a função de seleção dada;
3. Caso C possa chamar algum procedimento em P com sucesso:
 - a. selecione um destes procedimentos e gere o novo valor de OC conforme explicado acima;
 - b. Se OC for vazia, pare com SUCESSO, senão retorne para o passo (2);
4. Caso C fracasse em todas as chamadas para procedimentos do programa P , refaça a lista de objetivos que havia antes da atual, faça o valor de OC igual a esta lista e volte para o passo (2);

Este procedimento não determinístico é uma abstração de procedimentos de refutação seguindo resolução-LSD. Um particular procedimento é fixado escolhendo-se uma função de seleção para determinar o literal **C** no passo (2) e uma função de seleção para determinar a ordem com que as cláusulas de **P** são consideradas no passo (3a). A máquina Prolog, definida na seção 10.3, é um refinamento deste procedimento.

7.5 O MÉTODO DA RESOLUÇÃO-LSDNF

O uso de resolução-LSD impõe que o conjunto de entrada deva ser composto de cláusulas definidas e de exatamente uma cláusula-objetivo, o que acarreta limitações importantes. Por exemplo, uma simples cláusula da forma " $p(a) \ p(b)$ " não poderá pertencer a um conjunto de entrada para resolução-LSD. Esta seção explora então brevemente uma extensão da resolução-LSD que admite conjuntos de entrada mais gerais, mas mantém parte da simplicidade da resolução-LSD. O leitor deve se referir a Lloyd [1984] ou Clark [1978] para uma discussão detalhada dos resultados apresentados.

A extensão considerada trata a negação como "impossibilidade de prova", ou seja, ela permite inferir $\neg P$ em presença de um dado conjunto de cláusulas **S**, se não for possível provar **P** a partir de **S**. Este tratamento não tradicional da negação baseia-se na chamada "hipótese do mundo fechado" e simplifica consideravelmente a criação de programas em lógica.

Novamente, por simplicidade, os alfabetos de primeira ordem passarão a conter os símbolos " \square " e " \leftarrow ". Os conjuntos de entrada admitidos pela resolução-LSDNF deverão ser quase-pseudo-definidos, noção definida de forma inteiramente semelhante à de conjunto quase-definido.

Definição 7.10:

Seja **A** uma alfabeto de primeira ordem.

- (a) Uma *cláusula pseudo-definida* sobre **A** é uma expressão da forma " $L \leftarrow M_1 \dots M_n$ ", onde **L** é um literal positivo e M_1, \dots, M_n são literais sobre **A**. O literal **L** é a *cabeça* e os literais M_1, \dots, M_n foram o *corpo* da cláusula.
- (b) Uma *cláusula pseudo-objetivo* sobre **A** é ou a cláusula vazia ou é uma expressão da forma " $\leftarrow M_1 \dots M_n$ ", onde M_1, \dots, M_n são literais sobre **A**. Os literais M_1, \dots, M_n foram o *corpo* da cláusula.

- (c) Uma *cláusula pseudo-Horn* sobre A é ou uma cláusula pseudo-definida ou uma cláusula pseudo-objetivo sobre A .
- (d) Um *conjunto quase-pseudo-definido* é um conjunto de cláusulas pseudo-definidas acrescido de exatamente uma cláusula pseudo-objetivo.

Por exemplo, " $p(x,a) \leftarrow$ ", " $p(x) \leftarrow q(y)$ " e " $p(x) \leftarrow \neg q(x) r(x,a)$ " são cláusulas pseudo-definidas e " \square " e " $\leftarrow r(x,y) \neg p(x)$ " são cláusulas pseudo-objetivo.

Note que toda cláusula definida é pseudo-definida e que toda cláusula-objetivo é pseudo-objetivo, mas não vice-versa. A semelhança entre os dois pares de conceitos é apenas superficial e decorrente da notação pois, diferentemente das cláusulas definidas e cláusulas-objetivo, os literais do corpo das cláusulas pseudo-definidas e das cláusulas pseudo-objetivo poderão ser positivos ou negativos.

Definição 7.11:

A *linguagem das cláusulas pseudo-Horn* sobre um alfabeto de primeira ordem A é o conjunto de todas as cláusulas pseudo-Horn sobre A .

A semântica das linguagens das cláusulas pseudo-Horn segue diretamente da semântica das linguagens de cláusulas, estendendo-se a noção de satisfabilidade da seguinte forma:

Definição 7.12:

Seja A um alfabeto de primeira ordem e I uma estrutura de A .

- (a) I satisfaz uma cláusula pseudo-definida " $L \leftarrow M_1 \dots M_n$ " sobre A se e somente se I satisfaz a cláusula " $L N_1 \dots N_n$ ", onde N_i é o literal complementar de M_i , para $i=1,\dots,n$. Diz-se ainda que as duas cláusulas são *equivalentes*.
- (b) I satisfaz uma cláusula pseudo-objetivo não-vazia " $\leftarrow M_1 \dots M_n$ " sobre A se e somente se I satisfaz a cláusula " $N_1 \dots N_n$ ", onde N_i é o literal complementar de M_i , para $i=1,\dots,n$. Diz-se ainda que as duas cláusulas são *equivalentes*.
- (c) I não satisfaz a cláusula vazia.

Uma cláusula pseudo-definida " $L \leftarrow M_1 \dots M_n$ " representa então a implicação $\forall(M_1 \wedge \dots \wedge M_n \rightarrow L)$ e uma cláusula pseudo-objetivo " $\neg\leftarrow M_1 \dots M_n$ " representa a fórmula $\neg\exists(M_1 \wedge \dots \wedge M_k)$.

Note ainda que toda cláusula genérica com no mínimo um literal positivo é equivalente a uma cláusula pseudo-definida e que toda cláusula genérica " $N_1 \dots N_n$ " é equivalente à cláusula pseudo-objetivo " $\neg\leftarrow M_1 \dots M_n$ ", onde M_i é o literal complementar a N_i . Portanto, como consequência da Definição 7.12, poderíamos definir equivalentemente uma cláusula pseudo-definida como qualquer cláusula com no mínimo um literal positivo e dispensar totalmente o conceito de cláusula pseudo-objetivo. A formulação dada na Definição 7.10 justifica-se apenas por traduzir a notação favorecida na literatura mais recente.

A noção de função de seleção estende-se trivialmente a cláusulas pseudo-definidas. Porém, a definição da noção de dedução a partir de um conjunto quase-pseudo-definido S , governada por uma função de seleção f , requer cuidados especiais pois, como as cláusulas pseudo-definidas em S e a cláusula pscudo-objetivo de S poderão ter um literal negativo no seu corpo, as cláusulas derivadas poderão ter literais positivos e negativos no seu corpo.

Se o literal selecionado de uma cláusula derivada for positivo, ele deverá ser cancelado com a cabeça de uma cláusula pseudo-definida do conjunto de entrada S através da aplicação da regra de f-extensão. Porém, se o literal selecionado for negativo, ele deverá ser básico (ou seja, sem ocorrências de variáveis) e poderá ser cancelado aplicando-se uma nova regra, chamada de negação por falha finita, que dita intuitivamente o seguinte:

NFF. Seja Q o conjunto das cláusulas pseudo-definidas pertencentes ao conjunto de entrada. Se o literal selecionado L de uma cláusula C é básico e negativo e é possível mostrar de forma finitária que $\neg L$ não pode ser refutado a partir de Q , então cancele L de C .

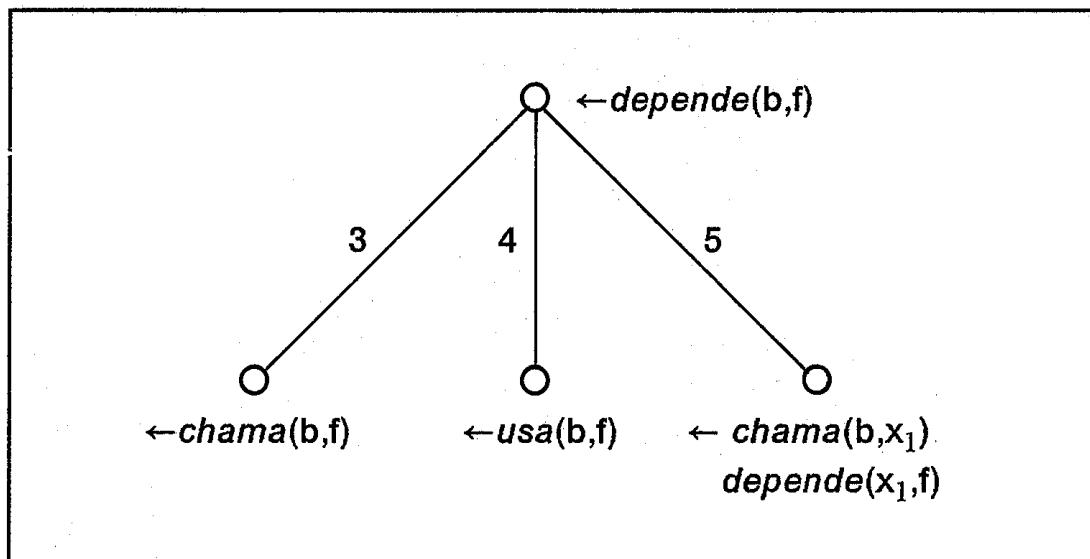
A forma finitária referida na regra é implementada construindo-se a árvore de refutação por resolução-LSDNF a partir de $Q \cup \{\neg L\}$, com a raiz rotulada por $\neg L$. Se a árvore for finita e não tiver nenhum nó de sucesso, então não há nenhuma refutação por resolução-LSDNF a partir de $Q \cup \{\neg L\}$.

Exemplo 7.4:

Seja f a função de seleção padrão. A seguinte seqüência de cláusulas é uma refutação por resolução-LSDNF, governada por f :

1. $chama(a,b) \leftarrow$
2. $usa(b,e) \leftarrow$
3. $depende(x,y) \leftarrow chama(x,y)$
4. $depende(x,y) \leftarrow usa(x,y)$
5. $depende(x,y) \leftarrow chama(x,z) depende(z,y)$
6. $\neg depende(a,w) \neg depende(w,f)$
7. $\neg chama(a,w) \neg depende(w,f)$. 3, EXS
8. $\neg \neg depende(b,f)$. 1, EXS
9. \square . NFF

A cláusula (9) é derivada da cláusula (8) pela regra da negação por falha finita pois a árvore abaixo é finita e não possui nenhum nó de sucesso.



O resto desta seção formaliza esta discussão.

Definição 7.13:

Seja f uma função de seleção. Sejam A' uma cláusula pseudo-objetivo e A'' uma cláusula pseudo-definida. Seja β uma renomeação de A'' em presença de A' . Suponha que A' seja da forma " $\leftarrow N_1 \dots N_m$ ", que $A''\beta$ seja da forma " $L \leftarrow M_1 \dots M_k$ " e que $f(A') = N_i$. Suponha ainda que N_i seja um literal positivo. Uma cláusula A é uma f -extensão de A' por A'' se e somente se existe um u.m.g θ de $\{L, N_i\}$ tal que A é da forma " $\leftarrow (N_1 \dots N_{i-1} M_1 \dots M_k N_{i+1} \dots N_m) \theta$ ".

Note que uma f -extensão de A' por A'' só existe se o literal de A' selecionado por f for positivo. Note ainda que, se A e B são f -extensões de A' por A'' , então A e B são variantes entre si. Portanto, a menos de renomeação de variáveis, a extensão de A' por A'' é única.

Definição 7.14:

Uma cláusula pseudo-objetivo A é uma f -extensão canônica de uma cláusula pseudo-objetivo A' em presença de um conjunto de cláusulas pseudo-definidas Q se e somente se A é a variante canônica de uma f -extensão de A com uma cláusula pseudo-definida em Q .

Definição 7.15:

Seja f uma função de seleção. Sejam Q um conjunto de cláusulas pseudo-definidas e C uma cláusula pseudo-objetivo. Uma árvore, com os nós rotulados por cláusulas pseudo-objetivo, é uma *árvore de LSDNF(f)-refutação* para Q e C se e somente se

- (i) o rótulo da raiz é C ;
- (ii) para cada nó X , o conjunto dos filhos de X é definido da seguinte forma. Seja A' o rótulo de X e suponha que A' seja da forma " $\leftarrow N_1 \dots N_k$ " e que $f(A') = N_i$. Há dois casos a considerar:

caso 1: N_i é um literal positivo. Para cada f -extensão canônica A de A' em presença de Q , existe exatamente um filho de X rotulado com A .

caso 2: N_i é um literal negativo. Se N_i for um literal básico da forma $\neg L$ e se a árvore de LSDNF(f)-refutação para Q e $\neg L$ for finita e não possuir um nó de sucesso, então o nó X terá um único filho rotulado por " $\neg N_1 \dots N_{i-1} N_{i+1} \dots N_k$ ". Se N_i contiver variáveis livres ou a árvore for infinita ou possuir um nó de sucesso, o nó X não terá filhos.

As noções de nó de sucesso, nó de fracasso, ramo de sucesso e ramo de fracasso seguem como anteriormente.

O tratamento dos literais negativos pode ser fatorado da construção da árvore de refutação e colocado da seguinte forma.

Definição 7.16:

Seja f uma função de seleção. Sejam C uma cláusula pseudo-objetivo da forma " $\neg N_1 \dots N_m$ " e Q um conjunto de cláusulas pseudo-definidas. Suponha que $f(C) = N_i$ e que N_i seja um literal básico negativo da forma $\neg L$. A cláusula " $\neg N_1 \dots N_{i-1} N_{i+1} \dots N_m$ " é a f -contração por falha finita de C em presença de Q se e somente se existe uma árvore de LSDNF(f)-refutação A a partir de Q e $\neg L$ tal que A é finita e não possui um nó de sucesso.

De posse destas definições, podemos então introduzir uma nova família de sistemas formais incluindo a negação por falha finita.

Definição 7.17:

Seja f uma função de seleção. O sistema formal da *resolução com função de seleção f para conjuntos quase-pseudo-definidos*, $RSDNF(f)$, consiste de:

Classe de Linguagens: linguagens das cláusulas pseudo-Horn

Axiomas: nenhum

Regras de Inferência:**f-Extensão (EXS):**

EXS: se A' é uma cláusula pseudo-objetivo,
 A'' é uma cláusula pseudo-definida e
 A é uma f-extensão de A' por A''
 derive A de A' e A''

Negação por Falha Finita (NFF):

NFF: se A' é uma cláusula pseudo-objetivo,
 Q é um conjunto de cláusulas pseudo-definidas e
 A é a f-contração por falha finita de A' em presença de Q
 derive A de A' e Q

A noção de LSDNF(f)-dedução segue como na seção 6.6, exceto que agora há uma nova regra para tratamento dos literais selecionados que forem básicos e negativos.

Definição 7.18:

Seja f uma função de seleção. Sejam Q um conjunto de cláusulas pseudo-definidas e P uma cláusula pseudo-objetivo. Uma *LSDNF(f)-dedução* de uma cláusula pseudo-Horn C a partir de $Q \cup \{P\}$ é uma seqüênci $D = (D_1, \dots, D_n)$ de cláusulas pseudo-Horn tal que $D_n = C$ e existe $r \leq n$ tal que

- (i) para todo $i < r$, D_i é uma cláusula de Q ;
- (ii) D_r é a cláusula P ;
- (iii) para todo $i > r$, D_i é uma cláusula obtida de D_{i-1} de duas formas:

caso 1: o literal de D_{i-1} selecionado por f é positivo. Então, D_i é uma f-extensão de D_{i-1} por uma cláusula pseudo-definida D_j , para algum $j < r$.

caso 2: o literal de D_{i-1} selecionado por f é básico e negativo. Então, D_i é a f-contração por falha finita de D_{i-1} em presença de $\{D_1, \dots, D_{r-1}\}$.

Definição 7.19:

Seja f uma função de seleção. Sejam \mathbf{Q} um conjunto de cláusulas pseudo-definidas e P uma cláusula pseudo-objetivo. Uma $LSDNF(f)$ -refutaçāo a partir de $\mathbf{Q} \cup \{P\}$ é uma $LSDNF(f)$ -dedução da cláusula vazia a partir de $\mathbf{Q} \cup \{P\}$.

A noção de $LSDNF(f)$ -dedução induz uma família de métodos de refutação definida da seguinte forma:

Definição 7.20:

Seja f uma função de seleção. O método de *resolução linear com função de seleção f para conjuntos quase-pseudo-definidos e negação por falha finita* ou, simplesmente, resolução-LSDNF(f), consiste do par $(RSDNF(f), DDNF(f))$, onde $RSDNF(f)$ é o sistema formal da resolução com função de seleção para conjuntos quase-pseudo-definidos e $DDNF(f)$ é o conjunto das $LSDNF(f)$ -deduções.

Embora a descrição da resolução-LSDNF seja consideravelmente mais elaborada, procedimentos de refutação baseados neste método operarão de forma quase idêntica aos procedimentos baseados em resolução-LSD. De fato, observe que a construção de uma árvore de LSDNF-refutação prossegue de forma idêntica à de uma árvore de LSD-refutação até que um literal negativo $\neg L$ seja selecionado da cláusula C rotulando um nó X . Se $\neg L$ for básico, o procedimento deverá ser chamado recursivamente para construir a árvore de LSDNF-refutação para $\neg L$ em presença das cláusulas pseudo-definidas de entrada. Se esta árvore for finita e não possuir nós de sucesso, o nó X terá um filho rotulado com a cláusula C sem o literal $\neg L$. Como $\neg L$ é básico, a chamada recursiva age como um simples teste e não gera substituições.

7.6 CORREÇÃO E COMPLETUDADE DO MÉTODO DA RESOLUÇÃO-LSDNF

7.6.1 Correção

A resolução-LSDNF não é correta no sentido de que, se existe uma LSDNF-refutação a partir de um conjunto quase-pseudo-definido S , então S é insatisfatível. De fato, considere o seguinte exemplo:

Exemplo 7.5:

Sejam \mathbf{Q} o conjunto vazio e \mathbf{C} a cláusula " $\leftarrow \neg p(a)$ ". Note que $f(\leftarrow \neg p(a)) = \leftarrow \neg p(a)$, para toda função de seleção f . Então há uma LSDNF(f)-refutação a partir de $\mathbf{Q} \cup \{\mathbf{C}\}$:

1. $\leftarrow \neg p(a)$
2. \square . NFF

De fato, (2) segue de (1) pela regra da negação por falha finita pois a árvore de LSDNF(f)-refutação para \mathbf{Q} e " $\leftarrow p(a)$ " se reduz apenas à raiz rotulada com a cláusula " $\leftarrow p(a)$ " já que não há nenhuma f -extensão desta cláusula por alguma cláusula pseudo-definida em \mathbf{Q} . Portanto, a árvore é finita e não tem nenhum nó de sucesso.

Porém, o conjunto $\mathbf{Q} \cup \{\mathbf{C}\}$ é satisfável pois possui, por exemplo, o conjunto $\{p(a)\}$ como H-modelo.

A correção da resolução-LSDNF(f) depende de uma suposição adicional, chamada de *Hipótese do Mundo Fechado* (HMF), desenvolvida originalmente para o contexto de bancos de dados e posteriormente estendido para bancos de dados dedutivos e programas em lógica.

Dizer que um banco de dados \mathbf{B} satisfaz a HMF significa assumir que \mathbf{B} contém todos os fatos verdadeiros acerca de um universo de discurso. O exemplo a seguir ilustra uma possível formalização desta noção vaga.

Exemplo 7.6:

Considere o banco de dados DICIONÁRIO contendo dados sobre programas. Suponha que as seguintes fórmulas atômicas denotem o estado \mathbf{E} deste banco em um determinado instante:

- F₁. $programa(A)$
- F₂. $programa(B)$
- F₃. $programa(C)$
- F₄. $chama(A,B)$
- F₅. $chama(A,C)$

Assuma que DICIONÁRIO satisfaz a HMF e que o alfabeto usado contenha apenas o símbolo predicativo unário *programa*, o símbolo predicativo binário *chama* e as constantes A, B, C e D. Estas

suposições significam que, embora as fórmulas F_1, \dots, F_5 descrevam o estado E , as consultas a E serão avaliadas contra o *fecho* de E , definido como as seguintes fórmulas:

- $P_1: \forall x(programa(x) \equiv (x=A \vee x=B \vee x=C))$
- $P_2: \forall x\forall y(chama(x,y) \equiv ((x=A \wedge x=B) \vee (x=A \wedge x=C)))$
- $E_1: A \neq B$
- $E_2: A \neq C$
- $E_3: A \neq D$
- $E_4: B \neq C$
- $E_5: B \neq D$
- $E_6: C \neq D$

Note que P_1 indica que x é um programa se e somente se x for A , B ou C . Observação semelhante vale para P_2 . As fórmulas E_1 a E_6 indicam que as constantes denotam elementos distintos.

A consulta " $\neg programa(D)$ " a DICIONÁRIO terá então como resposta SIM, quando avaliada contra o estado E , tendo em vista que P_1, E_1, E_2, \dots pertencem ao fecho de E . Note que, sem assumir que DICIONÁRIO satisfaz a HMF, a resposta desta consulta ao estado E deveria ser NÃO ou NÃO-SEI, pois o fato do banco não conter informação acerca de D não significa que "no mundo real" D não seja um programa. De fato, DICIONÁRIO poderia satisfazer o inverso da hipótese do mundo fechado, chamada de *Hipótese do Mundo Aberto*.

Uma das formalizações da HMF no contexto de Programação em Lógica, discutida a seguir, estende a noção de fecho a conjuntos de cláusulas pseudo-definidas e permite provar a correção da resolução-LSDNF no sentido de que, dados um conjunto Q de cláusulas pseudo-definidas e uma cláusula C , se existe uma LSDNF-refutação a partir de $Q \cup \{C\}$, então $fecho(Q) \cup \{C\}$ é insatisfável.

Como Q contém cláusulas pseudo-definidas, e não simplesmente fórmulas atômicas como em bancos de dados, a definição do fecho de Q exige alguns conceitos preliminares.

Definição 7.21:

Seja \mathbf{Q} um conjunto de cláusulas pseudo-definidas sobre um alfabeto de primeira ordem A . Seja p um símbolo predicativo n-ário de A que ocorre na cabeça de alguma cláusula de \mathbf{Q} . Sejam C_1, \dots, C_k todas as cláusulas em \mathbf{Q} cuja cabeça é um literal sobre p . Sejam x_1, \dots, x_n variáveis que não ocorrem em C_1, \dots, C_k . A *definição completa* de p em \mathbf{Q} é a fórmula

$$\forall x_1 \dots \forall x_n (p(x_1, \dots, x_n) \equiv (E_1 \vee \dots \vee E_k))$$

onde E_i , para $i = 1, \dots, k$, é a fórmula

$$\exists y_1 \dots \exists y_j (x_1 = t_1 \wedge \dots \wedge x_n = t_n \wedge M_1 \wedge \dots \wedge M_m)$$

se C_i é da forma " $p(t_1, \dots, t_n) \leftarrow M_1 \dots M_m$ " e as variáveis que ocorrem em C_i são y_1, \dots, y_j .

Definição 7.22:

Seja \mathbf{Q} um conjunto de cláusulas pseudo-definidas sobre um alfabeto de primeira ordem A . Seja q um símbolo predicativo n-ário de A que não ocorre na cabeça de alguma cláusula de \mathbf{Q} . A *definição completa* de q em \mathbf{Q} é a fórmula

$$\forall x_1 \dots \forall x_n (\neg q(x_1, \dots, x_n))$$

Note que a escolha da cabeça de uma cláusula pseudo-definida é então bastante importante pois influencia a construção da definição completa dos símbolos predicativos.

Exemplo 7.7:

Seja \mathbf{Q} o seguinte conjunto de cláusulas pseudo-definidas sobre um alfabeto A :

1. $p(y) \leftarrow q(y) \neg r(a,y)$
2. $p(f(z)) \leftarrow \neg q(z)$
3. $p(b) \leftarrow$

A definição completa de p será então a fórmula:

$$\begin{aligned} \forall x (p(x) \equiv (& \exists y ((x=y) \wedge q(y) \wedge \neg r(a,y)) \vee \\ & \exists z ((x=f(z)) \wedge \neg q(z)) \vee (x=b))) \end{aligned}$$

Como q e r não ocorrem no primeiro literal de alguma cláusula de Q , as definições completas de q e r em Q serão $\forall x(\neg q(x))$ e $\forall x \forall y(\neg r(x,y))$.

Finalmente, se s é um símbolo predicativo n-ário de A que não ocorre em Q , a definição completa de s em Q será a fórmula $\forall x_1 \dots \forall x_n (\neg s(x_1, \dots, x_n))$.

A correção da resolução-LSDNF depende ainda de duas suposições. Primeiro, o fecho de um conjunto de cláusulas pseudo-definidas, além da definição completa de cada símbolo predicativo, deverá conter uma teoria adequada da igualdade. Segundo, os indivíduos denotados diretamente por constantes distintas, ou indiretamente através dos símbolos funcionais, deverão ser distintos.

A definição final de fecho captura todos estes pontos (novamente VF denota o fecho universal de uma fórmula F).

Definição 7.23:

O *fecho* de um conjunto de cláusulas pseudo-definidas Q sobre um alfabeto de primeira ordem A é o conjunto de fórmulas, denotado por $\text{fecho}(Q)$, contendo a definição completa em Q de cada símbolo predicativo de A e o seguinte conjunto de fórmulas:

1. $c \neq d$
para todo par de constantes c e d de A
2. $\forall(f(x_1, \dots, x_n) \neq g(y_1, \dots, y_m))$
para todo par de símbolos funcionais distintos f e g
3. $\forall(f(x_1, \dots, x_n) \neq c)$
para todo símbolo funcional f e toda constante c
4. $\forall(x \neq t)$
para toda variável x e todo termo t em que x não ocorre
5. $\forall((x_1 \neq y_1) \vee \dots \vee (x_n \neq y_n) \rightarrow f(x_1, \dots, x_n) \neq f(y_1, \dots, y_n))$
para todo símbolo funcional f
6. $\forall x(x = x)$
7. $\forall x \forall y((x = y) \rightarrow (P \rightarrow P'))$
se P for uma fórmula atômica e P' for obtida de P substituindo-se zero ou mais ocorrências de x por y .

A noção de complemento leva a uma formalização da HMF no seguinte sentido:

Definição 7.24:

Sejam \mathbf{Q} um conjunto de cláusulas pseudo-definidas e C uma cláusula pseudo-objetiva sobre um alfabeto A . O conjunto quase-pseudo-definido $\mathbf{Q} \cup \{C\}$ é *insatisfatível em presença da HMF* se e somente se $\text{fecho}(\mathbf{Q}) \cup \{C\}$ for insatisfatível, onde $\text{fecho}(\mathbf{Q})$ é computado com relação a A .

Esta definição indica que assumir a HMF para um conjunto \mathbf{Q} de cláusulas pseudo-definidas significa testar insatisfatibilidade com relação ao fecho de \mathbf{Q} . Sob esta perspectiva, resolução-LSDNF é então correta, como estabelece o seguinte resultado, contido essencialmente em Clark [1978].

Teorema 7.3:

Seja f uma função de seleção. Sejam \mathbf{Q} um conjunto de cláusulas pseudo-definidas e C uma cláusula pseudo-objetiva. Se existe uma LSDNF(f)-refutação para $\mathbf{Q} \cup \{C\}$ então $\mathbf{Q} \cup \{C\}$ é insatisfatível em presença da HMF.

O próximo exemplo retoma a discussão do Exemplo 7.5 à luz destes novos conceitos.

Exemplo 7.8:

Sejam \mathbf{Q} o conjunto vazio e C a cláusula " $\leftarrow \neg p(a)$ ". Suponha que o alfabeto subjacente só contenha o símbolo predutivo unário p . Então, conforme visto no Exemplo 7.5, há uma LSDNF(f)-refutação a partir de $\mathbf{Q} \cup \{C\}$. Logo, pelo Teorema 7.3, $\mathbf{Q} \cup \{C\}$ deverá ser insatisfatível em presença da HMF.

Mostraremos agora que, de fato, $\text{fecho}(\mathbf{Q}) \cup \{C\}$ é insatisfatível. Observe inicialmente que $\text{fecho}(\mathbf{Q})$ contém, entre outras, a seguinte fórmula:

$$F_1. \forall x(\neg p(x))$$

A cláusula C , por definição, representa a fórmula:

$$F_2. p(a)$$

Logo, como o conjunto consistindo das fórmulas F_1 e F_2 é insatisfatível, $\text{fecho}(Q) \cup \{C\}$ também é insatisfatível.

7.6.2 Completude

Para uma dada função de seleção f , fixada a priori, em vista do Teorema 7.3, a completude da resolução-LSDNF(f) deveria estabelecer que, para todo conjunto de cláusulas pseudo-definidas Q e toda cláusula pseudo-objetiva C , se $Q \cup \{C\}$ é insatisfatível em presença da HMF, então existe uma LSDNF(f)-refutação para $Q \cup \{C\}$. Porém, este resultado não vale, como indica o seguinte exemplo.

Exemplo 7.9:

Seja f uma função de seleção qualquer. Sejam $Q = \{ q(a) \leftarrow p(a) \}$ e C a cláusula " $\leftarrow \neg p(x)$ ". Mostraremos inicialmente que o conjunto $Q \cup \{C\}$ é insatisfatível em presença da HMF. De fato, como p não ocorre na cabeça de nenhuma cláusula em Q , $\text{fecho}(Q)$ contém, por definição, a fórmula:

$$F_1. \quad \forall x(\neg p(x))$$

Também por definição, C é satisfatível se e somente se a seguinte fórmula for satisfatível:

$$F_2. \quad \forall x(p(x))$$

Mas o conjunto consistindo das fórmulas F_1 e F_2 é insatisfatível. Logo, o conjunto $\text{fecho}(Q) \cup \{C\}$ é insatisfatível.

Mostraremos agora que não há uma LSDNF(f)-refutação a partir de $Q \cup \{C\}$. Observe inicialmente que toda LSDNF(f)-dedução a partir de $Q \cup \{C\}$ terá C como cláusula inicial. Porém, como $f(C) = f(\leftarrow \neg p(x)) = \neg p(x)$ e como $\neg p(x)$ é um literal negativo, mas não é básico, nem a regra da negação por falha finita, nem a regra da f -extensão se aplicam. Portanto, não há nenhuma LSDNF(f)-refutação a partir de $Q \cup \{C\}$.

Note que o problema encontrado no exemplo anterior, a inexistência de uma LSDNF(f)-refutação, deve-se exclusivamente à seleção de um literal negativo não-básico. Portanto, uma técnica para tentar evitar tal

problema seria postergar a seleção de literais negativos até que eles se tornem básicos através de substituições ao longo da derivação. Esta técnica é utilizada no interpretador MU-Prolog (c.f., Naish [1982]) e é compatível com o método da resolução-LSDNF, como ilustra o seguinte exemplo.

Exemplo 7.10:

Sejam $\mathbf{Q} = \{ p(a) \leftarrow, q(b) \leftarrow \}$ e C a cláusula " $\leftarrow \neg p(x)q(x)$ ". Seja g a função de seleção definida da seguinte forma:

para toda cláusula pseudo-objetivo B ,

se B contém algum literal positivo ou algum literal negativo básico, então $g(B) = L$, onde L é o literal L mais à esquerda de B tal que L é positivo ou negativo básico

se todos os literais de B são negativos não-básicos, $g(B)$ é o literal mais à esquerda de B

Desta forma, temos que:

$$g(C) = g(\leftarrow \neg p(x)q(x)) = q(x)$$

Logo, há uma LSDNF(g)-refutação a partir de $\mathbf{Q} \cup \{C\}$:

1. $p(a) \leftarrow$
2. $q(b) \leftarrow$
3. $\leftarrow \neg p(x) q(x)$
4. $\leftarrow \neg p(b)$. 2, EXS
5. \square . NFF

Note que (5) segue de (4) pela regra da negação por falha finita. De fato, a árvore de refutação se reduz neste caso apenas à raiz rotulada com a cláusula " $\leftarrow p(b)$ " pois não há nenhuma f-extensão desta cláusula por alguma cláusula pseudo-definida em \mathbf{Q} .

Um terceiro exemplo enfatiza que não é possível levantar a restrição de que todo literal negativo selecionado deverá ser básico sem violar a correção da resolução-LSDNF.

Exemplo 7.11:

Sejam $\mathbf{Q} = \{ p(a) \leftarrow \neg q(x), q(a) \leftarrow \}$ um conjunto de cláusulas pseudo-definidas e C a cláusula pseudo-objetivo " $\leftarrow \neg p(a)$ ". Então, a definição completa de p e q em \mathbf{Q} será:

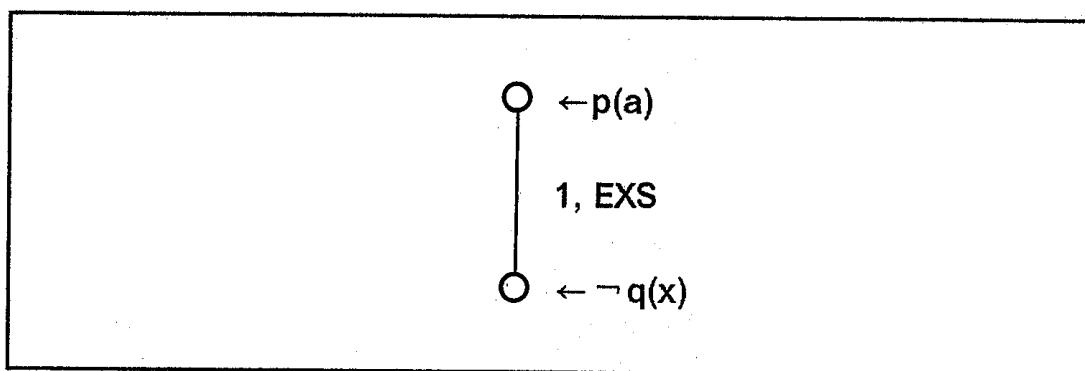
$$\begin{aligned}\forall y(p(y) &\equiv \exists x(y=a \wedge \neg q(x))) \\ \forall y(q(y) &\equiv (y=a))\end{aligned}$$

A cláusula C , por definição, representa a fórmula " $p(a)$ ". Logo, o conjunto $fecho(\mathbf{Q}) \cup \{C\}$ será satisfatível.

Suponha que o cancelamento de literais por falha finita se aplique a literais não-básicos. Então, para qualquer função de seleção f , haverá uma LSDNF(f)-refutação a partir de $\mathbf{Q} \cup \{C\}$. De fato, a seqüência abaixo é uma LSDNF(f)-refutação:

1. $p(a) \leftarrow \neg q(x)$
2. $q(a) \leftarrow$
3. $\leftarrow \neg p(a)$
4. \square . NFF

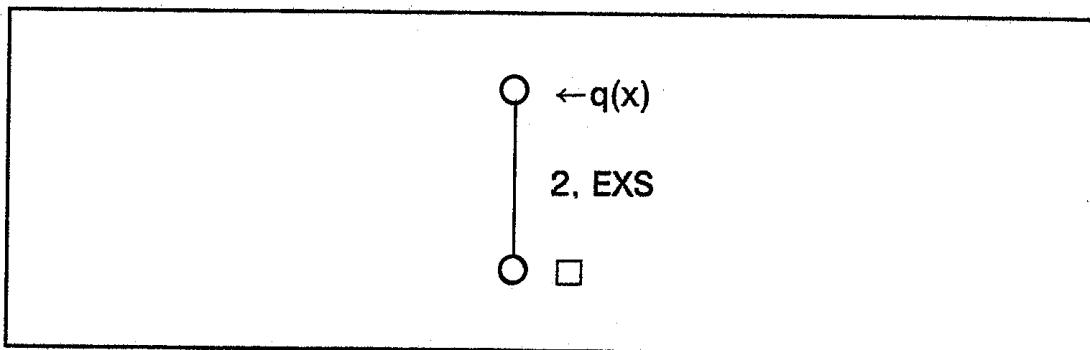
Para justificar a derivação de (4), observe que a árvore de LSDNF(f)-refutação a partir de \mathbf{Q} e $\leftarrow p(a)$ é finita e não possui nós de sucesso:



Para justificar o fato do segundo nó não ter filhos, observe que:

- o literal selecionado de $\leftarrow \neg q(x)$ é negativo
- o cancelamento por falha finita temporariamente se aplica a literais negativos não-básicos, por suposição

- a árvore de LSDNF(f)-refutação para \mathbf{Q} e $\neg q(x)$, apresentada abaixo, possui um ramo de sucesso:



Por fim observamos que a busca de condições em que resolução-LSDNF torna-se completa é um campo de investigação largamente em aberto, mas importante, devido à sua influência potencial sobre a construção de interpretadores Prolog. Clark [1978] apresenta um resultado neste sentido, mas as condições impostas são muito restritivas.

NOTAS BIBLIOGRÁFICAS

O método de resolução-LSD é discutido em bastante detalhe em Lloyd [1984]. Ele foi introduzido com este nome em Apt [1982] e investigado com o nome de resolução-LUSH em Hill [1974]. Ele é um caso particular do método de resolução-SL, introduzido em Kowalski e Kuehner [1971]. A negação por falha finita é tratada em Clark [1978] e Lloyd [1984]. Um tratamento alternativo da negação é discutido em Gabbay [1986].

PARTE III - PROGRAMAÇÃO EM LÓGICA

CAPÍTULO 8: PROGRAMAÇÃO EM CLÁUSULAS GENÉRICAS

Este capítulo considera a variante de Programação em Lógica em que os programas são conjuntos finitos de cláusulas. A seção 8.2 introduz os conceitos de programa, consulta e resposta e discute uma primeira sémântica para programas em cláusulas. A seção 8.3 mostra como computar respostas corretas através de resolução e a seção 8.4 através de eliminação de modelos. Finalmente, a seção 8.5 considera modelagem de bancos de dados dedutivos.

As seções recomendadas para cada nível de leitura são:

nível introdutório: 8.2, 8.3.1

nível intermediário: 8.2, 8.3.1 e 8.5

8.1 INTRODUÇÃO

Este capítulo considera a variante de Programação em Lógica em que os programas são conjuntos finitos de cláusulas, enfatizando os aspectos semânticos, o conceito de resposta e as técnicas para extrair respostas de deduções. Estes pontos extremamente importantes foram escolhidos por representarem conceitos ou técnicas estabilizados e por distinguirem

Programação em Lógica de outras aplicações de Lógica em Ciência da Computação.

Dentro do desenvolvimento atual, Programação em Lógica inclui também comandos extra-lógicos para suprir as facilidades tradicionais de linguagens de programação, como entrada e saída por exemplo, ou para guiar o procedimento de refutação. Estes comandos extra-lógicos fogem da proposta original de Programação Declarativa devendo, portanto, influenciar o menos possível o significado do programa. Este capítulo não discute estas e outras questões, como extensões envolvendo meta-linguagem, por ainda consistirem objeto de discussão (veja, por exemplo, Kowalski [1983a]).

A questão da semântica de Programação em Cláusulas é particularmente importante pois oferece várias alternativas. Primariamente, Programação em Cláusulas possui uma *semântica declarativa* derivada da semântica de linguagens de primeira ordem. A noção de resposta correta de uma consulta a um programa em cláusulas, introduzida na seção 8.2, é o tema central desta abordagem.

Porém, um sistema formal ou, alternativamente, um método de refutação *M* também induz uma *semântica procedural* para Programação em Cláusulas. A preocupação central neste caso é o problema de extrair respostas corretas das deduções admissíveis em *M*. Este enfoque generaliza a semântica procedural para Programação em Cláusulas Definidas, introduzida em Kowalski [1974], e já abordada na seção 6.6. As seções 8.3 e 8.4 tratarão das semânticas procedimentais induzidas, respectivamente, pelos métodos da resolução linear e da eliminação de modelos fraca.

Finalmente, a descrição de um procedimento de refutação abstrato induz uma *semântica operacional* para Programação em Cláusulas. Este enfoque, embora não abordado neste capítulo, permite um melhor tratamento dos comandos extra-lógicos, quando comparado com a semântica declarativa.

Para completar esta introdução, observamos que quatro fatos contribuíram para o desenvolvimento atual de Programação em Lógica. Primeiro, os trabalhos pioneiros de Robinson e outros sobre prova automática de teoremas, desenvolvidos a partir da segunda metade da década de 60, possibilitaram a criação de sistemas para decidir parcialmente se uma sentença é consequência lógica de outras, de forma

bem mais eficiente do que se poderia supor inicialmente. Um breve histórico deste trabalho já foi apresentado nos capítulos 4 a 6, embora convenha acrescentar a esta lista o sistema para perguntas e respostas, baseado em resolução, desenvolvido por Green [1969]. Segundo, Kowalski, em seus trabalhos pioneiros (veja principalmente Kowalski [1974]), introduziu uma interpretação procedural para conjuntos de cláusulas que facilita o desenvolvimento (e o entendimento) de um programa em cláusulas. Terceiro, a introdução da linguagem Prolog e o projeto japonês de computador de quinta geração, junto com outros sistemas, popularizou e motivou o uso de técnicas de Programação em Lógica. Um breve histórico desta parte aparecerá no Capítulo 10. Por fim, Programação em Lógica se beneficiou também da grande atenção dada recentemente a aplicações baseadas na representação e processamento de conhecimento, notadamente sistemas especialistas. Pelo seu estilo declarativo, Programação em Lógica se presta ao desenvolvimento de tais aplicações.

8.2 SINTAXE E SEMÂNTICA DECLARATIVA

Os principais conceitos relativos à sintaxe de Programação em Cláusulas são bastante simples:

Definição 8.1:

- (a) Um *programa em cláusulas*, ou simplesmente um *programa*, é um conjunto finito **C** de cláusulas. O *alfabeto* de **C** é o alfabeto de primeira ordem cujos símbolos não-lógicos são aqueles ocorrendo nas cláusulas de **C**.
- (b) Uma *consulta* a um programa **C** é uma fórmula **Q** sobre o alfabeto de **C**.
- (c) Uma *resposta* de uma consulta **Q** a um programa **C** é um conjunto de substituições $\beta = \{\beta_1, \dots, \beta_n\}$ de algumas das variáveis livres de **Q** por termos sobre o alfabeto de **C**.
- (d) Uma *resposta simples* é uma resposta com apenas uma substituição.
- (e) Uma *resposta genérica* é uma resposta que não é simples.

Respostas simples são chamadas de definidas e respostas genéricas de indefinidas em Reiter [1978].

Assim, por exemplo, se $\beta_1 = \{x/a, y/f(z)\}$ e $\beta_2 = \{x/b\}$ são substituições, o conjunto $\{\beta_1, \beta_2\}$ é uma resposta genérica e o conjunto unitário $\{\beta_1\}$ é uma

resposta simples. Identificaremos ainda uma resposta simples com a sua única substituição. Logo diremos, por exemplo, que β_1 é uma resposta simples. Em particular, a substituição vazia é também uma resposta simples.

Recorde que a expressão $\forall Q$ denota o fecho universal de uma fórmula Q e que $Q\beta_i$ denota a fórmula resultante de aplicar as substituições simples em β_i às variáveis livres de Q .

Definição 8.2:

Uma resposta $\beta = \{\beta_1, \dots, \beta_n\}$ de uma consulta Q a um programa C está *correta* se e somente se C implica logicamente $\forall(Q\beta_1 \vee \dots \vee Q\beta_n)$.

As noções de consulta e resposta são uma das principais características de Programação em Lógica, não estando presentes no desenvolvimento clássico da Lógica. Assim, convém ilustrar estes conceitos em detalhe.

As consultas mais elementares são aquelas expressas por fórmulas sem variáveis livres e capturam apenas a situação típica de prova automática de teoremas: dada uma sentença Q , testar se Q é um teorema ou não de um conjunto de sentenças (no caso, das cláusulas de um programa). Tais consultas ou não tem respostas corretas ou tem apenas uma resposta correta, que é representada pela substituição vazia. A inexistência de uma resposta correta deve ser entendida como indicando " Q não é consequência lógica de C " e o fato da substituição vazia ser uma resposta correta deve ser entendido como " Q é consequência lógica de C ".

Exemplo 8.1:

Considere o seguinte programa representando um estado do dicionário de programas e arquivos:

Programa DICIONARIO

1. *programa(a,fortran)*
 2. *programa(b,pascal)*
 3. *programa(c,fortran)*
 4. *arquivo(d,sequencial)*
 5. *arquivo(e,direto)*
 6. *chama(a,b)*
 7. *chama(b,c)*
 8. *usa(a,d)*
 9. *usa(b,d) usa(b,e)*
-

Note que a última cláusula indica que o programa "b" usa ou o arquivo "d" ou o arquivo "e".

A fórmula abaixo

$$Q_1. \quad usa(a,d) \vee usa(b,d)$$

descreve uma consulta a este programa que, intuitivamente, pergunta se o programa "a" ou o programa "b" usam o arquivo "d" ou não. Como Q_1 não tem variáveis livres, a única resposta simples possível é a substituição vazia, que neste caso é correta, pois DICIONARIO implica logicamente Q_1 .

As noções de consulta e de resposta não se restringem, porém, a modelar apenas testes de hipóteses acerca de uma teoria. Uma resposta simples de uma consulta definida por uma fórmula Q com variáveis livres consiste de uma substituição das variáveis de Q por termos e representa informação extraída do conteúdo do programa. O exemplo seguinte ilustra o caso mais elementar de uma resposta simples consistindo de substituições de variáveis por termos sem ocorrências de variáveis.

Exemplo 8.2:

A fórmula abaixo

$$Q_2. \quad usa(a,x) \wedge arquivo(x,sequencial)$$

descreve uma consulta a DICONARIO que, intuitivamente, pergunta pelos arquivos seqüenciais que o programa "a" usa. Como símbolos funcionais não ocorrem em DICONARIO, as possíveis respostas simples (corretas e incorretas) de Q_2 são substituições da forma $\{x/t\}$ onde t é um elemento do conjunto de variáveis e constantes do alfabeto de DICONARIO, listado abaixo:

$$T = \{a,b,c,d,e,fortran,pascal,sequencial,direto\} \cup \{x,y,\dots\}$$

Porém, a única resposta simples correta da consulta Q_2 é $\beta = \{x/d\}$. De fato, a fórmula " $usa(a,d) \wedge arquivo(d,sequencial)$ " é consequência lógica de DICONARIO, mas nenhuma outra fórmula deste tipo, com "d" substituída por outro termo em T , tem esta propriedade.

Já uma resposta simples indicando substituições de variáveis por termos com ocorrências de variáveis representa uma inferência simples a partir do programa. De fato, seja β uma resposta simples de uma consulta Q a um programa C . Suponha que β possui pelo menos um par do tipo x/t tal que t possui ocorrências de variáveis. Então, β será correta, por definição, se C implicar logicamente $\forall(Q\beta)$.

Uma resposta genérica $\beta = \{\beta_1, \dots, \beta_n\}$ indica que β_1, \dots, β_n são respostas simples alternativas para a consulta em questão. Respostas genéricas capturam então informação indefinida, sob forma de listas de alternativas, permitindo dar respostas parciais a consultas que de outra forma poderiam não ter nenhuma resposta simples correta.

Exemplo 8.3:

A fórmula abaixo

$$Q_3. \quad usa(b,x)$$

descreve uma consulta a DICONARIO que, intuitivamente, pergunta pelos arquivos que o programa "b" usa. Novamente, as

possíveis respostas simples (corretas e incorretas) de Q_3 seriam substituições $\{x/t\}$ onde t pertence a T , e as respostas genéricas seriam conjuntos de tais substituições. Não há respostas simples corretas para Q_3 . Porém, o conjunto de substituições $\beta = \{\{x/d\}, \{x/e\}\}$ é uma resposta correta pois DICIONARIO implica logicamente a fórmula " $usa(b,d) \vee usa(b,e)$ ".

Exemplo 8.4: (Chang e Lee [1973])

Considere o seguinte cenário:

- F₁. Se João tem menos do que 5 anos de idade, deverá tomar o remédio "a";
- F₂. Se João tem 5 anos de idade ou mais, deverá tomar o remédio "b";

e a seguinte pergunta:

PE. "Qual remédio João deverá tomar?".

Interprete $M(x)$ como "x tem menos do que 5 anos" e $T(x,y)$ como "x deve tomar y". O seguinte programa captura então F₁ e F₂:

1. $T(\text{João},a) \neg M(\text{João})$
2. $T(\text{João},b) M(\text{João})$

Note que a primeira cláusula é equivalente à fórmula " $M(\text{João}) \rightarrow T(\text{João},a)$ " e que a segunda é equivalente a " $\neg M(\text{João}) \rightarrow T(\text{João},b)$ ".

A pergunta, por sua vez, se transforma na consulta:

3. $T(\text{João},x)$

Não há respostas simples corretas a esta consulta, mas a resposta genérica $\{\{x/a\}, \{x/b\}\}$ é correta pois (1) e (2) implicam logicamente " $T(\text{João},a) \vee T(\text{João},b)$ ".

Para terminar esta seqüência de exemplos, convém mencionar que não há restrições sobre o uso da negação para formular consultas. Porém, o programa deve estar preparado para responder a tais consultas de acordo com o significado pretendido.

Exemplo 8.5:

A fórmula abaixo

$$Q_4. \quad \neg \text{usa}(a,x)$$

também descreve uma consulta a DICONARIO perguntando intuitivamente pelos objetos que "a" não usa.

As possíveis respostas simples (corretas e incorretas) de Q_4 também seriam substituições $\{x/t\}$ onde t é tirado do conjunto T , mas não há respostas simples corretas pois o programa DICONARIO não implica logicamente " $\neg \text{usa}(a,t)$ ", onde t é qualquer dos termos em T . Também não há respostas genéricas corretas.

8.3 SEMÂNTICA PROCEDIMENTAL INDUZIDA POR RESOLUÇÃO

Esta seção investiga em detalhe como extrair respostas corretas de refutações no sistema formal da resolução. A seção 8.3.1 introduz a estratégia de extração de respostas informalmente, deixando para a seção 8.3.2 o desenvolvimento formal. Em particular, os resultados da seção 8.3.2 mostram que o sistema formal da resolução e, em particular, o método da resolução linear realmente induzem uma semântica procedural adequada para Programação em Cláusulas.

8.3.1 Exemplos de Computação de Respostas

Recorde do Capítulo 4 que R-dedução e R-refutação são abreviações para dedução e refutação no sistema formal da resolução.

Observemos inicialmente que, como o sistema formal da resolução é correto e refutacionalmente completo, $\beta = \{\beta_1, \dots, \beta_n\}$ é uma resposta correta de uma consulta Q a um programa C se e somente se existe uma R-refutação a partir de $C \cup D$, onde D é uma representação clausal de $\neg \forall(Q\beta_1 \vee \dots \vee Q\beta_n)$. Esta caracterização é inadequada pois equivaleria, não a utilizar o sistema formal da resolução para gerar respostas corretas, mas a arbitrariamente gerar uma resposta β e então usar resolução para determinar se β é correta ou não.

A geração de respostas corretas por resolução de fato segue uma linha diferente. Dada uma R-refutação R a partir de C e da representação

clausal de $\neg Q$, é possível mostrar que as substituições efetuadas nas variáveis livres de Q durante a construção de R formam uma resposta correta de Q a C .

Exemplo 8.6:

Seja **C** o seguinte programa:

1. *chama(a,b)*
2. *usa(a,c)*
3. *depende(x₁,y₁)* \neg *usa(x₁,y₁)*
4. *depende(x₂,y₂)* \neg *chama(x₂,y₂)*
5. *depende(x₃,y₃)* \neg *chama(x₃,z₃)* \neg *depende(z₃,y₃)*

e **Q** a seguinte consulta a este programa:

depende(a,w)

Uma R-refutação a partir de **C** e da representação clausal de $\neg Q$ seria (cada linha inclui também os unificadores usados na derivação):

1. *chama(a,b)*
2. *usa(b,e)*
3. *depende(x₁,y₁)* \neg *chama(x₁,y₁)*
4. *depende(x₂,y₂)* \neg *usa(x₂,y₂)*
5. *depende(x₃,y₃)* \neg *chama(x₃,z₃)* \neg *depende(z₃,y₃)*
6. \neg *depende(a,w)*
7. \neg *chama(a,z₃)* \neg *depende(z₃,w)* . 6a, 5a, {x₃/a,y₃/w}
8. \neg *depende(b,w)* . 7a, 1a, {z₃/b}
9. \neg *usa(b,w)* . 8a, 4a, {x₂/b,y₂/w}
10. \square . 9a, 2a, {w/e}

A substituição $\beta = \{w/e\}$ é então a resposta computada pela R-refutação, pois a variável "w" da consulta foi substituída pela constante "e" no passo (10). Note que, de fato, β é uma resposta correta pois **C** implica logicamente "*depende(a,e)*".

Porém, ao contrário do que sugere o exemplo anterior, em geral é bastante difícil determinar quais são as substituições efetuadas nas variáveis livres da consulta. Essencialmente os problemas são de três naturezas. Primeiro, tais substituições são oriundas de aplicações da regra da resolução e, portanto, envolvem renomeações efetuadas em cláusulas, unificadores mais

gerais usados para gerar fatores e unificadores mais gerais usados para gerar resolventes binários. Segundo, a representação clausal da negação da consulta pode consistir de mais de uma cláusula. Por fim, tais cláusulas podem ser usadas repetidas vezes na R-refutação.

O resto desta sub-seção exemplifica como uma idéia simples, devida originalmente a Green [1969], evita todos estes problemas, deixando para a sub-seção seguinte as definições e resultados formais.

Exemplo 8.7:

Considere novamente o cenário do Exemplo 8.4, repetido abaixo:

- F₁. Se João tem menos do que 5 anos de idade, deverá tomar o remédio "a";
- F₂. Se João tem 5 anos de idade ou mais, deverá tomar o remédio "b";

Considere a pergunta:

PE: "Qual remédio João deverá tomar?"

Interprete M(x) como "x tem menos do que 5 anos" e T(x,y) como "x deve tomar y". Dentro desta interpretação, o programa **C** abaixo captura então os fatos F₁ e F₂:

- 1. T(João,a) \neg M(João)
- 2. T(João,b) M(João)

e a consulta Q abaixo captura a pergunta:

Q. T(João,x)

Seja D a cláusula " \neg T(João,x)". Considere a seguinte R-refutação a partir de **C** \cup {D}:

1. $T(\text{João}, a) \neg M(\text{João})$
2. $T(\text{João}, b) M(\text{João})$
3. $\neg T(\text{João}, x)$
4. $\neg M(\text{João})$. 3a, 1a
5. $T(\text{João}, b)$. 4a, 2b
6. \square . 5a, 3a

Note que a cláusula (3) é utilizada duas vezes na refutação, o que significa que, para extrair uma resposta da refutação, temos que acompanhar dois conjuntos de substituições simultaneamente.

Para evitar este problema, introduziremos na cláusula **D** um *literal de resposta*, " $r(x)$ ", que apenas registrará as substituições efetuadas em x , a variável da consulta, onde " r " é um novo símbolo predicativo. Além disto, terminaremos a R-refutação quando gerarmos uma cláusula só com literais da forma " $r(t)$ ".

A R-refutação anterior se transforma então na R-dedução abaixo:

1. $T(\text{João}, a) \neg M(\text{João})$
2. $T(\text{João}, b) M(\text{João})$
3. $\neg T(\text{João}, x) r(x)$
4. $\neg M(\text{João}) r(a)$. 3a, 1a
5. $T(\text{João}, b) r(a)$. 4a, 2b
6. $r(a) r(b)$. 5a, 3a

Como esta R-dedução termina na cláusula " $r(b) r(a)$ ", o conjunto de substituições $\beta = \{\{x/b\}, \{x/a\}\}$ é uma resposta correta para a consulta **Q** a **C**. De fato, **C** implica logicamente " $T(\text{João}, a) \vee T(\text{João}, b)$ ".

Exemplo 8.8: (Chang e Lee [1973])

Este exemplo ilustra o uso de resolução para geração de planos, que será abordado novamente no Capítulo 12.

Suponha que um objeto d está inicialmente na posição A e que não há uma forma direta de mover d da posição A para uma posição C . Suponha ainda que existe uma ação f capaz de mover d de A para uma posição B e uma ação g capaz de mover d de B para C . A pergunta em questão é "Como mover d de A para C ?".

Para axiomatizar este problema, introduziremos inicialmente um alfabeto cujos símbolos não-lógicos são as constantes " e_0 ", "A", "B" e "C", o símbolo predicativo binário "p" e dois símbolos funcionais unários "f" e "g". Usaremos estes símbolos com o seguinte significado pretendido:

constantes e_0 é o estado inicial do sistema e A, B e C são posições

$p(x,y)$ o objeto está na posição x no estado y

$v=f(u)$ se o objeto está na posição A no estado u, após a ação f, o objeto estará na posição B no estado v; se o objeto está na posição $x \neq A$ no estado u, após a ação f, o objeto continuará na posição x no estado v;

$v=g(u)$ se o objeto está na posição B no estado u, após a ação g, o objeto estará na posição C no estado v; se o objeto está na posição $x \neq B$ no estado u, após a ação g, o objeto continuará na posição x no estado v;

O programa **C** abaixo formaliza então o problema:

1. $p(A,e_0)$
2. $p(B,f(z_1)) \neg p(A,z_1)$
3. $p(C,g(z_2)) \neg p(B,z_2)$

e a consulta **Q** abaixo, a pergunta:

Q. $p(C,z_3)$

Como no exemplo anterior, para computar uma resposta de **Q** a **C**, construiremos primeiro uma representação clausal de $\neg Q$, acrescentando a cada cláusula o literal de resposta:

4. $\neg p(C,z_3) r(z_3)$

Considere a seguinte R-dedução a partir das cláusulas (1) a (4), iniciando em (4):

1. $p(A, e_0)$
2. $p(B, f(z_1)) \neg p(A, z_1)$
3. $p(C, g(z_2)) \neg p(B, z_2)$
4. $\neg p(C, z_3) r(z_3)$
5. $\neg p(B, z_2) r(g(z_2))$. 4a, 3a
6. $\neg p(A, z_1) r(g(f(z_1)))$. 5a, 2a
7. $r(g(f(e_0)))$. 6a, 1a

Como esta R-dedução termina na cláusula " $r(g(f(e_0)))$ ", a substituição $\beta = \{z_3/g(f(e_0))\}$ é uma resposta simples correta para a consulta Q. A resposta β deve ser interpretada como "o estado final z_3 em que o objeto está na posição C, requisitado pela consulta em (4), pode ser atingido a partir do estado inicial e_0 , em que o objeto está na posição A (fato expresso pelo axioma (1)), executando primeiro a ação f sobre o objeto no estado inicial e_0 e, em seguida, executando a ação g sobre o objeto no estado intermediário atingido."

Exemplo 8.9:

Seja **C** o seguinte programa:

1. $chama(a,b) \vee usa(a,c)$

e **Q** a seguinte consulta a este programa:

$chama(a,x) \vee usa(a,y)$

Aplicando a técnica sugerida pelos exemplos anteriores, primeiro temos que obter uma representação clausal de $\neg Q$ e acrescentar a cada cláusula o literal de resposta " $r(x,y)$ ", pois Q agora possui x e y como variáveis livres:

2. $\neg chama(a,x) r(x,y)$
3. $\neg usa(a,y) r(x,y)$

Uma R-dedução a partir destas cláusulas seria:

1. $chama(a,b) \wedge usa(a,c)$
2. $\neg chama(a,x) \wedge r(x,y)$
3. $\neg usa(a,y) \wedge r(x,y)$
4. $chama(a,b) \wedge r(x,c)$. 3a, 1b
5. $r(b,y) \wedge r(x,c)$. 4a, 2a

Logo uma resposta correta seria $\beta = \{ \{x/b, y/y\}, \{x/x, y/c\} \}$, o que de fato é verdade pois **C** implica logicamente

$$(chama(a,b) \vee usa(a,y)) \vee (chama(a,x) \vee usa(a,c))$$

Porém, uma resposta mais adequada poderia ser obtida utilizando-se dois literais de resposta e substituindo-se as cláusulas (2) e (3) por:

- 2'. $\neg chama(a,x) \wedge r_1(x)$
- 3'. $\neg usa(a,y) \wedge r_2(y)$

Uma R-dedução a partir destas cláusulas seria:

1. $chama(a,b) \wedge usa(a,c)$
2. $\neg chama(a,x) \wedge r_1(x)$
3. $\neg usa(a,y) \wedge r_2(y)$
4. $chama(a,b) \wedge r_2(c)$. 3a, 1b
5. $r_1(b) \wedge r_2(c)$. 4a, 2a

Logo uma resposta correta seria $\beta = \{\{x/b, y/c\}\}$, o que de fato é verdade pois **C** implica logicamente

$$chama(a,b) \vee usa(a,c)$$

Os resultados e definições da seção 8.3.2 poderão ser alterados para acomodar este refinamento.

8.3.2 Correção e Completude da Computação de Respostas

Esta seção introduz formalmente o conceito de uma resposta computada a partir de uma dedução no sistema formal da resolução e mostra que toda resposta computada é correta e que, para toda resposta correta, existe uma resposta computada que é mais geral. Na verdade, esta seção estabelece um resultado ainda mais forte afirmando que para toda resposta correta existe uma resposta mais geral computada por uma dedução linear.

Definição 8.3:

Seja \mathbf{C} um programa e \mathbf{Q} uma consulta a \mathbf{C} com n variáveis livres distintas. Um literal R é um *literal de resposta* para \mathbf{C} e \mathbf{Q} se e somente se R é da forma " $r(x_1, \dots, x_n)$ ", onde " r " é um símbolo predicativo n -ário que não ocorre no alfabeto de \mathbf{C} e x_1, \dots, x_n é uma lista das variáveis livres de \mathbf{Q} .

No que se segue, adotaremos as seguintes convenções:

- \mathbf{C} denotará um programa;
- \mathbf{Q} denotará uma consulta com variáveis livres x_1, \dots, x_n ;
- R denotará um literal de resposta para \mathbf{Q} e \mathbf{C} da forma " $r(x_1, \dots, x_n)$ ";
- $\forall P$ denotará o fecho universal de uma fórmula P ;
- $CL(P)$ denotará uma representação clausal de uma fórmula P ;
- se $\beta = \{\beta_1, \dots, \beta_m\}$ for um conjunto de substituições, $P\beta$ denotará a disjunção " $P\beta_1 \vee \dots \vee P\beta_m$ ".

Portanto, " r " não ocorre em \mathbf{C} ou \mathbf{Q} e, se $\beta_i = \{x_1/t_1, \dots, x_n/t_n\}$ é uma substituição, a expressão $R\beta_i$ representa o literal $r(t_1, \dots, t_n)$.

Definição 8.4:

- (a) Uma resposta $\beta = \{\beta_1, \dots, \beta_n\}$ de \mathbf{Q} a \mathbf{C} é *computada* por uma R-dedução \mathbf{D} se e somente \mathbf{D} é uma R-dedução a partir de $\mathbf{C} \cup CL(\forall(Q \rightarrow R))$ terminando na cláusula " $R\beta_1 \dots R\beta_n$ ".
- (b) Uma resposta de \mathbf{Q} a \mathbf{C} é *computada* se e somente se for computada por alguma R-dedução a partir de $\mathbf{C} \cup CL(\forall(Q \rightarrow R))$.

A prova de que toda resposta computada é correta segue do Lema 4.4 e de um lema bastante simples:

Lema 8.1:

$$\mathbf{C} \cup CL(\forall(Q \rightarrow R)) \models \forall R\beta \quad \text{se e somente se} \quad \mathbf{C} \models \forall Q\beta$$

Demonstração

(\rightarrow) Observe inicialmente que $\mathbf{C} \cup CL(\forall(Q \rightarrow R)) \models \forall R\beta$ se e somente se $\mathbf{C} \cup \forall(Q \rightarrow R) \models \forall R\beta$, por definição de representação clausal. Logo, basta mostrar que $\mathbf{C} \cup \forall(Q \rightarrow R) \models \forall R\beta$ implica $\mathbf{C} \models \forall Q\beta$. Suponha que $\mathbf{C} \cup \forall(Q \rightarrow R) \models \forall R\beta$. Seja I um modelo de \mathbf{C} . Como r não ocorre

em **C** ou **Q**, é possível estender **I** para um modelo **J** de $\forall(Q \equiv R)$. Então **J** é um modelo de **C** e de $\forall(Q \rightarrow R)$. Logo, **J** é um modelo de $\forall R\beta$, por suposição. Como $\forall(Q \equiv R)$ e $\forall R\beta$ implicam logicamente $\forall Q\beta$, **J** é então um modelo de $\forall Q\beta$. Mas, como "r" não ocorre em **Q** e **I** é igual a **J** em todos os símbolos não-lógicos, exceto "r", **I** é um modelo de $\forall Q\beta$. Assim, todo modelo de **C** é também um modelo de $\forall Q\beta$, ou seja, **C** $\models \forall Q\beta$.

(\leftarrow) Segue de forma semelhante.

Teorema 8.1: (Correção das Respostas Computadas por Resolução)

Se β é uma resposta computada de **Q** a **C**, então β é uma resposta correta de **Q** a **C**.

Demonstração

Suponha que $\beta = \{\beta_1, \dots, \beta_m\}$ seja uma resposta computada de **Q** a **C**. Então existe uma R-dedução **D** a partir de $\mathbf{C}' = \mathbf{C} \cup CL(\forall(Q \rightarrow R))$ terminando na cláusula " $R\beta_1 \dots R\beta_m$ ". Então, pelo Lema 4.4, **C'** implica logicamente $\forall R\beta$. Logo, pelo Lema 8.1, **C** implica logicamente $\forall Q\beta$, ou seja, β é uma resposta correta de **Q** a **C**.

O converso do Teorema 8.1 exigiria que toda resposta correta θ fosse computada. Porém, sob esta forma, o resultado converso seria muito forte pois uma R-dedução utiliza apenas unificadores mais gerais, gerando desta forma uma resposta computada β possivelmente "mais geral" do que θ . A definição seguinte introduz a noção de resposta mais geral de forma precisa.

Definição 8.5:

Sejam β e θ duas respostas de **Q** a **C**. A resposta θ é *mais geral* do que β se e somente se $\forall Q\theta$ implica logicamente $\forall Q\beta$.

Recorde do Capítulo 5, que RL-dedução e RL-refutação são abreviações para R-dedução linear e R-refutação linear.

O resto desta seção estabelece que, de fato, para toda resposta correta existe uma resposta mais geral computada por uma RL-dedução. Portanto, no contexto de computação de respostas, como no contexto de refutação de cláusulas, basta trabalhar com resolução linear. Este resultado depende de dois lemas auxiliares enunciados a seguir.

Lema 8.2:

Se $\forall R\theta \models \forall R\beta$ então $\forall Q\theta \models \forall Q\beta$

Demonstração

Suponha $\forall R\theta \models \forall R\beta$. Seja I um modelo de $\forall Q\theta$. Como r não ocorre em C ou Q , é possível estender I para um modelo J de $\forall(Q \equiv R)$. Note que J é então também um modelo de $\forall Q\theta$, $\forall(Q\beta \equiv R\beta)$ e $\forall(Q\theta \equiv R\theta)$. Portanto, J é um modelo de $\forall R\theta$ e, pela suposição, também de $\forall R\beta$. Então, J é um modelo de $\forall Q\beta$. Mas " r " não ocorre em Q e I é idêntico a J em todos os símbolos, exceto " r ". Logo, I é um modelo de $\forall Q\beta$.

Lema 8.3:

Seja S um conjunto de cláusulas básicas. Seja D uma RL-refutação a partir de S . Suponha que D possui uma subseqüência $D_{i-1} D_i D_{i+1}$ tal que

$$\begin{aligned}(i) \quad D_{i-1} &= \{L, M\} \cup A \\ D_i &= \{M\} \cup A \cup B \\ D_{i+1} &= A \cup B \cup C\end{aligned}$$

(ii) D_i é obtida resolvendo-se D_{i-1} com uma cláusula $\{L'\} \cup B$, onde os literais escolhidos são L e L' , e D_{i+1} é obtida resolvendo-se D_i com uma cláusula $\{M'\} \cup C$, onde os literais escolhidos são M e M' .

Então existe uma RL-refutação D' a partir de S tal que $D'_k = D_k$, para todo $k \neq i$ e $D'_i = \{L\} \cup A \cup C$.

Demonstração

Suponha que D seja uma RL-refutação a partir de S satisfazendo as condições do lema. Por suposição, o literal M pode ser cancelado com M' . Logo, a cláusula $D'_i = \{L\} \cup A \cup C$ é um resolvente de D_{i-1} e da cláusula $\{M'\} \cup C$. Mas L pode ser cancelado com L' . Logo, a cláusula $D_{i+1} = A \cup B \cup C$ é um resolvente de D'_i e da cláusula $\{L'\} \cup B$. Então existe a RL-refutação D' satisfazendo às condições do lema.

Teorema 8.2: (Completude das Respostas Computadas por Resolução Linear)

Suponha que \mathbf{C} seja um conjunto satisfatível de cláusulas e que \mathbf{Q} seja satisfatível. Se β é uma resposta correta de \mathbf{Q} a \mathbf{C} então existe uma resposta θ de \mathbf{Q} a \mathbf{C} tal que θ é mais geral do que β e θ é computada por uma RL-dedução.

Demonstração

Suponha que $\beta = \{\beta_1, \dots, \beta_m\}$ seja uma resposta correta de \mathbf{Q} a \mathbf{C} . Então \mathbf{C} implica logicamente $\forall Q\beta$. Logo, pelo Lema 8.1, \mathbf{C}' implica logicamente $\forall R\beta$, onde $\mathbf{C}' = \mathbf{C} \cup CL(\forall(Q \rightarrow R))$. Além disto, como \mathbf{C} e \mathbf{Q} são satisfatíveis, \mathbf{C}' é satisfatível.

Seja $\mathbf{R} = \{\{\neg R\beta_1\varphi\}, \dots, \{\neg R\beta_m\varphi\}\}$, onde φ é uma substituição das variáveis de $R\beta$ por constantes de Skolem distintas que não ocorrem em \mathbf{C}' . Então, $\mathbf{C}'' = \mathbf{C}' \cup \mathbf{R}$ é insatisfatível, pois \mathbf{C}' implica logicamente $\forall R\beta$ e \mathbf{R} é uma representação clausal de $\neg \forall R\beta$. Pelo Teorema de Herbrand, há então um conjunto finito \mathbf{B}'' de instâncias básicas de cláusulas em \mathbf{C}'' tal que \mathbf{B}'' é insatisfatível. Como as cláusulas em \mathbf{R} são básicas, $\mathbf{C}'' = \mathbf{C}' \cup \mathbf{R}$ e \mathbf{C}' é satisfatível, temos que $\mathbf{R} \cap \mathbf{B}'' \neq \emptyset$. Além disto, \mathbf{R} é satisfatível. Assim, $\mathbf{B}'' - \mathbf{R}$ é um conjunto de suporte para \mathbf{B}'' . Logo, pelo Corolário 5.2, há então uma RL-refutação \mathbf{D} a partir de \mathbf{B}'' iniciando-se em alguma cláusula \mathbf{C} tal que $\mathbf{C} \in \mathbf{B}'' - \mathbf{R}$. Usando o Lema 8.3 e o fato de \mathbf{D} iniciar em uma cláusula que não pertence a \mathbf{R} , é então possível provar por indução que há uma segunda RL-refutação \mathbf{D}' tal que há um inteiro i tal que todas as cláusulas em \mathbf{D}' a partir de \mathbf{D}'_i são da forma " $r(t_1) \dots r(t_k)$ " e, para todo $j > i$, \mathbf{D}'_j é obtida resolvendo-se \mathbf{D}'_{j-1} contra uma cláusula em \mathbf{R} . Pelo Lema da Promoção para resolução, há então uma RL-refutação \mathbf{E} a partir de \mathbf{C}'' tal que há um inteiro i tal que todas as cláusulas em \mathbf{E} a partir de \mathbf{E}_i são da forma " $r(u_1) \dots r(u_k)$ " e, para todo $j > i$, \mathbf{E}_j é obtida resolvendo-se \mathbf{E}_{j-1} contra uma cláusula em \mathbf{R} . Seja \mathbf{E}' a RL-dedução a partir de \mathbf{C}'' obtida truncando-se o sufixo de \mathbf{E} a partir de \mathbf{E}_{i+1} . Então, se \mathbf{E}'_i é a cláusula " $R\theta_1 \dots R\theta_k$ ", o conjunto de substituições $\theta = \{\theta_1, \dots, \theta_k\}$ é a resposta computada por \mathbf{E}' . Além disto, o sufixo de \mathbf{E} a partir de \mathbf{E}_i é uma RL-refutação a partir de $\mathbf{R}'' = \{\mathbf{E}_i\} \cup \mathbf{R}$, ou seja, \mathbf{R}'' é insatisfatível. Mas \mathbf{E}_i é uma representação clausal de $\forall R\theta$ e \mathbf{R} de $\neg \forall R\beta$. Logo, temos que $\forall R\theta$ implica logicamente $\forall R\beta$. Assim, pelo Lema 8.2, $\forall Q\theta$ implica logicamente $\forall Q\beta$, ou seja, θ é mais geral do que β .

Portanto, os Teoremas da Correção e Completude para computação de respostas implicam em que o sistema formal da resolução e, em particular, o método da resolução linear induzem uma semântica procedural adequada para Programação em Cláusulas no sentido de que toda resposta computada é correta e, para toda resposta correta, existe uma resposta computada mais geral.

8.4 SEMÂNTICA PROCEDIMENTAL INDUZIDA POR ELIMINAÇÃO DE MODELOS

As definições das seções 8.2 e 8.3 se adaptam imediatamente ao contexto do sistema formal da eliminação de modelos introduzido no Capítulo 6, exceto que um programa agora é um conjunto finito de cadeias e que as deduções são neste sistema formal.

Porém, para que a computação de uma resposta se faça corretamente, é necessário alterar as regras de inferência do sistema formal de eliminação de modelos para postergar o mais possível a seleção dos literais sobre o símbolo predicativo usado no literal de resposta escolhido. O exemplo a seguir ilustra este problema.

Exemplo 8.10:

Seja **C** o seguinte programa em cadeias:

1. $T(\text{João}, a) \neg M(\text{João})$
2. $T(\text{João}, b) M(\text{João})$

e **Q** a consulta " $T(\text{João}, x)$ ". Tome o literal de resposta **R** para **C** e **Q** como sendo " $r(x)$ ". Uma representação clausal de " $\forall(Q \rightarrow R)$ " será então:

3. $\neg T(\text{João}, x) r(x)$

Considere a seguinte dedução a partir de **C** e $CL(\forall(Q \rightarrow R))$:

1. $T(\text{João}, a) \neg M(\text{João})$
2. $T(\text{João}, b) M(\text{João})$
3. $\neg T(\text{João}, x) r(x)$
4. $\neg M(\text{João}) [\neg T(\text{João}, a)] r(a)$. 1a
5. $T(\text{João}, b) [\neg M(\text{João})] [\neg T(\text{João}, a)] r(a)$. 2b
6. $r(b) [T(\text{João}, b)] [\neg M(\text{João})] [\neg T(\text{João}, a)] r(a)$. 3a

Note que esta dedução nem termina em uma cadeia da forma $R\beta_1 \dots R\beta_n$, ou seja, não computa nenhuma resposta, nem pode continuar pois nenhuma regra se aplica a (6). De fato, contração não se aplica a (6) pois o primeiro elemento de (6) não é um R-literal. Da mesma forma, extensão ou redução também não se aplicam a (6) pois o seu primeiro literal, "r(b)", não pode ser cancelado contra nenhum literal das cadeias de entrada e nenhum R-literal de (6). Este problema não ocorria no sistema formal da resolução pois a regra da resolução não incluia uma política fixa de seleção de literais.

Para solucionar o problema apontado pelo exemplo anterior, suponha que o literal de resposta R para um programa C e uma consulta Q seja da forma " $r(x_1, \dots, x_n)$ ". Defina um *r-literal* como qualquer literal da forma " $r(t_1, \dots, t_n)$ " e um *rR-literal* como qualquer R-literal da forma " $[r(t_1, \dots, t_n)]$ ". Ao aplicar as regras de extensão, redução ou contração a uma cadeia C , se C contiver algum literal ou R-literal, ignore todos os r-literais ou rR-literais de C ; se C contiver apenas r-literais ou rR-literais de resposta, proceda normalmente.

Esta alteração resolve o problema levantado pelo exemplo anterior, como ilustrado abaixo.

Exemplo 8.11:

Considere novamente o programa C , a consulta Q e o literal de resposta R do exemplo anterior. Considere a seguinte dedução a partir de C e $CL(\forall(Q \rightarrow R))$:

1. $T(\text{João}, a) \neg M(\text{João})$
2. $T(\text{João}, b) M(\text{João})$
3. $\neg T(\text{João}, x) r(x)$
4. $\neg M(\text{João}) [\neg T(\text{João}, a)] r(a)$. 1a
5. $T(\text{João}, b) [\neg M(\text{João})] [\neg T(\text{João}, a)] r(a)$. 2a
6. $r(b) [T(\text{João}, b)] [\neg M(\text{João})] [\neg T(\text{João}, a)] r(a)$. 3a
7. $r(a) r(b)$. CN

Como esta dedução termina na cadeia " $r(b) r(a)$ ", o conjunto de substituições $\beta = \{\{x/b\}, \{x/a\}\}$ é uma resposta correta para a consulta Q a C .

Note que a aplicação da regra de contração à cadeia (6) ignorou o r -literal ocorrendo no início da cadeia. Note ainda que nesta dedução não ocorre nenhum rR -literal. Na verdade, nenhum rR -literal aparecerá em deduções computando respostas pois tais literais só são necessários na prova do Teorema da Completude para respostas computadas por eliminação de modelos, enunciado abaixo.

De forma semelhante à computação de respostas por resolução, podemos estabelecer a correção e completude da computação de respostas por eliminação de modelos, no seguinte sentido.

Teorema 8.3: (Correção da Computação de Respostas por Eliminação de Modelos)

Seja C um programa e Q uma consulta a C . Se β é uma resposta computada de Q a C no sistema formal de eliminação de modelos, então β é uma resposta correta de Q a C .

Teorema 8.4: (Completude da Computação de Respostas por Eliminação de Modelos)

Se β é uma resposta correta de Q a C então existe uma resposta θ de Q a C tal que θ é mais geral do que β e θ é computada por uma dedução no método de eliminação de modelos fraco.

8.5 BANCOS DE DADOS DEDUTIVOS

Esta seção contém um breve comentário sobre bancos de dados dedutivos e deve ser considerada como um apêndice ao capítulo.

A diferença conceitual entre bancos de dados dedutivos e programas em cláusulas é, em princípio, pequena pois ambos são conjuntos de cláusulas. Na prática, bancos de dados dedutivos tendem a ter mais cláusulas definindo fatos acerca do universo de discurso e menos regras definindo propriedades ou fatos derivados do universo e podem conter cláusulas com papéis distintos.

Mais precisamente, um *banco de dados dedutivo* é um conjunto finito **S** de cláusulas, particionado em três subconjuntos:

- cláusulas definindo informação armazenada no banco em um dado instante;
- cláusulas definindo informação derivada;
- cláusulas definindo propriedades invariantes do universo de discurso.

Como estes três subconjuntos possuem papéis diferentes, eles devem estar perfeitamente identificados. O primeiro subconjunto define o *estado* do banco em um dado instante e, naturalmente, pode ser modificado ao longo do tempo. O segundo subconjunto corresponde à definição de *visões*, ou seja, a conjuntos de fatos definidos a partir daqueles efetivamente armazenados. O terceiro subconjunto define *restrições de integridade*, ou seja, cláusulas que devem ser sempre consistentes com os outros dois subconjunto. Este terceiro subconjunto restringe então as possíveis modificações no estado do banco (veja Nicolas e Yasdianian [1983] ou Casanova e Moura [1986] para um discussão completa sobre este ponto).

Como em Programação em Cláusulas, uma consulta a um banco de dados dedutivo é especificada através de uma fórmula. Responder a uma consulta não significa, porém, apenas recuperar dados do banco, mas sim deduzir informação das sentenças que definem o banco. Assim, o processamento de consultas identifica-se com o processo de dedução.

A adoção de cláusulas para representar o estado corrente, informação dedutiva e consultas oferece duas vantagens básicas. Primeiro, embora não seja comum representar estados de um banco de dados através de cláusulas, a flexibilidade assim obtida permite modelar informação de forma frequentemente mais simples que em um modelo de dados tradicional. Segundo, aumenta o poder de consulta pois envolve agora a noção de dedução.

A seqüência de exemplos no resto desta seção ilustra como definir o dicionário de programas e arquivos, descrito na seção 2.5.1, como um banco de dados dedutivo.

O primeiro passo da construção de um banco de dados dedutivo consiste em selecionar um alfabeto de primeira ordem apropriado. Os símbolos não-lógicos deste alfabeto deverão corresponder a conceitos, ações ou relacionamentos importantes do universo de discurso.

Exemplo 8.12:

No caso específico do dicionário, conforme já discutido na seção 2.5.1, um possível alfabeto A conteria, além de constantes, quatro símbolos predicativos binários - "*programa*", "*arquivo*", "*chama*" e "*usa*" - com a seguinte interpretação pretendida:

constantes	possíveis nomes de programas, arquivos, linguagens de programação e métodos de organização de arquivos.
<i>programa(n,m)</i>	<i>n</i> é um programa escrito na linguagem <i>m</i>
<i>arquivo(n,m)</i>	<i>n</i> é um arquivo com organização <i>m</i>
<i>chama(n,m)</i>	o programa <i>n</i> chama o programa <i>m</i>
<i>usa(n,m)</i>	o programa <i>n</i> usa o arquivo <i>m</i>

Para efeitos de comparação, poderíamos tomar um outro alfabeto com um único símbolo predicativo ternário, "*dic*", com a seguinte interpretação pretendida:

<i>dic(programa,n,m)</i>	<i>n</i> é um programa escrito na linguagem <i>m</i>
<i>dic(arquivo,n,m)</i>	<i>n</i> é um arquivo com organização <i>m</i>
<i>dic(chama,n,m)</i>	o programa <i>n</i> chama o programa <i>m</i>
<i>dic(usa,n,m)</i>	o programa <i>n</i> usa o arquivo <i>m</i>

Poderíamos mesmo considerar "*dic*" como um símbolo predicativo unário, onde "*dic*((t.n.m.*nil*))" teria o significado explicado acima, para t igual a "*programa*", "*arquivo*", "*chama*" ou "*usa*". Levando esta linha ao extremo, um estado inteiro do dicionário poderia ser representado por "*dic(L)*", onde L é uma lista de sublistas da forma (t.n.m.*nil*) com o significado anterior.

Um banco de dados dedutivo contém primordialmente um conjunto de cláusulas (sobre o alfabeto escolhido) representando o estado corrente.

Exemplo 8.13:

Os exemplos a seguir, organizados por tipo de informação, ilustram que classes de cláusulas poderiam compor o estado corrente do dicionário:

Informação factual: "*programa(a,pascal)*", indicando que "a" é um programa escrito em Pascal.

Informação genérica: "*chama(p,x)*", indicando que "p" chama todos os programas.

Informação negativa: " $\neg \text{chama}(s,x)$ ", indicando que "s" não chama nenhum programa.

Informação indefinida:

lista de alternativas:

"*programa(b,fortran)* *programa(b,pascal)*", indicando que "b" é um programa escrito ou em Fortran ou em Pascal.

valor desconhecido: "*programa(c,w)*", indicando que "c" é um programa escrito em uma linguagem presentemente desconhecida, mas identificada temporariamente por "w".

Este último exemplo requer comentários adicionais. A sentença " $\exists x(\text{programa}(c,x))$ " indica que o programa "c" existe, mas está escrito em uma linguagem desconhecida. Eliminando o quantificador existencial desta sentença, obtemos a cláusula "*programa(c,w)*", onde "w" é uma constante de Skolem.

Além das cláusulas representando o estado corrente, um banco de dados dedutivos pode conter ainda cláusulas definindo visões. Em geral, a definição de cláusulas com este propósito segue duas etapas: introdução de um novo símbolo predicativo no alfabeto para representar a visão e obtenção da representação clausal da definição da visão. Estas duas etapas correspondem diretamente à introdução de símbolos predicativos por definição em uma teoria (veja a seção 2.4). Porém, elas também cobrem situações mais complexas, como ilustra o item (b) do exemplo a seguir.

Exemplo 8.14:

- (a) Para definir a visão dos programas em Fortran que usam arquivos em acesso direto, inicialmente introduzimos no alfabeto *A* um novo símbolo predicativo unário, "*pgm*". Em seguida, especificamos o axioma de definição desta visão através da seguinte sentença:

$$1. \forall x (pgm(x) \equiv (programa(x,fortran) \wedge \exists y(usa(x,y) \wedge arquivo(y,direto))))$$

Por fim, obtemos a representação clausal de (1):

2. $\neg programa(x,fortran) \neg usa(x,y) \neg arquivo(y,direto) pgm(x)$
3. $\neg pgm(x) programa(x,fortran)$
4. $\neg pgm(x) usa(x,f(x))$
5. $\neg pgm(x) arquivo(f(x),direto)$

Estas quatro cláusulas, quando acrescidas à descrição do dicionário como um banco de dados dedutivo, definem a visão desejada. Note que a cláusula (2) captura uma parte da bicondicional em (1), enquanto que as cláusulas (3), (4) e (5) capturaram a outra parte (veja também a discussão no final da seção 9.5).

- (b) Para definir o conjunto dos pares (x,y) tais que x usa ou chama y , ou x chama transitivamente um programa que usa ou chama y , introduzimos um novo símbolo predicativo binário, "*depende*", cuja definição é:

$$1. \forall x \forall y (depende(x,y) \equiv (usa(x,y) \vee chama(x,y) \vee \exists z(chama(x,z) \wedge depende(z,y))))$$

A representação clausal desta sentença fornece então as cláusulas desejadas:

2. $\neg \text{usa}(x,y) \text{ depende}(x,y)$
3. $\neg \text{chama}(x,y) \text{ depende}(x,y)$
4. $\neg \text{chama}(x,z) \neg \text{depende}(z,y) \text{ depende}(x,y)$
5. $\neg \text{depende}(x,y) \text{ usa}(x,y) \text{ chama}(x,y) \text{ chama}(x,f(x,y))$
6. $\neg \text{depende}(x,y) \text{ usa}(x,y) \text{ chama}(x,y) \text{ depende}(f(x,y),y)$

Observe que (1) não é um axioma de definição legítimo pois o símbolo definido, "*depende*", ocorre dos dois lados. Porém, as cláusulas de (2) a (6) corretamente representam (1) e devem fazer parte da descrição do dicionário como um banco de dados dedutivo para que se possa utilizar "*depende*".

Por fim, um banco de dados dedutivo pode conter cláusulas representando restrições de integridade do universo de discurso. Freqüentemente restrições de integridade aparecem sob forma de condicionais, podendo ser expressas diretamente em cláusulas. De fato, basta observar que uma fórmula da forma " $\forall(B_1 \wedge \dots \wedge B_n \rightarrow A_1 \vee \dots \vee A_m)$ " é representada diretamente pela cláusula " $A_1 \dots A_m \neg B_1 \dots \neg B_n$ ". Porém, deve-se tomar cuidado ao representar restrições que envolvam negação ou quantificação. Em ambos os casos, é recomendável escrever primeiro a sentença de primeira ordem equivalente para em seguida obter a representação clausal.

Exemplo 8.15:

Considere uma restrição de integridade indicando que todo programa Fortran só usa arquivos sequenciais ou em acesso direto. Esta restrição é capturada de forma simples pela cláusula:

1. $\neg \text{programa}(x,\text{fortran}) \neg \text{usa}(x,y)$
 $\text{arquivo}(y,\text{sequencial}) \text{ arquivo}(y,\text{direto})$

Suponha que a restrição acima seja substituída por outra indicando que todo programa Fortran não usa arquivos indexados sequenciais. Esta restrição é expressa pela seguinte sentença:

2. $\forall x(\text{programa}(x,\text{fortran}) \rightarrow \neg \exists y(\text{usa}(x,y) \wedge \text{arquivo}(y,\text{indexado})))$

cuja representação clausal é:

3. $\neg \text{programa}(x, \text{fortran}) \neg \text{usa}(x, y) \neg \text{arquivo}(y, \text{indexado})$

Considere uma restrição indicando que se x usa y então x deve ser um programa e y um arquivo. Esta restrição é expressa pela seguinte sentença:

4. $\forall x \forall y (\text{usa}(x, y) \rightarrow \exists z (\text{programa}(x, z)) \wedge \exists w (\text{arquivo}(y, w)))$

cuja representação clausal consiste do par de cláusulas:

5. $\neg \text{usa}(x, y) \text{ programa}(x, f(x))$

6. $\neg \text{usa}(x, y) \text{ arquivo}(y, g(y))$

Note que f e g são funções de Skolem introduzidas ao se eliminar os quantificadores $\exists z$ e $\exists w$, respectivamente.

NOTAS BIBLIOGRÁFICAS

Uma excelente introdução histórica a Programação em Lógica pode ser encontrada em Robinson [1983]. O conceito de resposta, formalizado na seção 8.2, encontra-se parcialmente em Chang e Lee [1973]. Bancos de dados dedutivos, o tópico da seção 8.5., são discutidos, por exemplo, em Gallaire, Minker e Nicolas [1984], Kunifugi e Yokota [1982], Lloyd [1983], Lloyd e Topor [1985a,b], Nicolas e Yasdanian [1983], Parker et all [1984] e Reiter [1984].

O texto pioneiro em Programação em Lógica, Kowalski [1974a], ainda é bastante atual, discutindo como formalizar em Lógica vários problemas de Inteligência Artificial e Ciência da Computação em geral. Hogger [1984] oferece uma introdução a Programação em Lógica, cobrindo inclusive o importante problema da construção de programas em lógica. Os outros textos comumente encontrados são em geral sobre a linguagem Prolog (ver notas bibliográficas do Capítulo 10). Um sistema para Programação em Cláusulas Genéricas é descrito em Vieira [1986].

Uma bibliografia bastante extensa sobre Programação em Lógica até o começo de 1984 pode ser encontrada em Poe et alli [1984].

CAPÍTULO 9: PROGRAMAÇÃO EM CLÁUSULAS DEFINIDAS

Este capítulo considera uma segunda variante de Programação em Lógica em que os programas são conjuntos finitos de cláusulas definidas. A seção 9.2 introduz os conceitos de programa, consulta e resposta correta. A seção 9.3 introduz uma semântica declarativa alternativa para esta variante baseada na noção de modelo mínimo de Herbrand. A seção 9.4 descreve como computar respostas corretas através do método de resolução-LSD. A seção 9.5 considera uma extensão de Programação em Cláusulas Definidas através da noção de negação por falha finita. Finalmente, a seção 9.6 trata da especificação de algoritmos através de cláusulas definidas.

As seções recomendadas para cada nível de leitura são:

nível introdutório: 9.2, 9.4.1

nível intermediário: 9.2, 9.4.1, 9.5

9.1 INTRODUÇÃO

Este capítulo considera uma segunda variante de Programação em Lógica em que os programas são conjuntos finitos de cláusulas definidas e consultas são simples conjunções de fórmulas atômicas. Apesar de ser um

caso particular de Programação em Cláusulas, esta variante merece um estudo em separado por vários motivos, abordados a seguir.

Em primeiro lugar, por um resultado apresentado na seção 9.2, Programação em Cláusulas Definidas dispensa o conceito de resposta genérica.

Em segundo lugar, conforme discutido em detalhe na seção 9.3, Programação em Cláusulas Definidas possui uma semântica declarativa alternativa que explora o fato de que todo conjunto de cláusulas definidas possui um modelo mínimo de Herbrand. Intuitivamente, isto significa que um programa pode ser visto como uma coleção de implicações definindo recursivamente conjuntos de fatos.

Terceiro, esta variante permite computar respostas corretas através do método de resolução-LSD. Este é o tópico da seção 9.4. Tal método é extremamente simples e leva à implementação de procedimentos de refutação muito eficientes, conforme adiantado na seção 7.2. Além deste ponto importante, conforme visto na seção 7.4, é possível dar a resolução-LSD o caráter de um método de resolução de problemas interpretando-se uma cláusula definida como uma regra de reescrita reduzindo um problema a outros mais simples. Esta interpretação intuitiva facilitou bastante a divulgação de Programação em Lógica e é devida a Kowalski (veja principalmente Kowalski [1974]).

Finalmente, o conceito de Programação em Cláusulas Definidas oferece uma base teórica para a linguagem Prolog, o assunto da Parte IV do texto. Na verdade, o sucesso de Prolog deve-se em grande parte ao reconhecimento da importância de resolução-LDS como um método de resolução de problemas, aliado à criação de procedimentos de refutação especialmente eficientes.

O uso apenas de cláusulas definidas impõe no entanto limitações importantes de expressividade, conforme já discutido no Capítulo 7. A seção 9.5 explora então uma outra variante de Programação em Lógica baseada no uso de cláusulas pseudo-definidas.

O desenvolvimento de Programação em Cláusulas Definidas confunde-se com o desenvolvimento da linguagem Prolog, ou mesmo, erroneamente, com Programação em Lógica. Esta confusão teve suas origens nos trabalhos de Kowalski, principalmente Kowalski [1979b], que propunham a idéia de que a descrição de um algoritmo pode ser desdobrada em uma

componente lógica, que especifica a semântica do algoritmo, e em uma componente de controle, que descreve como as definições da componente lógica deverão ser usadas. Apesar da grande influência sobre o desenvolvimento de Programação em Lógica, estes trabalhos se restringiam apenas à variante de cláusulas definidas para especificar a componente lógica e ao uso de resolução-LSD como componente de controle. A seção 9.6 aborda esta proposta em mais detalhe.

9.2 SINTAXE E SEMÂNTICA DECLARATIVA

Recorde da Seção 7.2 que uma cláusula definida é uma expressão da forma " $L \leftarrow M_1 \dots M_n$ " ou da forma " $L \leftarrow$ " e que uma cláusula-objetivo é ou a cláusula vazia ou uma expressão da forma " $\leftarrow M_1 \dots M_n$ ", onde L, M_1, \dots, M_n são literais positivos.

As noções de programa, consulta e resposta são bastante semelhantes às do Capítulo 8:

Definição 9.1:

- (a) Um *programa em cláusulas definidas* é um conjunto finito de cláusulas definidas.
- (b) Uma *consulta* a um programa em cláusulas definidas C é uma conjunção de literais positivos sobre o alfabeto de C .
- (c) Uma *resposta* de uma consulta Q a um programa em cláusulas definidas C é um conjunto $\beta = \{\beta_1, \dots, \beta_n\}$ de substituições de algumas das variáveis de Q por termos sobre o alfabeto de C .
- (d) Uma *resposta simples* é uma resposta com apenas uma substituição.
- (e) Uma *resposta genérica* é uma resposta que não é simples.

Quando for claro que o contexto é o de Programação em Cláusulas Definidas, usaremos apenas os termos 'programa' e 'consulta'.

Como na seção 8.3, se P é uma fórmula, denote o fecho universal de P por " $\forall P$ " e, se $\beta = \{\beta_1, \dots, \beta_m\}$ for um conjunto de substituições, denote a disjunção " $P\beta_1 \vee \dots \vee P\beta_m$ " por $P\beta$.

A semântica declarativa de Programação em Cláusulas naturalmente se aplica ao caso especial de Programação em Cláusulas Definidas, em vista

da Definição 7.3. Logo, a noção de resposta correta aplica-se sem modificações:

Definição 9.2:

Uma resposta $\beta = \{\beta_1, \dots, \beta_n\}$ de uma consulta Q a um programa C está correta se e somente se C implica logicamente $\forall(Q\beta_1 \vee \dots \vee Q\beta_n)$.

Porém, programas em cláusulas definidas possuem propriedades semânticas adicionais não compartilhadas por programas em cláusulas genéricas. Nesta seção enunciaremos apenas uma propriedade indicando que a noção de resposta genérica é de certa forma desnecessária.

Teorema 9.1:

Se $\beta = \{\beta_1, \dots, \beta_n\}$ é uma resposta genérica correta de Q a C , então existe $i \in [1, n]$ tal que β_i é uma resposta simples correta de Q a C .

Demonstração

Seja C um programa em cláusulas definidas e Q uma consulta a C da forma " $Q_1 \wedge \dots \wedge Q_m$ ". Suponha que $\beta = \{\beta_1, \dots, \beta_n\}$ seja uma resposta genérica correta de Q a C . Seja D a cláusula " $\neg Q_1 \dots \neg Q_m$ ". Seja φ uma substituição das variáveis livres de $Q\beta$ por constantes distintas que não ocorrem em Q ou C . Então o conjunto de cláusulas $D = \{D\beta_1\varphi, \dots, D\beta_n\varphi\}$ é uma representação clausal de $\neg \forall Q\beta$ pois, pela convenção notacional, $\neg \forall Q\beta$ denota o fecho existencial da fórmula

$$(\neg Q_1 \vee \dots \vee \neg Q_m)\beta_1 \wedge \dots \wedge (\neg Q_1 \vee \dots \vee \neg Q_m)\beta_n$$

Mapeie cada cláusula definida em C na sua cláusula equivalente (isto requer apenas uma mudança de notação). Seja E o conjunto das cláusulas assim obtidas. Como β é uma resposta correta de Q a C , C implica logicamente $\forall Q\beta$. Logo, pela Definição 7.3, E implica logicamente $\forall Q\beta$. Portanto, $E' = E \cup D$ é insatisfatível.

Como todas as cláusulas em E possuem exatamente um literal positivo, E é satisfatível. Logo $E' - D$ é satisfatível. Pelo Corolário 5.2, existe então uma refutação linear R a partir de E' iniciando-se em alguma cláusula D' de D . Suponha que D' seja a cláusula $D\beta_i\varphi$, para algum i . Como D' só possui literais negativos e E' possui apenas cláusulas com no máximo um literal positivo, todas as cláusulas derivadas em R só terão literais negativos. Mas não é possível resolver duas cláusulas apenas com literais

negativos. Portanto, todas as cláusulas derivadas em \mathbf{R} são obtidas resolvendo-se a cláusula anterior contra uma cláusula em \mathbf{E} . Logo, \mathbf{R} é então uma refutação linear a partir de $\mathbf{E}'' = \mathbf{E} \cup \{\mathbf{D}'\}$. Portanto, \mathbf{E}'' é insatisfatível.

Mas \mathbf{D}' é a cláusula $D\beta_i\varphi$, ou seja, \mathbf{D}' é a representação clausal de $\neg\forall Q\beta_i$. Logo, como \mathbf{E}'' é insatisfatível, \mathbf{E} implica logicamente $\forall Q\beta_i$. Portanto, novamente pela Definição 7.3, \mathbf{C} implica logicamente $\forall Q\beta_i$. Ou seja, β_i é uma resposta simples correta de \mathbf{Q} a \mathbf{C} .

Em vista deste resultado, podemos nos restringir apenas a respostas simples no contexto de Programação em Cláusulas Definidas, em contraste com a situação em Programação em Cláusulas.

9.3 SEMÂNTICA DO MODELO MÍNIMO

Cláusulas definidas possuem um grupo de propriedades importantes que levam a uma nova interpretação semântica para Programação em Cláusulas Definidas.

Recorde da seção 3.4 que o universo de Herbrand de um conjunto de cláusulas \mathbf{S} é o conjunto $UH[\mathbf{S}]$ dos termos sem variáveis gerados a partir das constantes (ou de uma constante escolhida arbitrariamente, se nenhuma ocorrer em \mathbf{S}) e dos símbolos funcionais ocorrendo em \mathbf{S} . Recorde ainda que a base de Herbrand para \mathbf{S} é o conjunto $BH[\mathbf{S}]$ de todas as fórmulas atômicas da forma " $p(t_1, \dots, t_n)$ ", onde p é um símbolo predicativo n -ário ocorrendo em \mathbf{S} e t_1, \dots, t_n são termos em $UH[\mathbf{S}]$, para todo $n > 0$. Recorde ainda da seção 3.5 que uma H -interpretação de \mathbf{S} é um subconjunto de $BH[\mathbf{S}]$.

Todas estas definições e todos os resultados do Capítulo 3 aplicam-se a cláusulas de Horn via a Definição 7.3. Podemos provar ainda as seguintes propriedades adicionais, explorando a sintaxe especial de cláusulas definidas:

Proposição 9.1:

Seja \mathbf{S} um conjunto de cláusulas definidas. Uma H -interpretação \mathbf{H} para \mathbf{S} é um H -modelo para \mathbf{S} se e somente se, para toda instância básica $L \leftarrow M_1 \dots M_n$ de uma cláusula definida em \mathbf{S} , se M_1, \dots, M_n pertencem a \mathbf{H} , então L pertence a \mathbf{H} .

Proposição 9.2:

Seja S um conjunto de cláusulas definidas.

- (a) A base de Herbrand de S é um H-modelo de S .
- (b) A interseção de todos os H-modelos de S é um H-modelo de S .

Note que este resultado contrasta com a situação geral pois um conjunto de cláusulas arbitrárias pode não ter um H-modelo ou pode possuir dois H-modelos cuja interseção é vazia. Por exemplo, o conjunto consistindo das cláusulas " $p(a)$ " e " $\neg p(a)$ " não tem H-modelos. O conjunto consistindo apenas da cláusula " $p(a) \ p(b)$ " por sua vez possui dois H-modelos, $\{p(a)\}$ e $\{p(b)\}$, cuja interseção é vazia.

Como consequência da Proposição 9.2, o conjunto de H-modelos de um conjunto de cláusulas definidas tem um mínimo (único) com relação à ordem parcial induzida pela relação de inclusão de conjuntos.

Definição 9.3:

O *H-modelo mínimo* de um conjunto de cláusulas definidas S , denotado por $M[S]$, é a interseção de todos os H-modelos de S .

Em vista da Proposição 9.2, podemos então legitimamente introduzir a *semântica do modelo mínimo* para Programação em Cláusulas Definidas que mapeia cada programa em cláusulas definidas C no H-modelo mínimo de C . Esta semântica induz então uma outra noção de resposta correta:

Definição 9.4:

Uma resposta simples β de uma consulta Q a um programa em cláusulas definidas C está *correta com relação ao H-modelo mínimo* (ou *H-correta*) se e somente se o modelo mínimo de C é um modelo da sentença $\forall Q\beta$.

O exemplo abaixo ilustra esta nova definição semântica.

Exemplo 9.1:

O programa em cláusulas definidas abaixo representa um estado do dicionário acrescido da definição recursiva da relação "*depende*":

```

programa(a,fortran) ←
programa(b,pascal) ←
programa(c,fortran) ←
arquivo(d,sequencial) ←
arquivo(e,direto) ←
chama(a,b) ←
chama(a,c) ←
usa(a,d) ←
usa(b,e) ←
depende(x,y) ← chama(x,y)
depende(x,y) ← usa(x,y)
depende(x,y) ← depende(x,z) depende(z,y)

```

O H-modelo mínimo deste programa é o seguinte subconjunto da base de Herbrand do programa:

```

{programa(a,fortran), programa(b,pascal), programa(c,fortran),
 arquivo(d,sequencial), arquivo(e,direto),
 chama(a,b), chama(a,c),
 usa(a,d), usa(b,e),
 depende(a,b), depende(a,c), depende(a,d),
 depende(b,e), depende(a,e)}

```

É importante observar que a semântica do modelo mínimo não é equivalente à semântica declarativa, como mostra o seguinte exemplo.

Exemplo 9.2: (Lloyd [1984])

Seja **C** o programa consistindo da cláusula definida " $p(a) \leftarrow$ ". Então, o universo de Herbrand de **C** será o conjunto $UH[\mathbf{C}] = \{a\}$ e o H-modelo mínimo de **C** será o conjunto $M[\mathbf{C}] = \{p(a)\}$. Seja **Q** a consulta a **C** expressa pela cláusula " $p(x)$ ". Então a resposta $\beta = \{x/x\}$ é H-correta pois a sentença " $\forall x(p(x)\beta)$ " é verdadeira em $M[\mathbf{C}]$. Porém, β não é uma resposta correta pois " $\forall x(p(x)\beta)$ " não é consequência lógica de **C**.

A semântica do modelo mínimo tem várias propriedades interessantes, exploradas parcialmente a seguir. O primeiro resultado enuncia uma situação em que os dois conceitos de resposta correta coincidem.

Proposição 9.3:

Seja **C** um programa em cláusulas definidas e **Q** uma consulta a **C**. Seja β uma resposta simples de **Q** a **C** e suponha que $Q\beta$ seja uma fórmula sem variáveis livres. Então, β é uma resposta correta se e somente se é uma resposta H-correta.

O segundo resultado mostra que os literais da base de Herbrand de **C** que são consequência lógica de **C** são exatamente os literais do H-modelo mínimo de **C**.

Proposição 9.4:

Seja **C** um programa em cláusulas definidas. Então

$$M[\mathbf{C}] = \{L \in BH[\mathbf{C}] / L \text{ é consequência lógica de } \mathbf{C}\}$$

Este resultado tem uma interpretação interessante em termos de respostas corretas. Defina uma consulta *elementar* a um programa **C** como uma consulta da forma " $p(x_1, \dots, x_m)$ ", onde " p " é um símbolo predutivo do alfabeto de **C**. Pelo teorema anterior, uma substituição $\beta = \{x_1/t_1, \dots, x_m/t_m\}$ será uma resposta simples correta da consulta elementar " $p(x_1, \dots, x_m)$ " se e somente se " $p(t_1, \dots, t_m)$ " for um literal básico do H-modelo mínimo de **C**. Logo, o conjunto de todas as respostas corretas de todas as consultas elementares de **C** é o H-modelo mínimo de **C**.

Há, porém, uma outra forma mais elegante de computar o H-modelo mínimo de **C** que interpreta as cláusulas em **C** como definições recursivas. Para tornar esta observação precisa, associaremos inicialmente a cada programa uma função definida da seguinte forma.

Definição 9.5:

Seja **C** um programa em cláusulas definidas. A função $T[\mathbf{C}]$ mapeando H-interpretações de **C** em H-interpretações de **C** é definida da seguinte forma:

para toda H-interpretação **H** de **C**, $T[\mathbf{C}](H) = I$ se e somente se

$$I = \{A \in BH[\mathbf{C}] / \text{existe uma instância básica da forma } "L \leftarrow M_1 \dots M_n", \text{ para } n \geq 0, \text{ de uma cláusula em } \mathbf{C} \text{ tal que } M_1, \dots, M_n \text{ pertencem a } H\}$$

Por simplicidade, consideraremos um programa em cláusulas definidas **C** fixo no que se segue e usaremos **M** e **T** em lugar de **M[C]** e **T[C]**, respectivamente.

Intuitivamente, para computar $I = T(H)$, prossiga da seguinte forma:

1. Inicialize **I** com todos os literais básicos **L** tais que "**L** ←" é uma instância de uma cláusula de **C** com o corpo vazio;
2. Construa agora uma instância básica $L \leftarrow M_1 \dots M_n$ de uma cláusula em **C**. Se M_1, \dots, M_n pertencem ao conjunto **H** então adicione **L** a **I**;
3. Repita o passo (2) até que nenhum novo literal básico possa ser acrescentado a **I**;

Defina agora as potências crescentes de **T** de tal forma que, para toda **H**-interpretação **I**:

$$\begin{aligned}(T \uparrow 0)(I) &= \emptyset \\ (T \uparrow n)(I) &= T((T \uparrow (n-1))(I))\end{aligned}$$

Defina $\text{sup}(T)$ como o seguinte subconjunto da base de Herbrand de **C**:

$$\text{sup}(T) = \{L \in BH[C] / L \in (T \uparrow i)(\emptyset), \text{ para algum } i \geq 0\}$$

Podemos provar que $\text{sup}(T)$ é o **H**-modelo mínimo de **C**.

Teorema 9.2:

Seja **C** um programa em cláusulas definidas. Então $M = \text{sup}(T)$.

Demonstração

Observe inicialmente que $\text{sup}(T)$ é um **H**-modelo de **C** por construção. Logo, $M \subseteq \text{sup}(T)$. Provaremos então que $\text{sup}(T) \subseteq M$. Ou seja, provaremos que se $L \in (T \uparrow i)(\emptyset)$ então $L \in M$, para todo $i \geq 0$.

Base: para $i=0$, não há nada a provar pois $(T \uparrow 0)(\emptyset) = \emptyset$.

Passo de Indução: Seja $i > 0$ e suponha que se $L \in (T \uparrow (i-1))(\emptyset)$ então $L \in M$. Seja $L \in (T \uparrow i)(\emptyset)$, ou seja, $L \in T((T \uparrow (i-1))(\emptyset))$. Logo, existe uma instância básica **B** de uma cláusula de **C** tal que **B** é da forma $L \leftarrow M_1 \dots M_k$ e M_1, \dots, M_k pertencem a $(T \uparrow (i-1))(\emptyset)$. Pela hipótese de

indução, B_1, \dots, B_k pertencem a M . Mas M é um modelo de C e, portanto, M é um modelo de qualquer instância de uma cláusula de C , logo de B . Portanto, como M_1, \dots, M_k estão em M , temos que L também pertence a M .

O exemplo abaixo ilustra como usar a função T para computar M e sugere porque T captura a idéia de que as cláusulas de C são uma definição recursiva de M .

Exemplo 9.3:

Considere novamente o mesmo programa do exemplo anterior, repetido abaixo:

```

programa(a,fortran) ←
programa(b,pascal) ←
programa(c,fortran) ←
arquivo(d,sequencial) ←
arquivo(e,direto) ←
chama(a,b) ←
chama(a,c) ←
usa(a,d) ←
usa(b,e) ←
depende(x,y) ← chama(x,y)
depende(x,y) ← usa(x,y)
depende(x,y) ← depende(x,z) depende(z,y)

```

Construiremos o modelo mínimo do programa computando as H-interpretações definidas por $(T^{\uparrow i})(\emptyset)$, para valores crescentes de i .

A construção do H-modelo mínimo do programa parte então do conjunto vazio, pois temos que:

$$(T^{\uparrow 0})(\emptyset) = \emptyset$$

O conjunto de literais definido por $(T^{\uparrow 1})(\emptyset)$ será exatamente o conjunto de todos os literais M tais que existe uma cláusula da forma " $L \leftarrow$ " no programa tal que M é uma instância básica de L . Intuitivamente, tais cláusulas definem os fatos que podemos assumir incondicionalmente:

$$(T \uparrow 1)(\emptyset) = T(\emptyset) =$$

*{programa(a,fortran), programa(b,pascal), programa(c,fortran),
arquivo(d,sequencial), arquivo(e,direto),
chama(a,b), chama(a,c), usa(a,d), usa(b,e)}*

O conjunto $(T \uparrow i)(\emptyset)$, para potencias superiores de i , é construído a partir de $(T \uparrow (i-1))(\emptyset)$ e das cláusulas da forma " $L \leftarrow M_1 \dots M_n$ ". Intuitivamente, tais cláusulas definem recursivamente novos fatos representados por instâncias do literal L a partir de instâncias de M_1, \dots, M_n presentes em $(T \uparrow (i-1))(\emptyset)$. Assim temos que:

$$(T \uparrow 2)(\emptyset) = T((T \uparrow 1)(\emptyset)) =$$

*{programa(a,fortran), programa(b,pascal), programa(c,fortran),
arquivo(d,sequencial), arquivo(e,direto),
chama(a,b), chama(a,c), usa(a,d), usa(b,e),
depende(a,b), depende(a,c), depende(a,d), depende(b,e)}*

$$(T \uparrow 3)(\emptyset) = T((T \uparrow 2)(\emptyset)) =$$

*{programa(a,fortran), programa(b,pascal), programa(c,fortran),
arquivo(d,sequencial), arquivo(e,direto),
chama(a,b), chama(a,c), usa(a,d), usa(b,e),
depende(a,b), depende(a,c), depende(a,d), depende(b,e),
depende(a,e)}*

$$(T \uparrow 4)(\emptyset) = T((T \uparrow 3)(\emptyset)) = (T \uparrow 3)(\emptyset)$$

Como $(T \uparrow 4)(\emptyset) = (T \uparrow 3)(\emptyset)$, podemos parar a construção das potências de T e o último conjunto de literais obtido é o H-modelo mínimo do programa.

Este exemplo ilustra um ponto bastante importante. Usualmente, ao construir um programa em cláusulas definidas, o programador escreve cláusulas definidas como se fossem equações recursivas definindo conjuntos de fatos. Esta forma de interpretar um programa em cláusulas definidas corresponde exatamente à semântica do modelo mínimo, entendida como sugere o exemplo anterior. Porém, como a noção de resposta correta nem sempre coincide com a noção de resposta H-correta,

deve-se tomar o devido cuidado para não confundir completamente as duas semânticas.

9.4 SEMÂNTICA PROCEDIMENTAL INDUZIDA POR RESOLUÇÃO-LSD

Como no contexto do sistema formal da resolução, a resposta computada por uma refutação no método de resolução-LSD continua sendo a composição das substituições feitas nas variáveis da consulta. Porém, como ilustra o exemplo a seguir, o processo de extrair a resposta de uma refutação é bastante simples.

Exemplo 9.4:

Seja **C** o seguinte programa em cláusulas definidas:

1. *chama(a,b) ←*
2. *usa(b,e) ←*
3. *depende(x,y) ← chama(x,y)*
4. *depende(x,y) ← usa(x,y)*
5. *depende(x,y) ← chama(x,z) depende(z,y)*

Seja **Q** a seguinte consulta:

depende(a,w)

A representação clausal de $\neg Q$ será então:

6. $\leftarrow \text{depende}(a,w)$

Seja f a função de seleção padrão (que sempre seleciona o literal mais à esquerda). Considere a seguinte LSD(f)-refutação a partir de **C** $\cup \{D\}$ (cada linha inclui também os unificadores usados na derivação):

1. $chama(a,b) \leftarrow$
2. $usa(b,e) \leftarrow$
3. $depende(x_1,y_1) \leftarrow chama(x_1,y_1)$
4. $depende(x_2,y_2) \leftarrow usa(x_2,y_2)$
5. $depende(x_3,y_3) \leftarrow chama(x_3,z_3) depende(z_3,y_3)$
6. $\leftarrow depende(a,w)$
7. $\leftarrow chama(a,y_1)$. 3, { $x_1/a, w/y_1$ }
8. \square . 1, { y_1/b }

A substituição $\theta = \{w/b\}$, obtida pela composição dos dois unificadores usados na refutação, restrita às variáveis da consulta, será então a resposta computada pela refutação.

Este exemplo ilustra duas propriedades genéricas simples, mas importantes, de Programação em Cláusulas Definidas, discutidas a seguir.

Seja **C** um programa em cláusulas definidas e **Q** uma consulta a **C**. Suponha que **Q** seja da forma " $Q_1 \wedge \dots \wedge Q_m$ ". Seja **D** a cláusula-objetivo " $\leftarrow Q_1 \dots Q_m$ ".

Primeiro, como **D** é uma representação clausal de $\neg Q$ e $\mathbf{C}' = \mathbf{C} \cup \{D\}$ é um conjunto quase-definido, o método de resolução-LSD(*f*) pode ser de fato usado para computar respostas no contexto de Programação em Cláusulas Definidas. Esta observação justifica então a restrição de que consultas devam ser conjunções de literais positivos.

Segundo, a resposta de **Q** a **C** computada por uma refutação **R** corresponde à composição dos unificadores usados em **R**, dispensando assim o uso do literal de resposta. Esta forma simples de obter a resposta computada por **R** deve-se a quatro fatos: **D** é uma cláusula-objetivo; **R** é linear de entrada; **R** dispensa fatoração; e **R** utiliza apenas uma vez a cláusula **D**. Note ainda que, em contraste com a discussão da seção 8.4 para Programação em Cláusulas, não é necessário alterar o sistema formal para computar respostas.

O resto desta seção pode ser ignorado em uma primeira leitura pois apresenta provas diretas, ou seja, independentes dos resultados do Capítulo 8, de que a computação de respostas por resolução-LSD é correta e, em certo sentido, completa. O desenvolvimento segue Lloyd [1984].

A definição seguinte apresenta de forma precisa o conceito de resposta computada por resolução-LSD, já considerando apenas respostas simples, em vista do Teorema 9.1.

Definição 9.6:

Seja **C** um programa em cláusulas definidas tal que duas cláusulas em **C** não possuem variáveis em comum. Seja **Q** uma consulta a **C** tal que **Q** é da forma " $Q_1 \wedge \dots \wedge Q_m$ ", onde Q_i são fórmulas atômicas. Seja **D** a cláusula-objetivo " $\leftarrow Q_1 \dots Q_m$ ". Uma resposta simples β de **Q** a **C** é *computada* pelo método de resolução-LSD(f) se e somente se existe uma LSD(f)-refutação **R** a partir de **C** $\cup \{D\}$ tal que β é a restrição da composição $\theta_1^0 \dots \theta_i$ às variáveis de **Q**, onde $\theta_1, \dots, \theta_i$ é a seqüência de unificadores usados para obter as cláusulas derivadas em **R**.

A ressalva de que duas cláusulas em **C** não possuem variáveis em comum apenas simplifica a definição, pois de outra forma as renomeações necessárias teriam que ser incluídas na composição das substituições que produz a resposta computada.

A prova da correção e completude da computação de respostas por resolução-LSD assume inicialmente que a função de seleção é a função padrão.

Teorema 9.3:

Seja f a função de seleção padrão. Se β é uma resposta de uma consulta **Q** a um programa em cláusulas definidas **C** computada pela resolução-LSD(f), então β é uma resposta correta.

Demonstração

Podemos supor sem perda de generalidade que quaisquer duas cláusulas em **C** não tem variáveis em comum. Provaremos o resultado por indução sobre o número de cláusulas derivadas das LSD(f)-refutações.

Base: Suponha que β seja uma resposta simples de **Q** a **C** computada por uma LSD(f)-refutação **R**. Suponha que há apenas uma cláusula derivada em **R**. Isto significa que **Q** é uma fórmula atômica, ou seja, a cláusula inicial de **R** é da forma $\leftarrow Q$ e que **C** possui uma cláusula unitária $L \leftarrow$ tal que $Q\beta = L\beta$. Então, $Q\beta$ é uma instância de uma cláusula em **C**. Logo, **C** implica logicamente $\forall Q\beta$ e, portanto, β é uma resposta correta.

Passo de Indução: Seja $i > 0$ e suponha que o resultado valha para LSD(f)-refutações com $i - 1$ cláusulas derivadas. Seja β uma resposta simples de Q a C computada por uma LSD(f)-refutação R e suponha que R tenha i cláusulas derivadas. Sejam $\theta_1, \dots, \theta_i$ os unificadores utilizados em R . Logo, $\beta = \theta_1^0 \dots \theta_i^0$. Mostraremos que C implica logicamente $VQ\beta$.

Suponha que Q seja da forma " $Q_1 \wedge \dots \wedge Q_m$ ". Seja D a cláusula " $\leftarrow Q_1 \dots Q_m$ ". Então, a primeira cláusula derivada de R é obtida estendendo-se D por alguma cláusula C em C . Suponha que C seja da forma " $L \leftarrow M_1 \dots M_q$ ". Logo, $(L)\theta_1 = (Q_1)\theta_1$ e o resultado é a cláusula D' da forma " $\leftarrow (M_1 \dots M_q \wedge Q_2 \dots Q_m)\theta_1$ ". Seja $C' = C \cup \{D'\}$. Então, há uma LSD(f)-refutação R' a partir de C' tal que R' possui $i - 1$ cláusulas derivadas e $\theta_2, \dots, \theta_i$ é a seqüência de u.m.g's usada. Seja Q' a consulta " $(M_1 \wedge \dots \wedge M_q \wedge Q_2 \wedge \dots \wedge Q_m)\theta_1$ ". Logo, β' , a restrição de $\theta_2^0 \dots \theta_i^0$ às variáveis de Q' , é a resposta de Q' a C computada por R' . Pela hipótese de indução, temos então que C implica logicamente " $\forall(M_1 \wedge \dots \wedge M_q \wedge Q_2 \wedge \dots \wedge Q_m)\theta_1^0 \theta_2^0 \dots \theta_i^0$ ". Portanto, C implica logicamente " $\forall(M_1 \wedge \dots \wedge M_q)\theta_1^0 \theta_2^0 \dots \theta_i^0$ " e " $\forall(Q_2 \wedge \dots \wedge Q_m)\theta_1^0 \theta_2^0 \dots \theta_i^0$ ". Mas " $L \leftarrow M_1 \dots M_q$ " pertence a C . Logo, C implica logicamente " $\forall(L)\theta_1^0 \theta_2^0 \dots \theta_i^0$ " e, portanto, " $\forall(Q_1)\theta_1^0 \theta_2^0 \dots \theta_i^0$ ". Finalmente, temos então que C implica logicamente " $\forall(Q_1 \wedge \dots \wedge Q_m)\theta_1^0 \theta_2^0 \dots \theta_i^0$ ".

A prova da completude da computação de respostas pelo método de resolução-LSD depende dos seguintes lemas auxiliares:

Lema 9.1:

Seja f a função de seleção padrão. Seja C um programa e Q uma consulta a C . Seja β uma resposta correta de Q a C . Então, a substituição-identidade é uma resposta computada de $Q\beta$ a C pela resolução-LSD(f).

Demonstração

Suponha que Q seja da forma " $Q_1 \wedge \dots \wedge Q_m$ ". Seja D a cláusula " $\leftarrow Q_1 \dots Q_m$ ". Seja $\varphi = \{x_1/a_1, \dots, x_k/a_k\}$ uma substituição das variáveis livres de $Q\beta$ por constantes distintas que não ocorrem em C ou Q . Logo, $D\beta\varphi$ é uma representação clausal de $\neg VQ\beta$. Como β é uma resposta correta de Q a C , temos que C implica logicamente $VQ\beta$. Logo, $C' = C \cup \{D\beta\varphi\}$ é insatisfatível. Mas C' é um conjunto quase-definido. Logo, há uma LSD(f)-refutação R a partir de C' . Mas as constantes

escolhidas em φ não ocorrem em \mathbf{C} ou \mathbf{Q} . Portanto, podemos substituir textualmente em \mathbf{R}' cada ocorrência de a_i por x_i e obter uma nova refutação \mathbf{R}' a partir de $\mathbf{C} \cup \{\mathbf{D}\beta\}$. Como x_1, \dots, x_k são as variáveis de $\mathbf{Q}\beta$ e não são afetadas por nenhuma substituição em \mathbf{R}' , a substituição-identidade é a resposta de $\mathbf{Q}\beta$ a \mathbf{C} computada por \mathbf{R}' .

Lema 9.2:

Seja f a função de seleção padrão. Seja \mathbf{C} um conjunto de cláusulas definidas, \mathbf{D} uma cláusula-objetivo, e θ uma substituição sobre as variáveis de \mathbf{D} . Suponha que duas cláusulas em $\mathbf{C} \cup \{\mathbf{D}\}$ não possuam variáveis em comum. Suponha que existe uma $\text{LSD}(f)$ -refutação \mathbf{R} a partir de $\mathbf{C} \cup \{\mathbf{D}\theta\}$. Então existe uma $\text{LSD}(f)$ -refutação \mathbf{R}' a partir de $\mathbf{C} \cup \{\mathbf{D}\}$ tal que \mathbf{R} e \mathbf{R}' tem o mesmo comprimento e, se $\theta_1, \dots, \theta_i$ e $\theta'_1, \dots, \theta'_i$, são as seqüências de unificadores usados em \mathbf{R} e \mathbf{R}' , então existe uma substituição φ tal que $\theta^0\theta_1^0\dots\theta_i^0 = \theta'_1^0\dots\theta_i^0\varphi$.

Demonstração

Provaremos por indução sobre o número de cláusulas derivadas das refutações.

Base: Suponha que \mathbf{R} seja uma refutação sem nenhuma cláusula derivada. Neste caso, \mathbf{D} tem que ser a cláusula vazia e o resultado segue trivialmente.

Passo de Indução: Seja $i > 0$ e suponha que o resultado valha para refutações com $i - 1$ cláusulas derivadas. Seja \mathbf{R} uma refutação com i cláusulas derivadas e suponha que as cláusulas derivadas em \mathbf{R} sejam $\mathbf{D}_1, \dots, \mathbf{D}_i$ e que os u.m.g's utilizados sejam $\theta_1, \dots, \theta_i$. Então, como $\mathbf{D}\theta$ é a cláusula inicial de \mathbf{R} , \mathbf{D}_1 é obtida estendendo-se $\mathbf{D}\theta$ por alguma cláusula \mathbf{C} em \mathbf{C} , tendo θ_1 como u.m.g.

Sejam $\mathbf{L}\theta$ e \mathbf{M} os literais de $\mathbf{D}\theta$ e \mathbf{C} selecionados na derivação de \mathbf{D}_1 . Logo, por escolha de θ_1 , temos que $(\mathbf{L}\theta)\theta_1 = \mathbf{M}\theta_1$. Por suposição, \mathbf{D} e \mathbf{C} não tem variáveis em comum e θ afeta apenas as variáveis de \mathbf{D} . Portanto, θ não afeta as variáveis de \mathbf{C} , o que implica em $\mathbf{M} = \mathbf{M}\theta$. Logo,

$$(1) \mathbf{L}(\theta^0\theta_1) = (\mathbf{L}\theta)\theta_1 = (\mathbf{M}\theta)\theta_1 = (\mathbf{M}\theta)\theta_1 = \mathbf{M}(\theta^0\theta_1)$$

Ou seja, \mathbf{L} e \mathbf{M} são unificáveis por $\theta^0\theta_1$. Seja β um u.m.g de \mathbf{L} e \mathbf{M} . Então, existe uma substituição β' tal que

$$(2) \beta^o \beta' = \theta^o \theta_1.$$

Seja D_1' o resultado de estender D por C usando β como u.m.g. Então, $(D_1')\beta' = D_1$. Portanto, existe uma refutação S a partir de $C \cup \{(D_1')\beta'\}$, tendo $(D_1')\beta'$ como cláusula inicial, tal que S possui $i - 1$ cláusulas derivadas. Além disto, as cláusulas derivadas de S são exatamente D_2, \dots, D_i e os u.m.g.'s usados são $\theta_2, \dots, \theta_i$.

Logo, pela hipótese de indução, existe uma refutação S' a partir de $C \cup \{D_1'\}$, tendo D_1' como cláusula inicial, tal que S' possui $i - 1$ cláusulas derivadas e, se os u.m.g.'s usados são $\theta'_2, \dots, \theta'_i$, então existe uma substituição φ tal que

$$(3) \beta'^o \theta_2^o \dots^o \theta_i^o = \theta'_2'^o \dots^o \theta_i'^o \varphi$$

Mas D_1' é uma extensão de D por C com u.m.g. β . Logo, existe uma refutação R' a partir de $C \cup \{D\}$ tal que R' tem i cláusulas derivadas e seqüência de u.m.g.'s $\theta'_1, \dots, \theta'_n$, onde

$$(4) \theta'_1 = \beta$$

Então, por (2), (3) e (4), temos que:

$$(5) \theta^o \theta_1^o \theta_2^o \dots^o \theta_i^o = \beta^o \beta'^o \theta_2^o \dots^o \theta_i^o = \beta^o \theta_2'^o \dots^o \theta_i'^o \varphi = \theta'_1'^o \theta_2'^o \dots^o \theta_i'^o \varphi$$

Portanto, R' satisfaz às condições desejadas.

Teorema 9.4:

Seja f a função de seleção padrão. Seja C um programa e Q uma consulta a C . Se β é uma resposta simples correta de Q a C então existe uma resposta θ de Q a C tal que θ é computada por resolução-LSD(f) e $\beta = \theta^o \varphi$, para alguma substituição φ .

Demonstração

Suponha que Q seja da forma " $Q_1 \wedge \dots \wedge Q_m$ ". Seja D a cláusula " $\leftarrow Q_1 \dots Q_m$ ". Logo, pelo Lema 9.1, existe uma refutação R a partir de $C \cup \{D\}$ tal que a resposta de $Q\beta$ a C computada por R é a substituição-identidade, " ε ". Ou seja, se $\theta_1, \dots, \theta_i$ é a seqüência de u.m.g.'s utilizada em R , então $\theta_1^o \dots^o \theta_i^o = \varepsilon$. Pelo Lema 9.2, existe então uma refutação R' a partir de $C \cup \{D\}$ tal que, se $\theta'_1, \dots, \theta'_i$ é a seqüência de

u.m.g.'s utilizada em \mathbf{R}' , então, para alguma substituição φ , temos que $\beta^o\theta_1^o \dots \theta_i^o = \theta_1^o \dots \theta_i^o\varphi$. Tome $\theta = \theta_1^o \dots \theta_i^o$. Então, θ é a resposta computada por \mathbf{R}' e $\beta = \beta^o\epsilon = \beta^o\theta_1^o \dots \theta_i^o = \theta_1^o \dots \theta_i^o\varphi = \theta^o\varphi$

Finalmente, os resultados gerais para qualquer função de seleção seguem como corolários dos teoremas anteriores e do Lema 7.2.

Corolário 9.1:

Seja f uma função de seleção. Se β é uma resposta de uma consulta \mathbf{Q} a um programa em cláusulas definidas \mathbf{C} computada pelo método de resolução-LSD(f), então β é uma resposta correta.

Corolário 9.2:

Seja f uma função de seleção. Seja \mathbf{C} um programa e \mathbf{Q} uma consulta a \mathbf{C} . Se β é uma resposta simples correta de \mathbf{Q} a \mathbf{C} então existe uma resposta computada θ de \mathbf{Q} a \mathbf{C} pelo método de resolução-LSD(f) tal que $\beta = \theta^o\varphi$, para alguma substituição φ .

9.5 EXTENSÃO PARA CLÁUSULAS PSEUDO-DEFINIDAS

Esta seção explora uma variante de Programação em Lógica, chamada Programação em Cláusulas Pseudo-Definidas, que estende o poder de expressão de Programação em Cláusulas Definidas.

Recorde da Seção 7.5 que uma cláusula pseudo-definida é uma expressão da forma " $L \leftarrow M_1 \dots M_n$ " ou da forma " $L \leftarrow$ " e que uma cláusula pseudo-objetivo é uma expressão da forma " $\leftarrow M_1 \dots M_n$ ", onde L é um literal positivo e M_1, \dots, M_n são literais positivos ou negativos.

Os principais conceitos sintáticos de Programação em Cláusulas Pseudo-Definidas são definidos da seguinte forma.

Definição 9.7:

- Um *programa em cláusulas pseudo-definidas* é um conjunto finito de cláusulas pseudo-definidas.
- Uma *consulta* a um programa em cláusulas pseudo-definidas \mathbf{C} é uma conjunção de literais sobre o alfabeto de \mathbf{C} .

Programação em Cláusulas Pseudo-Definidas é menos geral do que Programação em Cláusulas pois não permite a inclusão nos programas de cláusulas consistindo apenas de literais negados e não permite a definição de consultas através de fórmulas arbitrárias. Por outro lado, esta variante é mais geral do que Programação em Cláusulas Definidas, pois admite a ocorrência de cláusulas com mais de um literal positivo nos programas e a definição de consultas através de conjunções de literais, e não apenas de literais positivos.

A semântica declarativa e a noção de resposta correta seguem como em Programação em Cláusulas.

A semântica procedural induzida pela resolução-LSDNF segue de forma semelhante à seção 9.4. A resposta computada por uma LSDNF-refutação **R** é a composição das substituições feitas nas variáveis da consulta, ignorando as derivações originadas pela aplicação da regra da negação por falha finita, pois estas não geram substituições.

Exemplo 9.5:

Seja *f* a função de seleção padrão. Seja **C** o seguinte programa em cláusulas pseudo-definidas:

1. *chama(a,b)* ←
2. *usa(b,e)* ←
3. *depende(x,y)* ← *chama(x,y)*
4. *depende(x,y)* ← *usa(x,y)*
5. *depende(x,y)* ← *chama(x,z)* *depende(z,y)*

Seja *Q* a seguinte consulta:

$$\text{depende}(a,w) \wedge \neg \text{depende}(w,f)$$

A representação clausal de $\neg Q$ será então:

6. ← *depende(a,w)* $\neg \text{depende}(w,f)$

A seguinte seqüência de cláusulas é uma LSDNF(*f*)-refutação (cada linha inclui também os unificadores usados na derivação):

1. $chama(a,b) \leftarrow$
2. $usa(b,e) \leftarrow$
3. $depende(x,y) \leftarrow chama(x,y)$
4. $depende(x,y) \leftarrow usa(x,y)$
5. $depende(x,y) \leftarrow chama(x,z) depende(z,y)$
6. $\leftarrow depende(a,w) \neg depende(w,f)$
7. $\leftarrow \neg chama(a,w) \neg depende(w,f)$. 3, EXS, {x/a, y/w}
8. $\leftarrow \neg depende(b,f)$. 1, EXS, {w/b}
9. \square . NFF

A substituição $\theta = \{w/b\}$, obtida pela composição dos dois unificadores usados na refutação e restrita às variáveis da consulta, será então a resposta computada pela refutação.

A definição seguinte apresenta de forma precisa o conceito de resposta computada pela resolução-LSDNF:

Definição 9.8:

Seja **C** um programa em cláusulas pseudo-definidas tal que duas cláusulas em **C** não possuem variáveis em comum. Seja **Q** uma consulta a **C** tal que **Q** é da forma " $Q_1 \wedge \dots \wedge Q_m$ ", onde Q_i são literais. Seja **D** a cláusula pseudo-objetivo " $\leftarrow Q_1 \dots Q_m$ ". Uma resposta simples β de **Q** a **C** é *computada* pelo método de resolução-LSDNF(*f*) se e somente se existe uma LSDNF(*f*)-refutação **R** a partir de **C** U {**D**} tal que β é a restrição da composição $\theta_1 \circ \dots \circ \theta_i$ às variáveis de **Q**, onde $\theta_1, \dots, \theta_i$ é a seqüência de unificadores usados nas aplicações da regra de f-extensão em **R**.

Para esta noção de resposta computada é possível provar o seguinte resultado:

Teorema 9.5:

Seja *f* a função de seleção padrão. Se β é uma resposta de uma consulta **Q** a um programa em cláusulas pseudo-definidas **C** computada pelo método de resolução-LSDNF(*f*), então β é uma resposta correta.

(para uma demonstração, veja Lloyd [1984]).

Terminaremos esta sub-seção com uma discussão adicional sobre a importância da Hipótese do Mundo Fechado em Programação em Lógica, que justifica a adoção da resolução-LSDNF para computar respostas.

Recorde que a correção da resolução-LSDNF era aferida com relação ao fecho dos conjuntos de cláusulas pseudo-definidas, medida justificada com base na Hipótese do Mundo Fechado. Isto significa em termos simples que basta incluir em um programa "metade das definições", conforme ilustrado pelo seguinte exemplo:

Exemplo 9.6:

Considere a seguinte definição de "tio", a partir de "pai", "mãe" e "irmão", em Lógica de Primeira Ordem:

$$(1) \forall x \forall y (\text{tio}(x,y) \equiv \exists z ((\text{pai}(z,y) \vee \text{mãe}(z,y)) \wedge \text{irmão}(z,x)))$$

Esta definição pode ser desdobrada em duas:

$$(2) \forall x \forall y (\exists z ((\text{pai}(z,y) \vee \text{mãe}(z,y)) \wedge \text{irmão}(z,x)) \rightarrow \text{tio}(x,y))$$

que permite provar fatos da forma " $\text{tio}(t,u)$ " a partir de fatos da forma " $\text{pai}(v,u)$ ", " $\text{mãe}(v,u)$ " e " $\text{irmão}(v,t)$ ", e

$$(3) \forall x \forall y (\neg \exists z ((\text{pai}(z,y) \vee \text{mãe}(z,y)) \wedge \text{irmão}(z,x)) \rightarrow \neg \text{tio}(x,y))$$

que permite provar fatos da forma " $\neg \text{tio}(x,y)$ " a partir de fatos da forma " $\neg \text{pai}(v,u)$ ", " $\neg \text{mãe}(v,u)$ " e " $\neg \text{irmão}(v,t)$ ".

Considere a representação clausal de (1) expressa através de cláusulas pseudo-definidas:

- c₁. $\text{tio}(x,y) \leftarrow \text{pai}(z,y) \text{ irmão}(z,x)$
- c₂. $\text{tio}(x,y) \leftarrow \text{mãe}(z,y) \text{ irmão}(z,x)$
- c₃. $\text{pai}(f(x,y),y) \leftarrow \text{tio}(x,y) \neg \text{mãe}(f(x,y),y)$
- c₄. $\text{irmão}(f(x,y),x) \leftarrow \text{tio}(x,y)$

Assim, um programa **C** deveria conter em princípio todas estas quatro cláusulas para exprimir exatamente (1). Porém, em presença da Hipótese do Mundo Fechado, formalizada pela noção de fecho, basta que **C** contenha apenas as cláusulas c₁ e c₂ para exprimir (1) (supondo que **C** não contenha nenhuma outra cláusula cujo primeiro literal seja

sobre o símbolo predicativo "tio"). De fato, a definição completa de "tio" em $\mathbf{C} = \{c_1, c_2\}$

$$(4) \forall x \forall y (\text{tio}(x,y) \equiv (\exists z(\text{pai}(z,y) \& \text{irmão}(z,x)) \vee \exists z(\text{mãe}(z,y) \& \text{irmão}(z,x)))$$

é equivalente à definição original contida em (1).

Finalmente, note que c_1 e c_2 representam (2), ou seja, "metade da definição" de "tio". Portanto, a Hipótese do Mundo Fechado, formalizada através da noção de fecho de um conjunto de cláusulas pseudo-definidas, permite simplificar consideravelmente a construção de programas em cláusulas.

9.6 MODELAGEM DE ALGORITMOS

Esta seção discute a modelagem de algoritmos em Programação em Lógica e deve ser considerada como um apêndice ao capítulo.

A modelagem de algoritmos em Programação em Lógica reflete a tese, exposta em Kowalski [1979], de que a descrição de um algoritmo pode ser desdobrada em uma *componente lógica*, que especifica a semântica do algoritmo, e em uma *componente de controle*, que descreve como as definições da componente lógica deverão ser usadas. A componente lógica especifica o conhecimento a ser usado na resolução do problema, ou seja, *o que* deve ser feito, e a componente de controle determina a estratégia que guiará o uso do conhecimento, ou seja, *como* deve ser feito. Portanto, a componente de controle afeta apenas a eficiência, mas não o significado de um algoritmo. Já a componente lógica afeta também o comportamento do algoritmo, além de fixar o seu significado.

Os exemplos a seguir sugerem como especificar a componente lógica de um algoritmo através de um conjunto \mathbf{C} de cláusulas definidas, interpretadas como definições mutuamente recursivas dos conjuntos de literais do H-modelo mínimo de \mathbf{C} . O resultado final é efetivamente um programa em cláusulas definidas descrevendo o algoritmo desejado, segundo a semântica do modelo mínimo.

Exemplo 9.7:

O seguinte programa em cláusulas definidas define a componente lógica do algoritmo usual de fatoração:

Programa FATORAÇÃO

```
fatorial(0,1) ←
fatorial(x + 1,u) ← fatorial(x,v) prod(x + 1,v,u)
```

onde, intuitivamente:

- "*fatorial(x,y)*" indica que *y* é o fatorial de *x*
- "*prod(x,y,z)*" indica que *z* é o produto de *x* por *y*

A definição de "*prod*" foi omitida por ser em geral fornecida como parte do sistema para Programação em Lógica (ver seção 11.2).

A componente de controle mais natural para este algoritmo corresponde à implementação recursiva usual de fatorial, que reduz o problema de computar o fatorial de $n+1$ aos problemas de computar o fatorial de n e de computar o produto deste valor por $n+1$.

Programação em Lógica, na verdade, reduz a definição de um algoritmo apenas à definição da sua componente lógica já que a estratégia genérica do procedimento de refutação determina completamente a componente de controle. Programação em Lógica permite, desta forma, desenvolver a especificação de um algoritmo de forma completamente independente do seu comportamento computacional.

De um ponto de vista mais pragmático, os procedimentos de refutação em geral permitem variar a componente de controle através da especificação de informação extra-lógica para guiar a busca de uma refutação. Portanto, para descrever um algoritmo, o programador deve criar várias descrições da componente lógica e considerar como cada uma delas se comporta perante as várias opções de controle oferecidas pelo procedimento de refutação adotado. A escolha final da componente lógica deverá recair sobre aquela que maximiza a clareza da especificação e que é executada com eficiência aceitável pelo procedimento de refutação para as opções de controle especificadas.

O exemplo abaixo ilustra de forma clara o efeito de componentes lógicas distintas para o mesmo algoritmo.

Exemplo 9.8:

Considere os seguintes programas em cláusulas para ordenação:

Programa ORDENAÇÃO_POR_PERMUTAÇÃO

```

ordena(x,y) ← permuta(x,y) ordenada(y)
ordenada(nil) ←
ordenada(x.nil) ←
ordenada(x.y.z) ← menor-ou-igual(x,y) ordenada(y.z)

permuta(nil,nil) ←
permuta(x,u.v) ← remove(u,x,z) permuta(z,v)

remove(x,x.y,y) ←
remove(x,y.z,y.w) ← remove(x,z,w)

```

onde, intuitivamente

- “*ordena(x,y)*” indica que a lista y é uma ordenação (crescente) da lista x
- “*ordenada(x)*” indica que a lista x está ordenada (crescentemente)
- “*permuta(x,y)*” indica que a lista y é uma permutação da lista x
- “*menor-ou-igual(x,y)*” indica que x é menor ou igual a y
- “*remove(x,y,z)*” indica que z é a lista obtida removendo-se x da lista y

Programa ORDENAÇÃO_RÁPIDA

```

ordena(nil,nil) ←
ordena(x,y,z) ← particiona(x,y,u,v)
    ordena(u,u1)
    ordena(v,v1)
    concat(u1,x,v1,z)

concat(nil,x,x) ←
concat(u,x,y,u,z) ← concat(x,y,z)

particiona(x,nil,nil,nil) ←
particiona(x,z,y,z,u,v) ← menor-ou-igual(z,x) particiona(x,y,u,v)
particiona(x,z,y,u,z,v) ← maior(z,x) particiona(x,y,u,v)

```

onde, intuitivamente

- "*ordena(x,y)*" indica que a lista *y* é uma ordenação (crescente) da lista *x*
- "*particiona(x,y,u,v)*" indica que *u* é a lista dos elementos da lista *y* menores ou iguais a *x* e *v* é a lista dos elementos de *y* maiores do que *x*.
- "*concat(x,y,z)*" indica que a lista *z* é concatenação da lista *x* com a lista *y*
- "*menor-ou-igual(x,y)*" indica que *x* é menor ou igual a *y*
- "*maior(x,y)*" indica que *x* é maior do que *y*

Nota: As definições de "*menor-ou-igual*" e "*maior*" foram omitidas por serem dependentes do tipo de dados dos átomos das listas.

Para a maioria dos procedimentos de refutação, mesmo com opções de controle sofisticadas, o segundo programa se comportará de forma muito mais eficiente do que o primeiro. De fato, o segundo programa define a lógica do algoritmo QUICKSORT (veja, por exemplo, Horowitz e Sahni [1976]) enquanto que o primeiro exige gerar todas as permutações da lista de entrada até encontrar uma que seja a ordenação desta lista, o que pode levar a uma explosão combinatorial.

NOTAS BIBLIOGRÁFICAS

Lloyd [1984] é um excelente tutorial sobre Programação em Cláusulas Definidas, reunindo de forma coerente resultados de várias fontes. As seções 9.2 a 9.5 contém os resultados mais importantes desta referência, reinterpretados em termos de conceitos de capítulos anteriores. A discussão sobre as componentes de algoritmos, que forma o material da seção 9.6, foi tirada diretamente de Kowalski [1979b]. Há inúmeras sugestões para extensões de Programação em Cláusulas Definidas propostas na literatura, algumas das quais são discutidas em Kowalski [1978,1979a,1981,1983]. Em particular, a hipótese do mundo fechado e a negação por falha são tratada em Clark [1978], Lloyd [1984] e Reiter [1978,1984], e a inclusão de conceitos meta-lingüísticos em Bowen e Kowalski [1982].

PARTE IV - A LINGUAGEM PROLOG

CAPÍTULO 10: A LINGUAGEM PROLOG BÁSICA

Este capítulo apresenta a linguagem Prolog como uma implementação de Programação em Cláusulas Definidas, descrevendo aspectos relativos a sintaxe, semântica e processamento de listas. A versão de Prolog descrita baseia-se na sintaxe do IBM Prolog (VM/Prolog (IBM [1985]) ou MVS/Prolog (IBM [1986])), semelhante à do Waterloo Prolog (Roberts [1977]). Com pequenas alterações de sintaxe, as codificações apresentadas podem ser convertidas para outros dialetos Prolog.

As seções recomendadas para cada nível de leitura são:

nível introdutório: 10.2.1, 10.3.2 (em parte), 10.4

nível intermediário: 10.2.1, 10.3.2, 10.4, 10.5

10.1 INTRODUÇÃO

A idéia de usar Lógica como um formalismo executável em computador, explorada principalmente por Kowalski ([1972] e [1974]) e Hayes [1973], recebeu um grande ímpeto com o advento da linguagem PROLOG (PROgramming in LOGic), que deve ser entendida como uma implementação das idéias de Programação em Lógica para o subconjunto das cláusulas definidas. Esta linguagem foi projetada e implementada por

Colmerauer e seu Grupo de Inteligência Artificial (GIA), na Universidade de Marseille, onde foi escrito o primeiro interpretador Prolog na linguagem ALGOL-W (Colmerauer et alli [1973]). Posteriormente, Battani e Méloni [1973] implementaram uma versão mais eficiente deste interpretador, escrita parcialmente em FORTRAN e Roberts [1977] implementou na Universidade de Waterloo, uma versão, conhecida como Waterloo Prolog, escrita totalmente em linguagem de máquina. Mas a linguagem Prolog só passou a atrair um amplo interesse a nível mundial, quando Pereira, Pereira e Warren implementaram, na Universidade de Edinburgh, uma versão muito eficiente conhecida como DEC-10 Prolog ou Edinburgh Prolog (Pereira, Pereira e Warren [1978], Warren [1979]), que incluiu o primeiro compilador Prolog escrito em Prolog. A seguir, o anúncio efetuado pelo Japão do projeto do computador de quinta geração (Moto-Oka [1982]) focalizou uma grande atenção sobre Prolog por escolher esta linguagem (ou alguma atualização futura da mesma) como o núcleo de programação deste empreendimento, cuja realização está prevista para a década de 90. O interesse crescente por Prolog em áreas de Inteligência Artificial é outro evento que muito tem contribuído para o incentivo de pesquisa e desenvolvimento nesta linguagem e, de um modo mais amplo, em Programação em Lógica.

A linguagem Prolog tem sido utilizada para desenvolver aplicações nas áreas de:

- engenharia de software
- banco de dados relacionais
- manipulação simbólica de fórmulas matemáticas
- prova automática de teoremas
- construção de compiladores
- projetos apoiados por computador (PAC)
- inteligência artificial, nas sub-áreas de:
 - engenharia de conhecimento
 - processamento de linguagem natural
 - robótica

Prolog é uma linguagem interativa que permite resolver problemas que envolvem representação simbólica de objetos e seus relacionamentos. O advento da linguagem Prolog reforçou a tese de que a Lógica é um formalismo conveniente para representar e processar conhecimento. Prolog evita que o programador descreva procedimentos para obter a

solução de um problema, permitindo que ele expresse declarativamente apenas a estrutura lógica do problema através de termos, fórmulas atômicas e cláusulas.

Um programa Prolog possui três interpretações semânticas básicas. Na interpretação declarativa entende-se que as cláusulas que definem o programa descrevem uma teoria de primeira ordem. Na interpretação procedural as cláusulas são vistas como entrada para um método de refutação. Finalmente, na interpretação operacional as cláusulas são vistas como comandos para um particular procedimento de refutação.

Estas alternativas para a semântica são valiosas em termos de entendimento e codificação de um programa Prolog. A interpretação declarativa permite que o programador modele um dado problema através de assertivas acerca dos objetos do domínio de discurso, simplificando a tarefa de programar em Prolog, em comparação com linguagens tipicamente procedimentais, como a linguagem Pascal. A interpretação procedural permite que o programador identifique e descreva o problema pela redução do mesmo a subproblemas, através da definição de uma série de chamadas de procedimentos. Por fim, a semântica operacional reintroduz a idéia de controle de execução, que é irrelevante do ponto de vista da semântica declarativa, através da ordem das cláusulas em um programa Prolog e das fórmulas atômicas em uma cláusula Prolog. Ou seja, esta interpretação é semelhante à semântica operacional de muitas linguagens de programação tradicionais e deve ser considerada, principalmente, em grandes programas, por questões de eficiência de execução. É interessante notar que o programador pode comutar de uma interpretação para outra, em busca de um efeito sinérgico, para facilitar codificação na linguagem Prolog.

Em resumo, as principais características da linguagem Prolog são:

- orientada para processamento simbólico;
- representa uma implementação da Lógica como linguagem de programação;
- apresenta uma semântica declarativa inerente à Lógica;
- permite a definição de programas invertíveis, ou seja, programas que não distinguem entre argumentos de entrada e de saída. Como consequência, permite a definição de programas com mais de uma finalidade, que podem ser chamados com formas diferentes de entrada/saída;

- permite a obtenção de respostas alternativas;
- incorpora um mecanismo uniforme para passagem, análise, seleção e criação de estruturas de dados;
- suporta estrutura de dados que permite simular registros ou listas;
- permite recuperação dedutiva de informação;
- suporta codificações recursivas para descrição de processos e problemas dispensando os mecanismos tradicionais de controle, como o comando "goto" e laços "do", "for" e "while";
- permite aproximar o processo de especificação do processo de codificação de programas;
- representa programas e dados através do mesmo formalismo (cláusulas);
- incorpora facilidades computacionais extra e metalógicas.

As limitações atuais desta linguagem ocorrem devido a:

- existência de diferentes implementações (dialetos) divergindo entre si pela sintaxe e pelas facilidades oferecidas;
- inexistência, em diversas implementações, de facilidades para comunicação com programas escritos em linguagens de programação tradicionais;
- inexistência de ambientes e facilidades de programação adequados que permitam suportar o desenvolvimento e a utilização de programas de grande porte (editores potentes, facilidades gráficas e facilidades de depuração que considerem as particularidades inerentes a uma linguagem baseada em Lógica);
- limitação do poder de expressão a cláusulas definidas (na linguagem básica);
- tratamento da negação lógica apenas através do conceito restrito de "Negação por Falha Finita".

10.2 SINTAXE DA LINGUAGEM PROLOG BÁSICA

Esta seção define a sintaxe da linguagem Prolog básica, que corresponde a uma implementação de Programação em Cláusulas Definidas, repetindo quando necessário, alguns conceitos referentes às linguagens de primeira ordem.

10.2.1 Sintaxe das Cláusulas Prolog

O alfabeto da linguagem Prolog básica é definido da seguinte forma:

Definição 10.1:

O alfabeto básico Prolog consiste dos seguintes símbolos:

pontuação: (,), . , ' , "

conectivos: & (conjunção)

<- (implicação)

letras: a, b, ..., z, A, B, ..., Z

dígitos: 0, 1, ..., 9

especiais: *, _, +, -, /, ...

Nota: O conjunto de símbolos especiais é deixado em aberto, exceto que, necessariamente, deverá conter o asterisco.

Definição 10.2:

(a) Um *átomo* é:

- (i) uma cadeia de letras e dígitos, iniciando com uma letra minúscula;
- (ii) uma cadeia de símbolos do alfabeto básico Prolog (podendo incluir o branco) delimitada por aspas.

(b) Uma *constante Prolog* é:

- (i) um átomo;
- (ii) uma cadeia de dígitos com no máximo uma ocorrência do símbolo ":";
- (iii) uma cadeia de símbolos do alfabeto básico Prolog (podendo incluir o branco) delimitada por apóstrofos.

(c) Uma *variável Prolog* é:

- (i) uma cadeia de letras e dígitos iniciando com uma letra maiúscula;
- (ii) o símbolo "*", chamado de *variável anônima*.

Nota: Em alguns dialetos Prolog uma variável é a variável anônima ou uma cadeia de letras e dígitos iniciando com um asterisco. Observe que,

se o nome de um átomo inicia com uma letra minúscula e não contém brancos ou símbolos especiais, a codificação de aspas é desnecessária. Mas o uso do símbolo especial "_" no nome de um átomo não implica na necessidade de uso de aspas.

Exemplo 10.1:

(a) Átomos:

a, d, fortran, sequencial, 1e0A, permitido, depende, chama
 "a b", "Joao Silva", "Pedro I", "<-"

(b) Constantes Prolog:

a, d, fortran, sequencial, 1e0A, permitido, depende, chama
 "a b", "Joao Silva", "Pedro I", "<-"
 15, 123, 15.5, 123.32
 'a b', 'Joao Silva', 'Pedro I', '<-'

(c) Variáveis Prolog:

X, Y, Z, W, Ma, Pa, X523x, Xyz, Nome, RESPOSTA, A11, *

Como nos capítulos anteriores, usaremos aspas para destacar a ocorrência de um átomo, constante ou variável Prolog dentro de texto corrido. Porém, as aspas utilizadas desta forma e aquelas do alfabeto Prolog serão tipograficamente diferentes.

O conjunto dos termos Prolog é definido de forma idêntica aos termos de primeira ordem:

Definição 10.3:

O conjunto dos *termos Prolog*, ou simplesmente dos *termos*, é o menor conjunto satisfazendo às seguintes condições:

- (i) toda variável Prolog é um termo Prolog;
- (ii) toda constante Prolog é um termo Prolog;

- (iii) se t_1, \dots, t_n são termos Prolog e f é um átomo, então $f(t_1, \dots, t_n)$ é um termo Prolog, onde o átomo f tem o papel de um símbolo funcional n -ário. Diz-se ainda que a expressão $f(t_1, \dots, t_n)$ é um termo funcional Prolog.

Definição 10.4:

Uma *fórmula atômica Prolog* é uma cadeia da forma $p(t_1, \dots, t_n)$, para $n \geq 0$, onde t_1, \dots, t_n são termos Prolog e p é um átomo no papel de um símbolo predicativo n -ário.

Quando n for igual a zero, por abuso de linguagem, escreveremos $p()$ como p .

Note que 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 o mesmo representa.

Exemplo 10.2:

(a) Termos Prolog:

```
x, a, "a b", fortran, 'Joao Silva', 15.5, X,  
livro(Autor,Editora,Ano)
```

(b) Fórmulas Atômicas Prolog:

```
x(a,"a b",fortran,'Joao Silva',15.5,X)  
livro(Autor,Editora,Ano)
```

Observe que uma mesma expressão pode ser um termo ou uma fórmula atômica, dependendo do contexto na qual a mesma se encontre inserida.

Definição 10.5:

O conjunto das *cláusulas Prolog*, ou simplesmente das *cláusulas*, é o menor conjunto satisfazendo às seguintes condições:

- (i) se A é uma fórmula atômica Prolog, então " $A.$ " é uma cláusula Prolog, chamada de *cláusula unitária Prolog* (ou *cláusula terminal Prolog*);

- (ii) se A e B_1, \dots, B_n são fórmulas atômicas Prolog, então a expressão " $A \leftarrow B_1 \ \& \dots \& \ B_n$." é uma cláusula Prolog, chamada de *cláusula não-unitária Prolog*, onde A é a *cabeça* e " $B_1 \ \& \dots \& \ B_n$ " é o *corpo* da cláusula;
- (iii) se C_1, \dots, C_n são fórmulas atômicas Prolog, então " $\leftarrow C_1 \ \& \dots \& \ C_n$." é uma cláusula Prolog, chamada de *cláusula objetivo Prolog*.

Definição 10.6:

Uma *cláusula definida Prolog* é uma cláusula unitária ou uma cláusula não-unitária Prolog.

Definição 10.7:

Uma cláusula definida C é *sobre* um átomo p , no papel de um símbolo predutivo n -ário, se e somente se a cabeça de C for da forma " $p(t_1, \dots, t_n)$ " ou C for uma cláusula unitária da forma " $p(t_1, \dots, t_n)$ ". Diremos que p é o átomo da cláusula definida.

Exemplo 10.3:

(a) Cláusula Unitária Prolog:

`programa(a, fortran).`

leia-se: "a é um programa fortran"

`arquivo(d, sequencial).`

leia-se: "d é um arquivo sequencial"

Note que a primeira cláusula é sobre o átomo "programa" e a segunda sobre o átomo "arquivo" ambos no papel de símbolos preditivos binários.

(b) Cláusula Não-Unitária Prolog:

```
depende(X,Y) :- chama(X,Z) & depende(Z,Y).
```

leia-se: "para todo X, Y e Z, X depende de Y se X chama Z e Z depende de Y"

Note que esta cláusula é sobre o átomo "depende" no papel de um símbolo predutivo binário. Note também que esta cláusula não unitária é recursiva, no sentido que a cabeça da cláusula é referenciada no corpo da mesma.

(c) Cláusula Objetivo Prolog:

```
<- depende(a,e).
```

leia-se: "a depende de e?"

```
<- depende(X,b) & programa(X,fortran).
```

leia-se: "existe algum X que depende de b e é um programa fortran?"

Definição 10.8:

- (a) Um *programa Prolog* é uma seqüência de cláusulas definidas Prolog.
- (b) Uma *consulta Prolog* é uma cláusula objetivo Prolog.

Permitiremos ainda a inclusão de comentários em um programa Prolog entre os delimitadores "/*" e "*/".

A ordem das fórmulas atómicas em uma cláusula Prolog, bem como a ordem das cláusulas em um programa Prolog, é importante. Assim, por exemplo, "A <- B₁ & B₂." e "A <- B₂ & B₁." são distintas cláusulas não-unitárias Prolog.

Convém enfatizar também que um mesmo átomo pode possuir, em um mesmo programa, diferentes papéis, identificados por diferentes aridades e contextos (i.e., diferentes ocorrências em termos ou fórmulas atómicas). O processo de unificação, naturalmente, distinguirá ocorrências do mesmo átomo em diferentes papéis.

Definição 10.9:

Dado um programa Prolog P , a subseqüência de cláusulas em P sobre um mesmo átomo p , no mesmo papel, é o *procedimento* p definido em P .

Evitamos usar o termo "predicado" neste texto pois, na literatura, este termo aparece no sentido ora de fórmula atômica, ora de símbolo predutivo ou mesmo de procedimento.

Exemplo 10.4: Um Dicionário de Programas e Arquivos

O programa apresentado a seguir, chamado PDIPA, corresponde à codificação em Prolog do dicionário de programas e arquivos formalizado nos exemplos 8.1 e 8.6. O programa utiliza os seguintes átomos e constantes:

Átomos no papel de símbolos preditivos binários:

programa, arquivo, chama, usa, depende

Constantes Prolog:

a, b, c, d, e, fortran, pascal, sequencial, direto

A interpretação pretendida para as fórmulas atômicas Prolog será a seguinte:

programa(X, Y) X é um programa escrito na linguagem Y

arquivo(X, Y) X é um arquivo com formato Y

chama(X, Y) X é um programa que chama o programa Y

usa(X, Y) X é um programa que usa o arquivo Y

depende(X, Y) X é um programa que depende do programa ou arquivo Y

```

/*
          Programa PDIPA
*/
c1: programa(a,fortran).
c2: programa(b,pascal).
c3: programa(c,fortran).
c4: arquivo(d,sequencial).
c5: arquivo(e,direto).
c6: chama(a,b).
c7: chama(a,c).
c8: usa(a,d).
c9: usa(b,e).
c10: depende(X,Y) <- chama(X,Y).
c11: depende(X,Y) <- usa(X,Y).
c12: depende(X,Y) <- chama(X,Z) & depende(Z,Y).

```

Observe que, no programa PDIPA as cláusulas Prolog estão precedidas do prefixo c_i ($i = 1, 2, \dots$). Este modo de referência será utilizado no texto sempre que houver necessidade de referenciar cláusulas Prolog em um programa. Estes prefixos, que, neste texto, constituem-se unicamente em uma facilidade para referências, são opcionalmente oferecidos em alguns dialetos Prolog, permitindo nomear cláusulas em um programa Prolog e fórmulas atômicas em uma cláusula Prolog. Trata-se, quando disponível, de uma facilidade extremamente útil, pois permite, por exemplo, distinguir cláusulas Prolog idênticas carregadas de arquivos distintos. Nos dialetos Prolog que dispõem desta facilidade, as fórmulas atômicas Prolog podem diferir, além de pelo átomo e aridade, também pelo prefixo.

Observe também que as cláusulas c_{10} , c_{11} e c_{12} formam o procedimento "depende" definido em PDIPA.

Exemplo 10.5:

As cláusulas Prolog c_i e c_j , apresentadas a seguir, apesar de serem sobre um mesmo átomo em papéis diferentes, podem ser cláusulas de um mesmo programa Prolog:

```

ci: depende(X,Y) <- usa(X,Y).
cj: depende(a,b,c).

```

leia-se (interpretação pretendida):

c_i : "para todo X e Y, X depende de Y se X usa Y"

c_j : "a depende de b e de c"

Evidentemente, esta flexibilidade da linguagem Prolog deve ser usada com restrições para evitar que a codificação de um programa Prolog torne-se confusa.

10.2.2 Sintaxe dos Operadores Prolog

A sintaxe da linguagem Prolog básica pode ser opcionalmente estendida através da declaração de *operadores Prolog* representados por átomos. Esta extensão melhora a legibilidade de programas Prolog, sem acrescentar poder extra, em termos de expressividade, à linguagem. Com o emprego desta facilidade sintática, termos ou fórmulas atómicas como " $r_1(t)$ " ou " $r_2(t_a, t_b)$ ", podem ser codificados como:

- $r_1 \ t$ (definindo-se r_1 como operador prefixo)
- $t \ r_1$ (definindo-se r_1 como operador sufixo)
- $t_a \ r_2 \ t_b$ (definindo-se r_2 como operador infixo)

Por exemplo, a cláusula unitária:

`chama(a,b).`

com a definição de "chama" como operador infixo, pode ser codificada como:

`a chama b.`

Analogamente, com o uso de operadores, expressões aritméticas podem ser representadas em uma forma usual, em substituição à notação adotada para termos. Por exemplo, o termo:

`*(X,-(Y,Z))`

declarando-se os operadores "*" e "-" em notação infixa, pode ser codificado como:

`X*(Y-Z)`

Informalmente, um operador Prolog é declarado codificando-se, antes de qualquer cláusula que contenha o mesmo, uma cláusula da forma:

`op(Nome, Tipo, Prior).`

onde:

Nome átomo identificando o operador.

Tipo símbolo predefinido indicando a posição e as regras de associatividade do operador, podendo ser:

prefix - indica notação prefixa

suffix - indica notação sufixa

lr - indica notação infixa com associatividade à esquerda

rl - indica notação infixa com associatividade à direita

Prior número inteiro especificando, em relação a outros operadores, a ordem de avaliação do operador em uma expressão. Operadores associados a números menores têm maior prioridade sobre operadores associados a números maiores.

De um modo geral, a ordem dos três argumentos de "op" e o modo de definir o tipo do operador dependem da particular implementação considerada e devem ser elucidados no manual do dialeto Prolog considerado. Neste manual, devem ser verificados também os operadores Prolog predefinidos.

Observe que "op" somente declara a existência de um operador Prolog, uma conveniência sintática, mas não define a sua semântica (o que o operador faz), apesar da palavra "operador" sugerir a ocorrência de uma operação ou ação. Esta definição, se necessária, deve ocorrer através da inclusão de cláusulas pertinentes no programa Prolog ou na área de trabalho Prolog (conceituada na seção 10.4).

Observe também que:

- operadores podem ser declarados para o papel de símbolos predicativos ou para o papel de símbolos funcionais;
- a prioridade e as opções "lr" e "rl" só fazem pleno sentido quando o operador ocorre no papel de um símbolo funcional. No caso de um

- símbolo predutivo, as opções "lr" ou "rl" significam apenas que o operador é infixo;
- um mesmo operador pode ser declarado mais de uma vez com diferentes tipos (*operador misto*) desde que as combinações sejam procedentes ("prefix" com "lr" ou "prefix" com "rl");
 - o tipo, através das regras de associatividade, e a prioridade dos operadores permitem resolver ambigüidades da forma " $t_a p_1 t_b p_2 t_c$ ", onde p_1 e p_2 são operadores Prolog declarados e t_a , t_b e t_c são termos Prolog. Mais especificamente:
 - o tipo, através das regras de associatividade, permite interpretar corretamente e resolver ambigüidades que surgem com o emprego sucessivo, sem utilização de parênteses, de operadores infixos iguais, como na expressão "1000/100/10";
 - a prioridade, permite interpretar corretamente e resolver ambigüidades que surgem com o emprego sucessivo, sem utilização de parênteses, de operadores desiguais, como na expressão "10*30+513".

Exemplo 10.6:

As cláusulas a seguir declaram e definem um operador infixo, denotado por "unifica", que permite verificar se duas estruturas arbitrárias são unificáveis

```
op(unifica,lr,50).      /* declaracao do operador */
X unifica X.            /* definicao do operador */
```

Nota: Este operador é predefinido nos dialetos Prolog existentes (através do símbolo "=", em geral).

Exemplo 10.7: Alguns operadores Prolog predefinidos

```
op("&",rl,10).
op("<-",rl,10).
op("<-",prefix,10).
op("|",rl,20).
op(",",rl,30).
op("¬",prefix,40).
op(".",rl,100).
```

Note que entre os operadores Prolog predefinidos estão o operador "&" (infixo) e o operador "<-" (infixo e prefixo).

Exemplo 10.8: Uma segunda forma do Dicionário

PDIPAA é o programa PDIPA alterado com a declaração dos seguintes operadores Prolog infixos: "op(eh_um_programa)", "op(eh_um_arquivo)" (em substituição aos símbolos predicativos "programa" e "arquivo", por questões de legibilidade), "chama", "usa" e "depende".

```
/*
Programa PDIPAA
*/
op(eh_um_programa,r1,60).
op(eh_um_arquivo,r1,60).
op(chama,r1,60).
op(usa,r1,60).
op(depende,r1,60).

a eh_um_programa fortran.
b eh_um_programa pascal.
c eh_um_programa fortran.
d eh_um_arquivo sequencial.
e eh_um_arquivo direto.
a chama b.
a chama c.
a usa d.
b usa e.

X depende Y <- X chama Y.
X depende Y <- X usa Y.
X depende Y <- X chama Z & Z depende Y.
```

10.3 SEMÂNTICA DE UM PROGRAMA PROLOG BÁSICO

10.3.1 Semânticas Declarativa e Procedimental

Um programa Prolog, com pequenas diferenças sintáticas, nada mais é do que um programa em cláusulas definidas. Portanto, todos os conceitos e resultados sobre as semânticas declarativa e procedural para programas

em cláusulas definidas, apresentados nas seções 9.2, 9.3 e 9.4, transferem-se para Prolog de forma quase imediata.

O único ponto importante consiste em explicitar o significado da variável anônima e do uso do mesmo átomo em papéis diferentes. Intuitivamente, estas facilidades sintáticas devem ser entendidas da seguinte forma:

- cada ocorrência da variável anônima representa a ocorrência de uma nova variável;
- um mesmo átomo em dois papéis diferentes representa símbolos funcionais ou predicativos diferentes.

O resto desta seção captura de forma mais precisa estas observações definindo duas transformações sintáticas.

Definição 10.10:

Sejam P e C um programa e uma consulta Prolog. A *regularização* de P e C são o programa R e a consulta D resultantes de aplicar as seguintes transformações a P e C :

- (i) Substitua cada ocorrência da variável anônima por uma nova variável;
- (ii) Se p é um átomo que ocorre em mais de um papel, substitua todas as ocorrências de p no mesmo papel por um novo átomo. Repita este passo até que todos os átomos ocorram em um único papel.

Note que R e D não são evidentemente únicos, mas é imediato mostrar que:

- R e D são realmente um programa e uma consulta Prolog;
- nenhuma variável anônima ocorre em R ou D ;
- nenhum átomo ocorre em R ou D em mais de um papel.

Exemplo 10.9:

Seja P o seguinte programa Prolog:

1. chama(a,b).
2. usa(b,e).
3. depende(X,Y) <- chama(X,Y).
4. depende(X,Y) <- usa(X,Y).
5. depende(X,Y) <- chama(X,Z) & depende(Z,Y).
6. depende(X,Y,Z) <- chama(X,Z) & depende(Z,Y).

e C a seguinte consulta Prolog:

7. <- depende(X,b,e).

A regularização de P e C resulta no programa R:

1. chama(a,b).
2. usa(b,e).
3. dep1(X,Y) <- chama(X,Y).
4. dep1(X,Y) <- usa(X,Y).
5. dep1(X,Y) <- chama(X,Z) & dep1(Z,Y).
6. dep2(X,Y,Z) <- chama(X,Z) & dep1(Z,Y).

e na consulta D:

7. <- dep2(X,b,e).

Note que R e D foram obtidos de P e C substituindo todas as ocorrências de "depende" no papel de um símbolo predicativo binário por "dep1" e todas as ocorrências de "depende" no papel de um símbolo predicativo ternário por "dep2" ("dep1" e "dep2" foram escolhidos arbitrariamente). Note ainda que esta segunda substituição afeta tanto a cláusula (6) do programa quanto a cláusula (7) exprimindo a consulta.

Uma vez de posse deste conceito podemos então definir uma equivalência entre programas e consultas em Prolog e programas e consultas em cláusulas definidas da seguinte forma:

Definição 10.11:

Sejam P e C um programa e uma consulta Prolog e R e D uma regularização de P e C. Um programa em cláusulas definidas T e uma consulta E a T são *equivalentes* a P e C se e somente se:

- (i) uma cláusula unitária Prolog " $A.$ " ocorre em R se e somente se existe uma cláusula " $A \leftarrow$ " em T ;
- (ii) uma cláusula não-unitária Prolog " $A \leftarrow B_1 \wedge \dots \wedge B_n.$ " ocorre em R se e somente se existe uma cláusula " $A \leftarrow B_1 \dots B_n$ " em T ;
- (iii) D é da forma " $\leftarrow C_1 \wedge \dots \wedge C_m.$ " se e somente se E for da forma " $C_1 \wedge \dots \wedge C_m$ ".

Note que:

- a definição exige uma regularização prévia de P e C para eliminar variáveis anônimas e átomos em mais de um papel;
- T é uma simples transformação sintática de R ;
- D é a representação clausal da negação de E .

Uma vez introduzidos estes conceitos, podemos então transferir para Prolog todas as definições relativas às semânticas declarativa e procedural das seções 9.2, 9.3 e 9.4. Para tal, dados um programa Prolog P e uma consulta Prolog C , basta substituí-los por um programa T e uma consulta E equivalentes no contexto de cláusulas definidas.

Em particular, podemos transferir para Prolog os importantes conceitos de resposta correta e de resposta computada por resolução-LSD.

10.3.2 Semântica Operacional

Esta seção define informalmente uma máquina abstrata, chamada *máquina Prolog* no que se segue, capturando as principais características dos interpretadores e compiladores Prolog. A máquina abstrata oferece assim uma semântica operacional para a linguagem Prolog.

A máquina Prolog é um particular procedimento de refutação baseado em resolução-LSD tal que:

- o literal selecionado de cada cláusula é sempre o mais à esquerda, ou seja, a função de seleção utilizada é a padrão;
- as cláusulas do programa são selecionadas na ordem em que ocorrem no programa.

Os interpretadores ou compiladores para Prolog que seguem estas duas estratégias de seleção são chamados de *padrão*.

A máquina aceita como entrada um programa Prolog **P** e uma consulta Prolog **C** e produz como saída **FALHA** ou **SUCESSO**. No segundo caso, a saída também inclui uma substituição θ para as variáveis de **C**. A máquina comporta-se de tal forma que, se a saída é **FALHA**, não há respostas corretas de **C** a **P** e, se a saída é **SUCESSO**, θ é uma resposta correta de **C** a **P**. Mas a máquina poderá divergir e não produzir qualquer saída, tanto no caso de haver uma resposta correta de **C** a **P**, quanto no caso contrário. Embora não seja considerado nesta seção, a máquina pode ser modificada para produzir também respostas corretas alternativas, se necessário.

A máquina inicialmente modifica o programa **P** e a consulta **C**, eliminando as variáveis anônimas e os átomos ocorrendo em mais de um papel. Como visto na seção 10.3.1, este passo é chamado de regularização. Portanto, ignorando pequenas variações sintáticas, resolução-LSD pode ser aplicada diretamente ao conjunto de cláusulas Prolog resultante da regularização. Em seguida, a máquina simula o caminhamento em pré-ordem em uma particular árvore de refutação para as cláusulas resultantes da regularização, conforme ilustrado pelo exemplo a seguir.

Exemplo 10.10:

Seja **P** o seguinte programa Prolog:

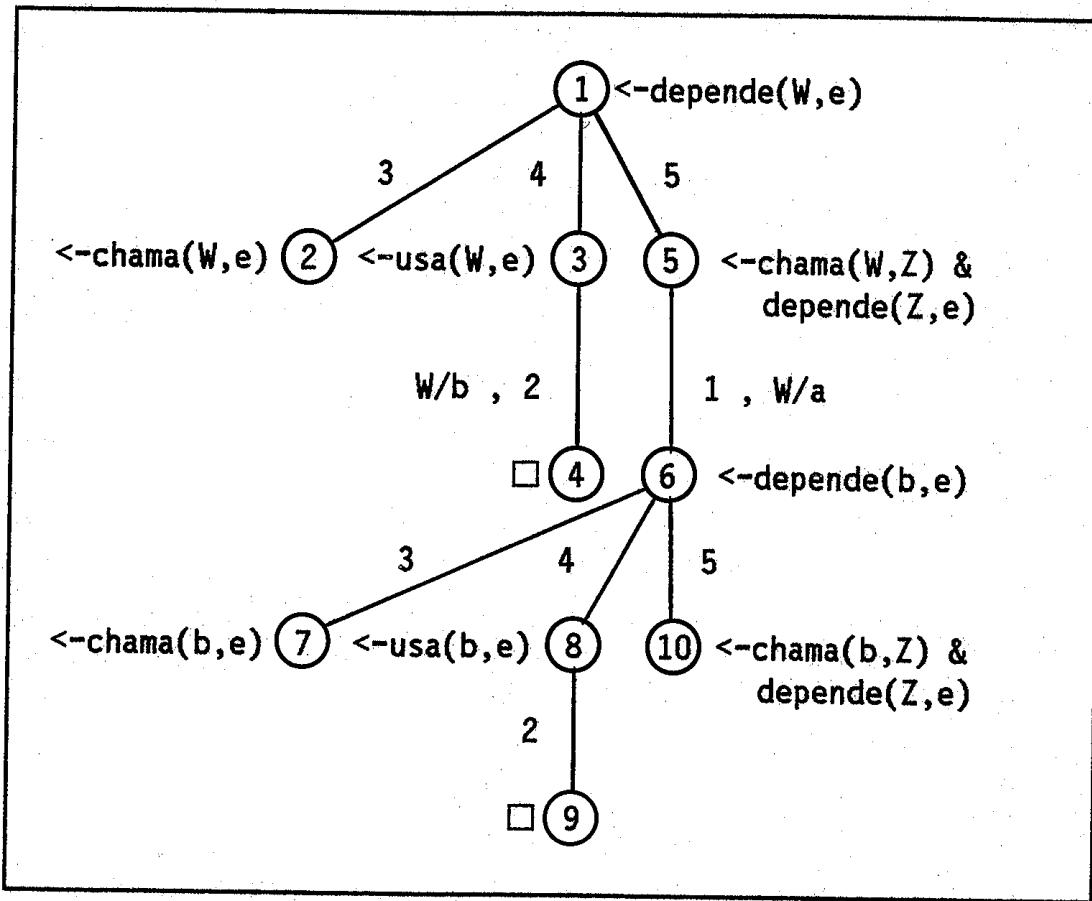
1. chama(a,b).
2. usa(b,e).
3. depende(X,Y) \leftarrow chama(X,Y).
4. depende(X,Y) \leftarrow usa(X,Y).
5. depende(X,Y) \leftarrow chama(X,Z) & depende(Z,Y).

e **C** a seguinte consulta Prolog:

6. \leftarrow depende(W,e).

Note que **P** e **C** já estão regularizados.

A árvore de refutação cujo caminhamento a máquina Prolog simula será a seguinte:



Esta é uma árvore de refutação por resolução-LSD construída de acordo com a seguinte estratégia:

- rotula-se a raiz com a consulta C;
- para cada nó N da árvore, com rótulo " $<-L_1 \& \dots \& L_m$ ":
 - seleciona-se o literal L_1 mais à esquerda;
 - para cada cláusula do programa " $M \leftarrow M_1 \& \dots \& M_n$ ", na ordem em que ocorre no programa e renomeando preliminarmente as variáveis se necessário, se a cabeça M unifica com L_1 , tendo θ como unificador mais geral, gera-se um novo filho de N rotulado com a cláusula objetivo " $<- (M_1 \& \dots \& M_n \& L_2 \& \dots \& L_m)\theta$ ".

Para facilitar o entendimento desta estratégia, os rótulos das arestas indicam as cláusulas do programa que geraram cada filho. Por exemplo, a cláusula 3 gerou o nó 2.

A máquina simula então o caminhamento em pré-ordem desta árvore. Novamente para facilitar o entendimento, os rótulos dos nós indicam a ordem em que a máquina os visita.

A máquina retorna **SUCESSO** assim que deteta o primeiro (em pré-ordem) nó de sucesso, que neste exemplo é o nó 4. Como a consulta possui a variável **W**, a saída inclui também a substituição **{W/b}**. Note que, de fato, esta substituição é uma resposta correta de **C** a **P** pois **P** implica logicamente "depende(**b,e**)".

A máquina pode ser modificada para continuar a pesquisa por outro nó de sucesso, que neste caso será o nó 9, retornando novamente a saída **SUCESSO** acompanhada da substituição **{W/a}**. Novamente, note que esta substituição é uma resposta correta de **C** a **P** pois **P** implica logicamente "depende(**a,e**)".

Note que o caminhamento em pré-ordem leva inevitavelmente a problemas de divergência, bastando para isto que a máquina comece a simular o caminhamento por um ramo infinito da árvore de refutação. O exemplo abaixo ilustra este ponto.

Exemplo 10.11:

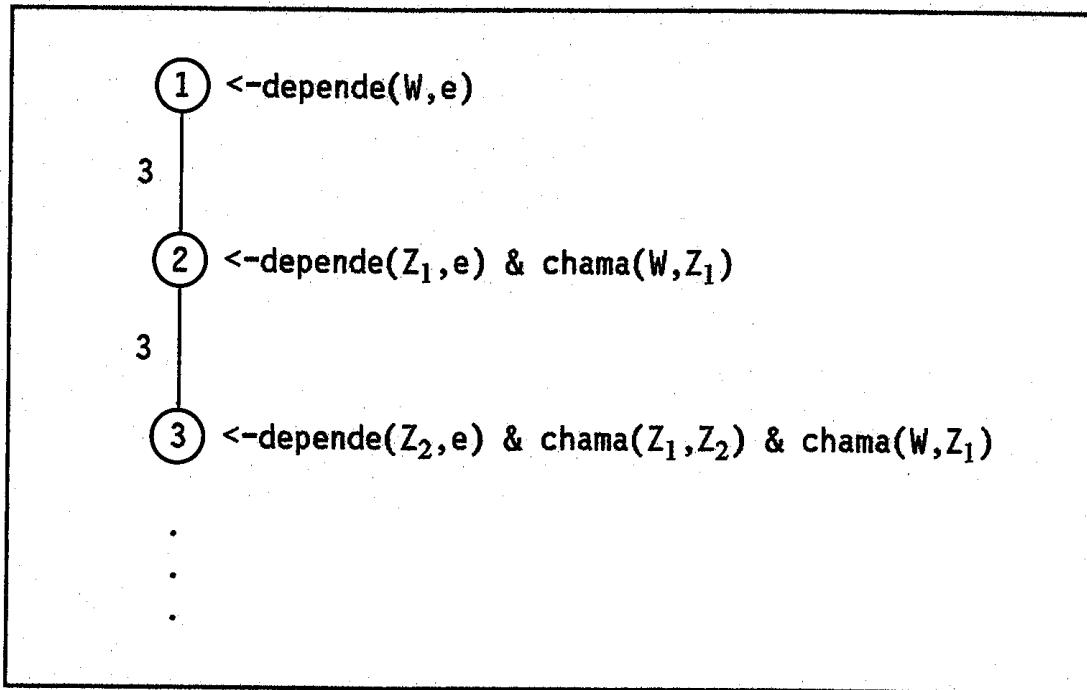
Seja **P** o seguinte programa Prolog:

1. **chama(a,b).**
2. **usa(b,e).**
3. **depende(X,Y) <- depende(Z,Y) & chama(X,Z).**
4. **depende(X,Y) <- chama(X,Y).**
5. **depende(X,Y) <- usa(X,Y).**

e **C** a seguinte consulta Prolog:

6. **<- depende(W,e).**

Note que **P** difere do programa do exemplo anterior apenas pela inversão da ordem das cláusulas (3) e (5) e inversão da ordem dos literais do corpo da cláusula (3) acima. No entanto, estas inversões são fatais pois geram uma árvore de refutação com um ramo infinito precedendo todos os nós de sucesso:



A máquina divergirá então tentando simular o caminhamento em pré-ordem por este ramo infinito. Portanto, não retornará qualquer saída.

Os dois exemplos anteriores ilustram claramente que o comportamento da máquina Prolog, ou seja, a semântica operacional de Prolog, depende da ordem das fórmulas atômicas dentro de uma cláusula e da ordem das cláusulas dentro do programa. Por outro lado, estes dois fatores não tem qualquer influência sobre a semântica declarativa de Prolog.

O resto desta seção apresenta duas especificações diferentes, mas equivalentes, da máquina Prolog. Esta parte pode ser dispensada em uma leitura a nível introdutório.

A máquina mantém essencialmente:

- a *cláusula objetivo corrente* (O);
- o *número da cláusula de entrada corrente* (N), que é o número da última cláusula de entrada resolvida contra a cláusula objetivo corrente;
- a *substituição corrente* (θ), que é a composição de todas as substituições efetuadas para gerar a cláusula objetivo corrente.

As duas especificações da máquina Prolog seguem, intuitivamente, a seguinte estratégia:

1. Inicialmente O é igual à consulta C ; N é 0, indicando que nenhuma cláusula do programa foi resolvida contra C ; e θ é a lista de substituições simples da forma X/X , onde X é uma variável que ocorre em C .
2. Selecionando para unificação o literal L mais à esquerda de O , a máquina deriva, através de um *passo de avaliação*, um novo valor para O . Esta derivação se dá pela unificação de L com a cabeça M da primeira cláusula D que ocorre em P e que não foi utilizada até então para este valor de O . Ou seja, D é a primeira cláusula que ocorre em P após a cláusula de ordem N . Note que uma renomeação preliminar das variáveis de D pode ser necessária.

Os valores correntes de O , N e θ são armazenados para utilização posterior no passo de retrocesso.

O novo valor de O é a cláusula formada substituindo-se L pelo corpo de D e aplicando-se o unificador mais geral utilizado, digamos, φ ; o novo valor de N será 0; e o novo valor de θ é a composição do valor anterior com φ .

3. Se o novo valor de O for a cláusula vazia, então a máquina Prolog retorna **SUCESSO**, junto com a composição de todas as substituições efetuadas sobre as variáveis de C , que é o valor de θ .
4. Se não for possível unificar L com a cabeça de alguma cláusula de P , o *passo de retrocesso* é automaticamente ativado. Retorna-se ao passo (2), considerando-se como valor de O , N e θ os seus valores anteriores.
5. Se o passo de retrocesso atingiu novamente a consulta C e o literal mais à esquerda de C não puder ser unificado com a cabeça de mais nenhuma cláusula de P , a máquina Prolog pára e retorna **FALHA**.

As duas especificações da máquina Prolog diferem no modo como gerenciam o procedimento de retrocesso, ou seja, no modo como mantém os vários valores da cláusula de objetivos corrente, do número da cláusula de entrada corrente e da substituição corrente. A primeira especificação implementa o procedimento de retrocesso implicitamente através de recursividade, enquanto que a segunda implementa tal procedimento explicitamente através de uma pilha, chamada *pilha de refutação*.

No que se segue, denotaremos a seqüência de cláusulas de um programa Prolog R por R_1, \dots, R_n e o número de cláusulas em R por $|R|$.

MÁQUINA PROLOG RECURSIVA

```
prolog(P,C)
```

```
/*
```

entrada:

P - um programa Prolog

C - uma consulta Prolog

saída:

FALHA - indicando que não há respostas corretas de *C* a *P*

SUCESSO - acompanhada de uma resposta correta de *C* a *P*

```
*/
```

```
begin
```

```
/* inicialização */
```

R, O := regularização de *P* e *C*;

θ := lista de substituições simples da forma *X/X*,
onde *X* é uma variável que ocorre em *O*;

```
/* pesquisa de um nó de sucesso */
```

```
mp(O,θ;β,OK);
```

```
/* saída */
```

```
if OK
```

then return 'SUCESSO', β;

else return 'FALHA'

```
end
```

MÁQUINA PROLOG RECURSIVA (Cont.)

mp($O, \theta; \beta, OK$)

/*

entrada:

R - um programa Prolog, passado implicitamente

O - uma consulta Prolog

θ - uma substituição

saída:

se a cláusula vazia não foi gerada

$\beta = \theta$

$OK = \text{FALSO}$

se a cláusula vazia foi gerada

$\beta = \text{composição de } \theta \text{ com as novas substituições efetuadas}$

$OK = \text{VERDADEIRO}$

*/

begin

$OK := \text{FALSO};$

for $N = 1$ **to** $|R|$ **while** $\neg OK$ **do**

begin

/* tentativa de unificação */

tentaunificar($O, R_N, \theta; E, \beta, OK$);

if $OK \wedge E \neq \square$

then

/* pesquisa continua recursivamente */

mp($E, \beta; \beta, OK$);

end

end

ROTINA DE UNIFICAÇÃO

tentaunificar(*O,D,θ;E,β,OK*)

/*

entrada:

O - uma cláusula objetivo Prolog

D - uma cláusula definida Prolog

θ - uma substituição

saída:

se *O* é extensível por *D*:

E = extensão de *O* por *D*

β = $\theta \circ \varphi$, onde φ foi o u.m.g usado na extensão

OK = VERDADEIRO

se *O* não é extensível por *D*:

OK = FALSO

*/

begin

D' := renomeação de *D* em presença de *O*;

if a cabeça de *D'* unifica com o literal mais à esquerda de *O*

then

begin

E := " $\leftarrow (M_1 \& \dots \& M_n \& L_2 \& \dots \& L_m) \varphi$ ",

se *O* é da forma " $\leftarrow L_1 \& \dots \& L_m$ ",

D' é da forma " $M \leftarrow M_1 \& \dots \& M_n$ ",

e φ é um u.m.g de *M* e *L₁*;

β := $\theta \circ \varphi$;

OK := VERDADEIRO;

end

else

OK := FALSO;

end

MÁQUINA PROLOG ITERATIVA

prolog(*P,C*)

/*

entrada:

P - um programa Prolog

C - uma consulta Prolog

saída:

FALHA - indicando que não há respostas corretas de *C* a *P*

SUCESSO - acompanhada de uma resposta correta de *C* a *P*

*/

begin

R, O := regularização de *P* e *C*;

N := 0;

θ := lista de substituições simples da forma *X/X*,
onde *X* é uma variável que ocorre em *O*;

PRF := CRIA_PILHA;

```

/* pesquisa da cláusula vazia */
repeat forever
begin
    /* tentativa de unificação */
    OK := FALSO;
    for i = N + 1 to |R| while ¬OK do
        tentaunificar(O,Ri,θ;E,β,OK);
    if OK
        then /* unificação bem sucedida */
            if E = □
                then /* pesquisa termina com sucesso */
                    return 'SUCESSO', β;
            else /* pesquisa iterativa continua */
                begin
                    PRF := ADICIONA((O,i,θ),PRF);
                    O := E;
                    N := 0;
                    θ := β;
                end
        else /* unificação mal sucedida */
            if VAZIA(PRF)
                then /* não há nós de sucesso */
                    return 'FALHA';
            else /* retrocesso */
                begin
                    (O,N,θ) := TOPO(PRF);
                    PRF := REMOVE(PRF);
                end
            end
    end
end

```

Nota: O programa acima utiliza as seguintes operações sobre pilhas:

- | | |
|------------|---|
| CRIA_PILHA | - cria uma nova pilha, inicialmente vazia |
| ADICIONA | - adiciona um elemento ao topo da pilha |
| REMOVE | - remove o elemento do topo da pilha |

TOPO	- retorna o elemento do topo da pilha
VAZIA	- retorna VERDADEIRO se a pilha estiver vazia

Finalmente observamos que ambos os algoritmos podem ser facilmente modificados para pesquisar mais de uma resposta correta. Como na maioria dos interpretadores Prolog, o usuário requererá respostas alternativas forçando o procedimento de retrocesso através do comando ";", abordado na seção 10.4, ou "fail", definido na seção 11.4.2.

10.4 CODIFICAÇÃO DE CLÁUSULAS PROLOG

Esta seção fixa (e de alguma forma repete) conceitos já apresentados nas seções anteriores através de exemplos sobre codificação, interpretação e submissão de cláusulas Prolog. Inclui também um programa Prolog relativo a relações familiares e uma sessão Prolog com este programa. As duas subseções iniciais tecem considerações, respectivamente, sobre a área de trabalho Prolog e sobre termos Prolog em um programa.

10.4.1 Área de Trabalho Prolog

O ambiente de desenvolvimento e execução de programas Prolog é uma área de memória, reservada ao usuário, referida a seguir como *Área de Trabalho Prolog* (neste texto, abreviada algumas vezes por ATP). Esta área, que permanece ativa durante uma sessão com a máquina Prolog e é associada a um editor, contém:

- cláusulas Prolog predefinidas (conhecidas como *primitivas Prolog*) que são carregadas quando a máquina Prolog é ativada;
- cláusulas Prolog correspondentes aos programas e procedimentos em desenvolvimento ou utilização;
- cláusulas Prolog independentes definidas interativamente pelo usuário.

Programas, procedimentos ou cláusulas independentes Prolog podem ser submetidos diretamente à máquina Prolog, mas, preferencialmente, devem ser criados em arquivos usando-se o editor normalmente associado à linguagem. Prolog permite ler e carregar na ATP arquivos que contenham programas, procedimentos ou cláusulas independentes Prolog, em um processo de consulta aos mesmos (normalmente efetuado através de comandos extralógicos conhecidos por *consult*, *reconsult* e *edconsult* descritos na seção 11.3). Após comandos de consulta, os programas,

procedimentos e cláusulas independentes consultados podem ser "executados" através da submissão de cláusulas objetivo relativas aos mesmos. Os editores permitem editar programas Prolog, interagir com a área de trabalho Prolog e com o ambiente computacional hospedeiro e, também, gerenciar o armazenamento e a recuperação dos programas e procedimentos em memória secundária.

10.4.2 Termos Prolog

A estrutura de dados básica propiciada pela linguagem Prolog é o termo Prolog, que, conforme visto na seção 10.2, pode ser uma constante, uma variável ou um termo funcional Prolog da forma $f(t_1, \dots, t_n)$, onde f é um átomo e t_1, \dots, t_n são também termos. Esta subseção descreve como os termos Prolog, em suas várias formas, são usados para representar objetos de um domínio de problema.

As constantes Prolog denotam elas mesmas.

As variáveis Prolog denotam objetos arbitrários do domínio de discurso, cujas estruturas são desconhecidas, podendo ser instanciadas por termos durante uma computação. O escopo de uma variável é limitado à cláusula Prolog onde ela ocorre, ou seja, variáveis são locais às cláusulas onde elas ocorrem. Deste modo, variáveis com o mesmo nome em cláusulas Prolog distintas, representam, em geral, objetos distintos. A variável anônima, um tipo particular de variável Prolog, identificada por um único símbolo ("*" neste texto), apresenta um significado especial: cada ocorrência de "*" como nome de variável, em uma dada cláusula, representa uma variável distinta (ou seja, máquinas Prolog substituem cada ocorrência de "*" em uma cláusula por uma distinta variável).

Os termos funcionais Prolog denotam, em forma linearizada, objetos estruturados (objetos com um ou mais componentes) do domínio de discurso, ou seja, objetos que são convenientemente representados por estruturas em árvores. Embora possam ter vários componentes, os objetos estruturados são tratados por Prolog, através dos símbolos funcionais e seus argumentos, como objetos simples ou recursivamente estruturados. Deve-se observar que, em Prolog, termos funcionais não são definidos explicitamente como funções (não computam valores), mas constituem-se em uma notação, na qual são tratados como símbolos indefinidos, usada para nomear indiretamente objetos do universo de discurso ou para construir estruturas de dados, como registros, árvores ou listas. As propriedades dos termos funcionais são deduzidas por Prolog a partir do

modo como eles aparecem como argumentos de símbolos predicativos ou de outros termos funcionais.

Exemplo 10.12: Termos Prolog

(a) Constantes Prolog:

1, janeiro, 1999, 5, 6, 0, 'Joao Silva', "a b", 1e0A

(b) Variáveis Prolog:

DD, MM, AA, X, Y, Z, S, C, Co, No, Po, Ni, *

(c) Termos Funcionais Prolog:

```
data(1,janeiro,1999),
data(DD,MM,AA),
ponto(5,6),
ponto(X,Y,Z),
suba(S),
pegue(suba(empurre(C,S,0))),
funcionario(codigo(Co),nome(No),posicao(Po),nivel(N1))
```

Observe a analogia entre o último termo funcional exemplificado acima e estruturas tipo registro suportadas pelas linguagens de programação tradicionais.

10.4.3 Cláusula Unitária Básica

Uma cláusula unitária básica tem a forma:

$$p(t_1, t_2, \dots, t_q).$$

onde, "p" é um átomo e t_k , $k \in [1, q]$, são constantes, neste particular tipo de cláusula Prolog.

A cláusula unitária básica representa um fato relativo a um determinado domínio de problema. A adição, na área de trabalho Prolog, de uma cláusula unitária básica como:

$$\text{pai('Joao VI',miguel)}.$$

que, para o programador, significa que "Joao VI é o pai de Miguel" (interpretação pretendida), fornece as seguintes informações à máquina Prolog:

- "pai" ocorre no papel de um símbolo predicativo de aridade 2;
- os argumentos "'Joao VI'" e "miguel" são constantes, pois o argumento "'Joao VI'" está entre apóstrofos e o argumento "miguel" inicia com letra minúscula;
- a relação binária denotada por "pai" é verdade para os objetos denotados por "'Joao VI'" e "miguel".

Deve-se observar que para a cláusula acima foi implicitamente suposto que "'Joao VI'" é pai de "miguel" e não o contrário. Ou seja, a ordem dos argumentos em uma fórmula atômica de uma cláusula Prolog é arbitrária. Mas, uma vez definida esta ordem para uma dada fórmula atômica, o programador deve ser consistente em relação à mesma ao longo de um programa Prolog. Evidentemente, uma cláusula unitária básica simétrica, como a cláusula "casado('Pedro I',leopoldina)", exige a codificação adicional de uma segunda cláusula, no caso a cláusula "casado(leopoldina,'Pedro I').", para levar em consideração as duas ordenações possíveis dos argumentos.

Exemplo 10.13: Cláusulas Unitárias Básicas

`mulher(leopoldina).`

leia-se: "leopoldina é uma mulher"

`mae('Carlota Joaquina',miguel).`

leia-se: "Carlota Joaquina é a mãe de Miguel"

Exemplo 10.14: Base de Fatos sobre a Dinastia de Bragança

Um conjunto de cláusulas unitárias básicas constitui uma Base de Fatos. Elas adicionam informação na área de trabalho Prolog. O programa Prolog apresentado a seguir, chamado PDBRAGA, define uma Base de Fatos sobre a Dinastia de Bragança. A codificação de relações familiares é um exemplo comum de programação Prolog pois, além de representar uma área de conhecimento geral, permite mostrar claramente a natureza declarativa da linguagem. O programa PDBRAGA utiliza os seguintes átomos e constantes:

Átomos no papel de símbolos predicativos unários: homem, mulher

Átomos no papel de símbolos predicativos binários: pai, mae

Constantes Prolog: 'Pedro III', 'Joao VI', 'Pedro I', miguel, 'Pedro II', 'Maria I', 'Carlota Joaquina', leopoldina e 'Maria da Gloria'

A interpretação pretendida para as fórmulas atômicas Prolog será a seguinte:

homem(X)	X é um homem
mulher(X)	X é uma mulher
pai(X,Y)	X é o pai de Y
mae(X,Y)	X é a mãe de Y

/*

Programa PDBRAGA

*/

```

homem('Pedro III').
homem('Joao VI').
homem('Pedro I').
homem(miguel).
homem('Pedro II').

mulher('Maria I').
mulher('Carlota Joaquina').
mulher(leopoldina).
mulher('Maria da Gloria').

pai('Pedro III','Joao VI').
pai('Joao VI','Pedro I').
pai('Joao VI',miguel).
pai('Pedro I','Pedro II').
pai('Pedro I','Maria da Gloria').

mae('Maria I','Joao VI').
mae('Carlota Joaquina','Pedro I').
mae('Carlota Joaquina',miguel).
mae(leopoldina,'Pedro II').
mae(leopoldina,'Maria da Gloria').

```

10.4.4 Cláusula Objetivo

Uma cláusula objetivo Prolog corresponde a uma consulta atômica ou conjuntiva e força a execução de um programa Prolog, permitindo:

- certificar-se a respeito de relacionamentos entre objetos do domínio do problema;
- recuperar dedutivamente informação armazenada.

A máquina Prolog responderá (sempre em negrito, neste texto) a uma cláusula objetivo sem variáveis livres, supondo a inexistência de laços infinitos, com mensagens de **SUCESSO** ou **FALHA**. Uma mensagem de **SUCESSO** indica que a cláusula objetivo pode ser provada e uma mensagem de **FALHA** indica o contrário.

Exemplo 10.15: Cláusulas Objetivo Básicas

`<-pai('Pedro I','Pedro II').`

leia-se: "Pedro I é o pai de Pedro II?"

`<-mae(leopoldina,'Pedro I').`

leia-se: "Leopoldina é a mãe de Pedro I?"

`<-rei('Pedro III',inglaterra).`

leia-se: "Pedro III é Rei da Inglaterra?"

Sessão 10.1: Submissão de Cláusulas Objetivo Básicas

Supondo o programa PDBRAGA carregado na ATP, tem-se:

`<-pai('Pedro I','Pedro II').`

SUCESSO

`<-mae(leopoldina,'Pedro I').`

FALHA

`<-rei('Pedro III',inglaterra).`

FALHA

Note que, nesta sessão, as cláusulas objetivo correspondem a consultas atômicas.

Cláusulas objetivo podem ser estendidas pela inclusão de variáveis. Neste caso, se a resposta for **SUCESSO**, a máquina Prolog repete a cláusula objetivo com as substituições ocorridas nas variáveis da mesma, ou então, conforme adotado neste texto, fornece as substituições destas variáveis após a resposta **SUCESSO**. Uma *resposta alternativa* para uma dada cláusula objetivo pode ser solicitada teclando-se ponto e vírgula (";") após uma resposta **SUCESSO**. Caso não mais existam respostas alternativas, Prolog encerra a execução do programa respondendo com uma mensagem de **FALHA** e aguarda por entrada adicional.

Evidentemente, o formato da resposta de um programa Prolog à uma cláusula objetivo Prolog é dependente do dialeto considerado e não necessariamente coincide com o formato proposto neste texto.

Exemplo 10.16: Cláusulas Objetivo com Variáveis

`<-irma('Maria da Gloria',X).`

leia-se: "existe um X tal que Maria da Glória é irmã de X?"

`<-irma(X,Y) & pai(*,Y).`

leia-se: "existem X e Y tais que X seja irmã de Y e Y tenha pai?"

`<-mae(X,'Pedro I') & mae(X,miguel).`

leia-se: "existe um X, tal que seja, simultaneamente, mãe de Pedro I e de Miguel?"

`<-pai(X,'Pedro I') & pai('Pedro I',Y).`

leia-se: "existem X e Y tais que X é pai de Pedro I e Pedro I é pai de Y?"

Sessão 10.2: Submissão de Cláusulas Objetivo com Variáveis

Supondo o programa PDBRAGA carregado na ATP, tem-se:

`<-irma('Maria da Gloria',X).`

SUCESSO: X= 'Pedro II' ;

FALHA

`<-irma(X,Y) & pai(*,Y).`

SUCESSO: X= 'Maria da Glória'

```

Y= 'Pedro II' ;
FALHA

<-mae(X,'Pedro I') & mae(X,miguel).
SUCESSO: X= 'Carlota Joaquina'

<-pai(X,'Pedro I') & pai('Pedro I',Y).
SUCESSO: X= 'Joao VI'
Y= 'Pedro II'

```

Note que, nesta sessão, a primeira cláusula objetivo corresponde a uma consulta atômica e as demais a consultas conjuntivas.

10.4.5 Cláusula Não-Unitária

Uma cláusula não-unitária permite expressar uma implicação entre relações existentes no domínio de problema. De um modo geral, implicações podem representar associações em geral, como: regras, definições, dependências de causa-efeito entre cláusulas unitárias, heurísticas, conhecimento, restrições de integridade, metaregras definindo métodos de arbitragem entre regras e metaregras definindo regras de aquisição de novas regras. Cláusulas não-unitárias permitem estender uma Base de Fatos, transformando-a em uma Base de Dados Dedutiva, na qual:

- átomos representam nomes de relações;
- cláusulas unitárias representam ênuplas das relações;
- cláusulas não-unitárias representam regras de dedução;
- cláusulas objetivo representam consultas que permitem recuperar valores de uma ou mais relações.

Exemplo 10.17: Cláusulas Não-Unitárias

`filho(X,Y) <- homem(X) & pai(Y,X).`

leia-se: "para todo X e Y, X é filho de Y se X for homem e Y for pai de X"

`irma(X,Y) <- mulher (X) &
 pai(Pa,X) & mae(Ma,X) &
 pai(Pa,Y) & mae(Ma,Y).`

leia-se: "para todo X, Y, Pa e Ma,

X é irmã de Y
 se X for mulher
 e os pais de X forem Pa e Ma
 e os pais de Y forem os mesmos Pa e Ma"

Nota: Observe que a codificação acima não evita que as variáveis X e Y sejam instanciadas por um mesmo objeto (ou seja, que uma dada pessoa seja considerada irmã de si mesma). Esta codificação será corrigida através de um exemplo na seção 11.2 do capítulo seguinte.

Exemplo 10.18: Uma Extensão do Programa PDBRAGA

A Base de Fatos do programa PDBRAGAX é a mesma do programa PDBRAGA. Esta Base de Fatos torna-se dedutiva com o acréscimo de cláusulas não-unitárias que estabelecem relações familiares através dos seguintes átomos no papel de símbolos predicativos binários: filho, filha, irmão, irmã, tio, tia, sobrinho, sobrinha, primo, prima, avo, avoh e ancestral. O significado destes átomos torna-se evidente através dos nomes adotados para os mesmos. Deve-se notar que o programa pode ser visualizado como uma Base de Conhecimento sobre relações familiares.

```
/*
  Programa PDBRAGAX
*/
homem('Pedro III').
homem('Joao VI').
homem('Pedro I').
homem(miguel).
homem('Pedro II').

mulher('Maria I').
mulher('Carlota Joaquina').
mulher(leopoldina).
mulher('Maria da Gloria').

pai('Pedro III','Joao VI').
pai('Joao VI','Pedro I').
pai('Joao VI',miguel).
pai('Pedro I','Pedro II').
pai('Pedro I','Maria da Gloria').
```

```

mae('Maria I','Joao VI').
mae('Carlota Joaquina','Pedro I').
mae('Carlota Joaquina',miguel).
mae('Leopoldina','Pedro II').
mae('Leopoldina','Maria da Gloria').

filho(X,Y) <- homem(X) & pai(Y,X).
filho(X,Y) <- homem(X) & mae(Y,X).
filha(X,Y) <- mulher(X) & pai(Y,X).
filha(X,Y) <- mulher(X) & mae(Y,X).

irmao(X,Y) <- homem(X) &
                pai(Pa,X) & mae(Ma,X) &
                pai(Pa,Y) & mae(Ma,Y).

irma(X,Y) <- mulher(X) &
                pai(Pa,X) & mae(Ma,X) &
                pai(Pa,Y) & mae(Ma,Y).

tio(X,Y) <- pai(Pay,Y) & irmao(X,Pay).
tio(X,Y) <- mae(May,Y) & irmao(X,May).

tia(X,Y) <- mae(May,Y) & irma(X,May).
tia(X,Y) <- pai(Pay,Y) & irma(X,Pay).

sobrinho(X,Y) <- homem(X) & tio(Y,X).
sobrinho(X,Y) <- homem(X) & tia(Y,X).

sobrinha(X,Y) <- mulher(X) & tio(Y,X).
sobrinha(X,Y) <- mulher(X) & tia(Y,X).

primo(X,Y) <- homem(X) &
                pai(Pax,X) & tio(Pax,Y).
primo(X,Y) <- homem(X) &
                pai(Pay,Y) & tio(Pay,X).
primo(X,Y) <- homem(X) &
                mae(Max,X) & tia(Max,Y).
primo(X,Y) <- homem(X) &
                mae(May,Y) & tia(May,X).

prima(X,Y) <- mulher(X) &
                pai(Pax,X) & tio(Pax,Y).
prima(X,Y) <- mulher(X) &
                pai(Pay,Y) & tio(Pay,X).
prima(X,Y) <- mulher(X) &
                mae(Max,X) & tia(Max,Y).
prima(X,Y) <- mulher(X) &
                mae(May,Y) & tia(May,X).

```

```

avo(X,Y)  :-  pai(X,Z)  &  pai(Z,Y).
avo(X,Y)  :-  pai(X,Z)  &  mae(Z,Y).

avoh(X,Y)  :-  mae(X,Z)  &  pai(Z,Y).
avoh(X,Y)  :-  mae(X,Z)  &  mae(Z,Y).

ancestral(X,Y)  :-  pai(X,Y).
ancestral(X,Y)  :-  mae(X,Y).
ancestral(X,Y)  :-  pai(X,Z)  &  ancestral(Z,Y).
ancestral(X,Y)  :-  mae(X,Z)  &  ancestral(Z,Y).

```

Sessão 10.3: Execução do Programa PDBRAGAX

Apresenta-se a seguir os resultados da execução do programa PDBRAGAX suposto carregado na área de trabalho Prolog. Observe a solicitação de resposta alternativa após cada resposta completada com SUCESSO.

<-irma(X,Y).

SUCESSO: X= 'Maria da Glória'

Y= 'Pedro II' ;

FALHA

<-tio(X,Y).

SUCESSO: X= miguel

Y= 'Pedro II' ;

SUCESSO: X= miguel

Y= 'Maria da Glória' ;

FALHA

<-ancestral(X,'Pedro I').

SUCESSO: X= 'João VI' ;

SUCESSO: X= 'Carlota Joaquina' ;

SUCESSO: X= 'Pedro III' ;

SUCESSO: X= 'Maria I' ;

FALHA

<-filho(X,'Carlota Joaquina') & pai(X,'Pedro II').

SUCESSO: X= 'Pedro I' ;

FALHA

<-pai(X, 'Pedro II') & ancestral('Pedro III', X).
 SUCESSO: X= 'Pedro I' ;
 FALHA

<-mae(X, 'Pedro I') & avoh(X, 'Pedro II').
 SUCESSO: X= 'Carlota Joaquina' ;
 FALHA

<-pai(X, Y) & tio(miguel, Y) & irmao('Pedro II', Y).
 SUCESSO: X= 'Pedro I'
 Y= 'Maria da Gloria' ;
 FALHA

<-pai(X, Y) & tio(Z, Y) & irmao('Pedro II', Y).
 SUCESSO: X= 'Pedro I'
 Y= 'Maria da Gloria'
 Z= miguel ;
 FALHA

10.5 PROCESSAMENTO DE LISTAS

Esta seção estende a sintaxe da linguagem Prolog básica para permitir a representação de listas e define várias operações em Prolog para o processamento de listas. Deve-se observar que os sistemas Prolog existentes, em contraste com os diversos dialetos da linguagem LISP, normalmente não incluem operações sobre listas entre suas primitivas.

10.5.1 Sintaxe para Listas

A representação de uma lista baseia-se na introdução de uma constante Prolog, "nil", para denotar a lista vazia e de um operador no papel de símbolo funcional binário, ".", para denotar o construtor de listas. Mais precisamente, tem-se:

Definição 10.12:

O conjunto dos termos Prolog denotando listas é definido recursivamente da seguinte forma:

- (i) nil é um termo Prolog denotando uma lista de comprimento zero chamada *lista vazia*;

(ii) se u é um termo Prolog e v é uma lista de comprimento $n-1$, então $(u.v)$ é um termo Prolog denotando uma lista de comprimento n . O termo Prolog u é a *cabeça* e o termo Prolog v é a *cauda* da lista $(u.v)$.

De acordo com esta definição, se e_1, e_2, \dots, e_n são termos, eles podem ser combinados para representar a seguinte lista de comprimento n :

$$(e_1.(e_2.(\dots(e_n.nil)\dots)))$$

A parentetização completa dos termos denotando listas é dispensável, assumindo-se que o agrupamento dos elementos ocorra pela direita. Considerando-se este fato, tem-se as seguintes representações:

plena: $(e_1.(e_2.(\dots(e_n.nil)\dots)))$

simplificada: $(e_1.e_2. \dots e_n.nil)$

onde " e_k " é o k -ésimo elemento da lista, podendo ser outra lista e nil desempenha o papel de último elemento da lista. A representação simplificada apresentada acima é conhecida por *notação por ponto*. Na literatura Prolog é conhecida também como *notação de Marseille* para listas.

Também comumente utilizada nas implementações Prolog é a *notação por colchete* (ou *notação de Edinburgh* para listas, na literatura Prolog), na qual $[]$ denota a lista vazia e $(u.v)$ escreve-se $[u|v]$. Deste modo, se e_1, e_2, \dots, e_n são termos, eles podem ser combinados para representar, na notação por colchete, a seguinte lista de comprimento n :

$$[e_1|[e_2|[\dots|[e_n|[]]\dots]]]$$

A colchetização completa dos termos denotando listas é dispensável, assumindo-se que o agrupamento dos elementos ocorra pela direita. Para tal, inicialmente omite-se $[]$. A seguir, se um símbolo " $|$ " é seguido por um colchete esquerdo, ele é substituído por vírgula com o colchete esquerdo e o correspondente colchete direito removidos. Tem-se então as seguintes representações por colchete:

plena: $[e_1|[e_2|[\dots|[e_n|[]]\dots]]]$

simplificada: $[e_1, e_2, \dots, e_n]$

onde " e_k " é o k -ésimo elemento da lista, podendo ser outra lista.

Observe que a notação por colchete é uma variante sintática para a notação por ponto (mais próxima da representação interna) no seguinte sentido:

- colchetes substituem parênteses como delimitadores;
- nil passa a ser denotado por [];
- [] passa a ser considerado implicitamente como último elemento da lista;
- o símbolo "," substitue "." para separar os elementos de uma mesma lista;
- o símbolo "|" substitue ":" para separar a cabeça da cauda de uma lista.

Exemplo 10.19: Notações por ponto e por colchete

Notação por ponto	Notação por colchete
nil	[]
(a. b. c. d. nil)	[a,b,c,d]
((a. b. nil). c. (d. e. f. nil). g. nil)	[[a,b],c,[d,e,f],g]
((A11. A12. nil). (A21. A22. nil). nil)	[[A11,A12],[A21,A22]]

Exemplo 10.20: Notação por colchete (Bratko [1986])

Na notação por colchete, uma mesma lista pode ser codificada de modos alternativos, como por exemplo:

[a,b,c] ou [a|[b,c]] ou [a,b|[c]] ou [a,b,c|[]]

Ou seja, nesta notação, pode-se codificar uma lista explicitando-se qualquer número de elementos iniciais da mesma seguidos por "|" e pela lista de elementos restantes.

Exemplo 10.21: Especificação, Cabeça e Cauda de Listas (Not. por ponto)

```

lista:      (a. b. c. d. nil)
cabeça:    a
cauda:     (b. c. d. nil)

lista:      ((a. b. nil). c. (d. e. f. nil). g. nil)
cabeça:    (a. b. nil)
cauda:     (c. (d. e. f. nil). g. nil)

lista:      ((A11. A12. nil). (A21. A22. nil). nil)
cabeça:    (A11. A12. nil)
cauda:     ((A21. A22. nil). nil)

```

Exemplo 10.22: Especificação, Cabeça e Cauda de Listas (Not. por colchete)

```

lista:      [a,b,c,d]
cabeça:    a
cauda:     [b,c,d]

lista:      [[a,b],c,[d,e,f],g]
cabeça:    [a,b]
cauda:     [c,[d,e,f],g]

lista:      [[A11,A12],[A21,A22]]
cabeça:    [A11,A12]
cauda:     [[A21,A22]]

```

10.5.2 Operações Básicas sobre Listas

O carácter recursivo inerente à definição de uma lista motiva definições recursivas para as operações sobre as mesmas, ou seja, definições através de:

- uma parte básica que define o valor da operação para valores conhecidos de seus argumentos, e
- uma parte recursiva, que define o valor da operação para valores de seus argumentos em termos do valor da operação para valores de alguma forma mais simples dos mesmos argumentos.

Considerando estes aspectos, as operações sobre listas podem ser definidas, de modo elegante, usando-se recursividade e algumas construções computacionais básicas que permitem:

- identificar uma lista vazia ou o último elemento de uma lista;
- construir uma lista de comprimento n dados o seu primeiro elemento (que pode ser uma lista) e uma lista de comprimento $n-1$;
- selecionar a cabeça de uma lista;
- selecionar a cauda de uma lista.

Na linguagem LISP, estas quatro construções são, respectivamente, "nil" e as funções "CONS", "CAR" e "CDR". Em Prolog, estas quatro construções estão embutidas na própria notação que a linguagem propicia para representação de listas. De fato, na notação por ponto, `nil` denota a lista vazia ou o último elemento da lista e `(H.T)`, dado que `.` é associativo à direita, denota uma lista cujo cabeça é `H` e a cauda `T`. Deste modo, a unificação de `(H.T)`, onde `H` e `T` são variáveis Prolog, com uma lista da forma `(e1.e2. ... en.nil)` resulta na substituição de `H` por `e1` e de `T` por `(e2. ... en.nil)`. Ou seja, `H` seleciona a cabeça da lista e `T` a cauda da lista.

Na notação por colchete, `[]` denota a lista vazia e `[H|T]` denota uma lista cujo cabeça é `H` e a cauda `T`. Deste modo, a unificação de `[H|T]`, onde `H` e `T` são variáveis Prolog, com uma lista da forma `[e1,e2,...,en]` resulta na substituição de `H` por `e1` e de `T` por `[e2,...,en]`, se $n > 1$, ou por `[]`, se $n = 1$. Ou seja, `H` seleciona a cabeça da lista e `T` a cauda da lista.

Alguns dialetos Prolog suportam as duas notações apresentadas para listas. A tabela de resumo destas notações e os dois exemplos que se seguem, mostrando unificações de listas L_1 e L_2 , elucidam e completam estes pontos.

Exemplo 10.23: Unificação de Listas (Notação por ponto)

lista L_1 :	<code>(a.nil)</code>
lista L_2 :	<code>(H.T)</code>
unificação:	<code>H/a</code> <code>T/nil</code>

lista L_1 :	<code>(a.b.c.d.nil)</code>
lista L_2 :	<code>(H.T)</code>
unificação:	<code>H/a</code>

$T/(b. c. d. \text{nil})$

lista L₁: (a. b. c. d. nil)

lista L₂: (E1. E2. E)

unificação: E1/a

E2/b

E/(c. d. nil)

lista L₁: ((a. b. nil). c. (d. e. f. nil). g. nil)

lista L₂: (H. T)

unificação: H/(a. b. nil)

T/(c. (d. e. f. nil). g. nil)

lista L₁: ((a. b. nil). c. (d. e. f. nil). g. nil)

lista L₂: ((X. Y). Z)

unificação: X/a

Y/(b. nil)

Z/(c. (d. e. f. nil). g. nil)

Exemplo 10.24: Unificação de Listas (Notação por colchete)

lista L₁: [a]

lista L₂: [H|T]

unificação: H/a

T/[]

lista L₁: [a,b,c,d]

lista L₂: [H|T]

unificação: H/a

T/[b,c,d]

lista L₁: [a,b,c,d]

lista L₂: [E1,E2|E]

unificação: E1/a

E2/b

E/[c,d]

lista L₁: [[a,b],c,[d,e,f],g]

lista L₂: [H|T]

unificação: H/[a,b]

T/[c,[d,e,f],g]

lista L_1 : $[[a,b],c,[d,e,f],g]$
 lista L_2 : $[[X|Y]|Z]$
 unificação: X/a
 $Y/[b]$
 $Z/[c,[d,e,f],g]$

Como indicam estes exemplos, funções típicas da linguagem LISP, que permitem extrair os elementos iniciais de uma lista (CAR,CADR,CADDR,...), são desnecessárias em Prolog, pois a própria notação e o conceito de unificação permitem recuperar os elementos iniciais de uma lista. Observe também que, como na linguagem LISP, o símbolo funcional binário “.” ou “|”, dado que é associativo à direita, permite extrair diretamente o primeiro elemento de uma lista. Mas, para extrair o último elemento de uma lista, é necessário definir uma operação recursiva sobre a mesma (como mostrado na seção 10.5.3).

Para facilidade de referência, os aspectos de representação e unificação de listas das notações por ponto e por colchete são resumidos a seguir.

Resumo das Notações por ponto e por colchete		
Notação por ponto (Marseille)	Notação por colchete (Edinburgh)	Unificação
nil	$[]$	lista vazia
$(H.T)$	$[H T]$	H unifica com a cabeça de uma lista T unifica com a cauda da mesma lista
$(E_1. \dots . E_n. \text{nil})$	$[E_1, \dots, E_n]$	E_k unifica com o k -ésimo elemento de uma lista
$(E_1. \dots . E_n)$	$[E_1, \dots, E_{n-1} E_n]$	E_k unifica com o k -ésimo elemento de uma lista, para $k \in [1, n-1]$ E_n unifica com a cauda da mesma lista

10.5.3 Outras Operações sobre Listas

A partir da notação Prolog para representar e processar listas e das construções básicas discutidas podemos definir operações sobre listas. O entendimento das operações apresentadas será fundamental para facilitar a compreensão dos exemplos de codificação e exercícios que serão apresentados nos capítulos seguintes. E também para codificação de outras operações sobre listas.

Operações sobre listas baseiam-se no conceito que uma lista é uma estrutura de dados que ou é vazia ou consiste de uma cabeça (que pode ser uma lista ou não) e uma cauda (que é uma lista). Todas as operações apresentadas a seguir são definidas recursivamente, sendo que, para cada operação, a condição de término é definida pela cláusula c_1 e a condição de recursividade pela cláusula c_2 que não fornece a solução para o processo recursivo. Sua função é remover (e processar, se necessário) sucessivamente a cabeça da lista e forçar o retorno para a cláusula c_1 , que testa o fim do processo recursivo. Observe também a convenção posicional adotada para os argumentos das operações definidas.

operação: lista

especificação: verifica se um termo é uma lista

argumento: termo

$c_1: \text{lista}(\text{nil}).$

$c_2: \text{lista}(*, L) \leftarrow \text{lista}(L).$

- A cláusula c_1 indica que `nil` é uma lista.
- A cláusula c_2 indica que o termo será uma lista se for possível decompor o mesmo em duas partes tais que a segunda seja uma lista.

operação: membro

especificação: verifica se um termo é membro de uma lista

1º argumento: termo

2º argumento: lista

$c_1: \text{membro}(X, X, *).$

$c_2: \text{membro}(X, *, Y) \leftarrow \text{membro}(X, Y).$

- A cláusula c_1 indica que o termo X é membro da lista se for a cabeça da lista.

- A cláusula c_2 indica que, o termo X será membro de uma lista cuja cabeça não é X , se, recursivamente, for membro da cauda da lista.

operação: **último**
 especificação: identifica o último elemento de uma lista
 1º argumento: último elemento da lista
 2º argumento: lista

$c_1: \text{ultimo}(X, X, \text{nil}).$
 $c_2: \text{ultimo}(X, *, Y) \leftarrow \text{ultimo}(X, Y).$

- A cláusula c_1 indica que o último elemento de uma lista de um único elemento é o próprio elemento.
- A cláusula c_2 indica que o último elemento de uma lista com mais de um elemento, é, recursivamente, o último elemento da cauda desta lista.

operação: **remove**
 especificação: elimina um termo de uma lista
 1º argumento: termo
 2º argumento: lista
 3º argumento: lista resultado

$c_1: \text{remove}(X, X, L, L).$
 $c_2: \text{remove}(X, Y, L_1, Y, L_2) \leftarrow \text{remove}(X, L_1, L_2).$

- A cláusula c_1 indica que se o termo X for a cabeça da lista, para removê-lo da mesma basta devolver a cauda da lista.
- A cláusula c_2 indica que, para remover-se um termo X de uma lista cuja cabeça não é X , deve-se, recursivamente, remover X da cauda da lista.

operação: **consec**
 especificação: verifica se 2 termos são consecutivos em uma lista
 1º argumento: primeiro termo
 2º argumento: segundo termo
 3º argumento: lista

$c_1: \text{consec}(X, Y, X, Y, *).$
 $c_2: \text{consec}(X, Y, *, L) \leftarrow \text{consec}(X, Y, L).$

- A cláusula c_1 indica que dois termos X e Y são consecutivos, se forem, respectivamente, o primeiro e o segundo elemento da lista.
- A cláusula c_2 indica que os dois termos X e Y serão consecutivos na lista cuja cabeça não é X , se, recursivamente, forem consecutivos na cauda desta lista.

operação: **concat**
 especificação: concatena duas listas
 1º argumento: primeira lista
 2º argumento: segunda lista
 3º argumento: lista concatenada

$c_1: concat(nil, L, L).$
 $c_2: concat(X. L_1, L_2, X. L_3) \leftarrow concat(L_1, L_2, L_3).$

- A cláusula c_1 indica que a concatenação da lista vazia com uma lista L resulta na própria lista L .
- A cláusula c_2 indica que a concatenação da lista, cuja cabeça é X e a cauda L_1 , com uma lista L_2 , resulta na lista cuja cabeça é X e a cauda L_3 , desde que, recursivamente, a concatenação da lista L_1 com L_2 resulte em L_3 .

operação: **inverte**
 especificação: inverte os elementos de uma lista
 1º argumento: lista
 2º argumento: lista invertida

$c_0: inverte(L_1, L_2) \leftarrow inverte(L_1, nil, L_2).$
 $c_1: inverte(nil, L, L).$
 $c_2: inverte(X. L_1, L_a, L) \leftarrow inverte(L_1, X. L_a, L).$

- A cláusula c_0 inicializa com nil uma lista auxiliar usada na inversão.
- A cláusula c_1 indica que a operação de inversão de uma lista termina quando a mesma se tornar a lista vazia pois neste caso a lista inversa sera a lista auxiliar por construção (conforme cláusula c_2).
- A cláusula c_2 indica que é possível inverter uma lista cuja cabeça é X e a cauda L_1 , sendo L_a uma lista auxiliar, de forma a obter uma lista L , se, recursivamente, a inversão de L_1 , passando X a ser a cabeça da lista auxiliar, resultar na mesma lista L .

Na notação por colchete a codificação destas operações torna-se:

operação: lista

```
c1: lista([]).
c2: lista([*|L]) <- lista(L).
```

operação: membro

```
c1: membro(X,[X|*]).
c2: membro(X,[*|Y]) <- membro(X,Y).
```

operação: ultimo

```
c1: ultimo(X,[X]).
c2: ultimo(X,[*|Y]) <- ultimo(X,Y).
```

operação: remove

```
c1: remove(X,[X|L],L).
c2: remove(X,[Y|L1],[Y|L2]) <- remove(X,L1,L2).
```

operação: consec

```
c1: consec(X,Y,[X,Y|*]).
c2: consec(X,Y,[*|L]) <- consec(X,Y,L).
```

operação: concat

```
c1: concat([],L,L).
c2: concat([X|L1],L2,[X|L3]) <- concat(L1,L2,L3).
```

operação: inverte

```
c0: inverte(L1,L2) <- inverte(L1,[],L2).
c1: inverte([],L,L).
c2: inverte([X|L1],La,L) <- inverte(L1,[X|La],L).
```

É importante ressaltar que as especificações apresentadas para as operações definidas devem ser entendidas em termos relativos, no sentido que a mesma operação pode ser usada com finalidades alternativas. As

sessões Prolog a seguir apresentam utilizações alternativas para algumas das operações definidas, supondo-as carregadas na ATP e codificadas na notação indicada.

Sessão 10.4: Operação "membro" (Notação por ponto)

```
/*
verificação se um termo é membro de uma lista
*/
<-membro(c, a. b. c. d. nil).
SUCESSO

/*
extração do primeiro elemento de uma lista
*/
<-membro(X, a. b. c. d. nil).
SUCESSO X= a

/*
extração de elementos sucessivos de uma lista
*/
<-membro(X, a. b. c. d. nil).
SUCESSO X= a ;
SUCESSO X= b ;
SUCESSO X= c ;
SUCESSO X= d ;
FALHA

/*
criação de listas com um dado elemento
*/
<-membro(m, X).
SUCESSO X= (m.V1) ;
SUCESSO X= (V1.m.V2) ;
SUCESSO X= (V1.V2.m.V3)
```

Sessão 10.5: Operação "remove" (Notação por colchete - Bratko [1986])

```
/*
  remoção de um elemento de uma lista
*/
<-remove(c, [a,b,c,d] , X).
SUCESSO:  X= [a,b,d]

/*
  remoção sucessiva de um elemento de uma lista
*/
<-remove(a, [a,b,a,a] ).
SUCESSO:  X= [b,a,a] ;
SUCESSO:  X= [a,b,a] ;
SUCESSO:  X= [a,b,a] ;
FALHA

/*
  adição sucessiva de um elemento em uma lista
*/
<-remove(a, X, [1,2,3]).
SUCESSO:  X= [a,1,2,3] ;
SUCESSO:  X= [1,a,2,3] ;
SUCESSO:  X= [1,2,a,3] ;
SUCESSO:  X= [1,2,3,a] ;
FALHA
```

Sessão 10.6: Operação "concat" (Notação por ponto)

```
/*
  verificação se duas listas dadas se
  concatenam em uma terceira lista dada
*/
<-concat(a.b.nil, c.d.nil, a.b.c.d.nil).
SUCESSO
```

```

/*
concatenação de duas listas
*/
<-concat(a.b.nil, c.d.nil, X).
SUCESSO: X= (a.b.c.d.nil)

/*
particionamento de uma lista em
todos pares de sublistas componentes
*/
<-concat(X, Y, a.b.c.d.nil).
SUCESSO: X= nil
          Y= (a.b.c.d.nil) ;
SUCESSO: X= (a.nil)
          Y= (b.c.d.nil) ;
SUCESSO: X= (a.b.nil)
          Y= (c.d.nil) ;
SUCESSO: X= (a.b.c.nil)
          Y= (d.nil) ;
SUCESSO: X= (a.b.c.d.nil)
          Y= nil ;
FALHA

```

Sessão 10.7: Operação "inverte" (Notação por colchete)

```

<-inverte([a,b,c,d], X).
SUCESSO: X= [d,c,b,a]

```

```

<-inverte(X, [d,c,b,a]).
SUCESSO: X= [a,b,c,d]

```

As operações definidas, em certos casos, permitem computar uma relação e suas inversas. Por exemplo, "concat" permite concatenar duas listas e, inversamente, decompor uma lista dada em todos os pares de sublistas componentes (ver penúltima sessão Prolog acima). Esta propriedade, que caracteriza um programa invertível (programa que não distingue entre argumentos de entrada e de saída), assegura um escopo amplo de aplicações para programas deste tipo. A operação "concat", por exemplo, devido a esta característica, torna-se uma das mais importantes em termos de processamento de listas.

Exemplo 10.25:

As cláusulas Prolog a seguir mostram como, em função de "concat", é possível simplificar as definições das operações "membro", "último" e "consec":

```
c1: membro(X,L)  :- concat(*, X.* , L).
c2: ultimo(X,L)  :- concat(*, X.nil , L).
c3: consec(X,Y,L)  :- concat(*, X.Y.* , L).
```

De acordo com estas codificações, tem-se:

Cláusula c₁: um termo X é membro de uma lista L, se for possível decompor L em duas sublistas tais que a segunda tenha X como cabeça.

Cláusula c₂: um termo X é o último elemento de uma lista L, se for possível decompor L em duas sublistas tais que a segunda contenha um único elemento X.

Cláusula c₃: os termos X e Y são consecutivos em uma lista L, se for possível decompor L em duas sublistas tais que a segunda contenha X como seu primeiro elemento e Y como seu segundo elemento.

Na notação por colchete estas codificações tornam-se:

```
c1: membro(X,L)  :- concat(*, [X|*] , L).
c2: ultimo(X,L)  :- concat(*, [X] , L).
c3: consec(X,Y,L)  :- concat(*, [X,Y|*] , L).
```

NOTAS BIBLIOGRÁFICAS

Bharath [1986], Rogers [1986], Burnham e Hall [1985], Ferguson [1981], Cuadrado e Cuadrado [1985] e Covington [1985,1986] contém introduções a Prolog, visando uma disseminação prática da linguagem.

Clark [1982] aborda, através de exemplos, a utilização de cláusulas definidas, enquanto Sowa [1982] contém uma discussão bastante completa da linguagem Prolog, baseando-se na versão de Prolog implementada na

Universidade de Waterloo. Coelho et alli [1980] é uma extensa coletânea de problemas resolvidos em Prolog.

O livro de Clocksin e Mellish [1984] representa a primeira contribuição em termos de disseminação da linguagem Prolog, com uma abordagem não pressupõe conhecimento prévio de Computação ou de Lógica.

O livro de Kluzniak e Szpakowicz [1985] admite que o leitor tenha alguma experiência em programação. Utiliza, com poucas variações documentadas, a versão do Edinburgh Prolog e descreve um interpretador Prolog para micros (escrito em Pascal e Prolog) fornecendo uma cópia completa das listagens. Apresenta também estudos de casos e uma apreciação de alguns dialetos Prolog.

Bratko [1986] apresenta Prolog em duas partes. A primeira parte introduz Prolog como uma linguagem de programação e a segunda parte introduz Prolog no contexto da área de Inteligência Artificial.

Giannesini et alli [1986] apresentam a experiência em Prolog e Prolog II do grupo de Marseille que, coordenado por Colmerauer, criou esta linguagem. A primeira parte do livro apresenta os princípios básicos de Prolog através de refinamentos em um programa inicial. A segunda parte aborda aplicações de Prolog em escrita de compiladores, compreensão de linguagem natural, base de dados e sistemas especialistas. Os apêndices apresentam Prolog II em termos de sintaxe, regras predefinidas, correspondência entre Prolog II e Micro-Prolog e entre Prolog II e Edinburgh Prolog.

Marcus [1986] e Garavaglia [1987] são livros voltados para uma introdução da linguagem e, principalmente, para mostrar o uso desta linguagem para desenvolvimento de aplicações em sistemas de base de dados, sistemas especialistas e sistemas de linguagem natural.

O livro de Sterling e Shapiro [1986] aborda Programação em Lógica, a linguagem Prolog, técnicas avançadas de programação Prolog e aplicações.

Walker et alli [1987] aborda, baseado no dialeto VM/Prolog (ou MVS/Prolog), os aspectos básicos e avançados de programação em Prolog e a utilização de Prolog para desenvolvimento de sistemas especialistas e de sistemas de linguagem natural. O capítulo 2 deste livro repete o trabalho citado de Sowa (Sowa [1982]), em uma forma revista, ampliada e orientada para VM/Prolog (ou MVS/Prolog). Neste capítulo encontra-se uma

discussão de métodos informais de análise de descrição de problemas e de consequente mapeamento de problemas em Prolog.

CAPÍTULO 11: A LINGUAGEM PROLOG ESTENDIDA

Este capítulo estende a linguagem Prolog básica apresentando as facilidades extra e metalógicas mais comuns às diversas implementações Prolog existentes.

As seções recomendadas para cada nível de leitura são:

nível introdutório: 11.2, 11.3, 11.4 (em parte) e 11.5

nível intermediário: 11.2, 11.3, 11.4, 11.5 e 11.6

11.1 INTRODUÇÃO

As facilidades extra e metalógicas expandem a linguagem Prolog permitindo simular estilos de programação mais próximos das linguagens de programação tradicionais. Para fins didáticos esta extensão será relatada em sete seções correspondendo às facilidades de:

- aritmética e comparação
- comunicação
- controle de execução
- processamento de cláusulas
- depuração

- programação na metalinguagem

As facilidades extralógicas são implementadas através de comandos. Alguns comandos com funções similares, mas com nomes distintos em outros dialetos Prolog, são apresentados como sinônimos. Para acomodar os comandos extralógicos a sintaxe básica de Prolog deve ser alterada para permitir a inserção dos mesmos, no corpo de uma cláusula, como se fossem fórmulas atômicas normais. Estes comandos devem ser ignorados quando se considera a interpretação declarativa de um programa Prolog. A máquina Prolog também deve ser modificada para acomodar estes novos comandos, segundo a definição semântica informal de cada um deles, apresentada neste capítulo. As modificações de sintaxe e semântica necessárias para acomodar programação na metalinguagem serão discutidas na seção pertinente.

11.2 FACILIDADES ARITMÉTICAS E DE COMPARAÇÃO

Programas em Lógica para efetuarem aritmética são elegantes mas não são práticos nem eficientes. Os dialetos Prolog existentes propiciam facilidades aritméticas e de processamento de cadeias de caracteres através de comandos predefinidos, que podem ser normalmente inseridos em uma cláusula no papel de fórmulas atômicas. Estes comandos fornecem um interface direto e conveniente para as capacidades aritméticas dos computadores.

Em adição, os dialetos Prolog suportam *expressões computáveis*, que são regidas pela semântica procedural de Prolog por serem sensíveis à ordem de processamento (quando avaliada, todos argumentos de uma expressão computável devem estar numericamente instanciados). Estas expressões são codificadas usando-se operadores predefinidos disponíveis nos diversos dialetos Prolog, de uma forma mais natural que aquela oferecida pelos comandos aritméticos e de processamento de cadeias de caracteres já mencionados.

Uma expressão computável é identificada posicionalmente como o argumento direito do operador infixo denotado por ":"= (ou "is" em outros dialetos), que dispara a avaliação da expressão computável, em um comando com a seguinte sintaxe:

X := *expressão-computável*

A execução deste comando resulta na instanciação de X pelo resultado da avaliação da expressão computável (podendo incluir concatenações com constantes Prolog).

O apêndice I apresenta tabelas resumindo os comandos aritméticos e de comparação mais usuais. Estas tabelas descrevem a sintaxe dos mesmos seguida pela sua semântica operacional. Este apêndice apresenta também uma tabela com um resumo dos operadores permitidos em uma expressão computável. O manual do particular dialeto Prolog considerado deverá ser consultado para acomodar as diferenças sintáticas existentes.

Seguem exemplos de utilização de expressões computáveis e um exemplo da utilização de um comando de comparação.

Exemplo 11.1: Expressão Computável

(a) Em uma Cláusula Não-Unitária:

```
calculo(Capital,Taxa,Dias,RBruto,RLiquido) <-
    RBruto := Taxa/100. * (Dias/30.) * Capital &
    RLiquido := 0.90 * RB .
```

(b) Em uma Cláusula Objetivo:

```
<- calculo(10000,10,30,RB,RL) &
    Valor := RL + 20000 * 1.2 .
```

Exemplo 11.2: Expressão Computável

As cláusulas a seguir permitem, dadas três listas numéricas de mesmo comprimento, gerar uma quarta lista, na qual cada elemento é a soma dos elementos correspondentes das 3 listas dadas.

```
somalistas(nil,nil,nil,nil).
somalistas(H1.T1,H2.T2,H3.T3,H4.T4) <-
    H4 := H1 + H2 + H3 &
    somalistas(T1,T2,T3,T4).
```

Exemplo 11.3: Expressão Computável

As cláusulas a seguir definem um procedimento invertível para efetuar o produto de 2 números.

```
c1: produto(X,Y,Z) :- Z := X * Y.  
c2: produto(X,Y,Z) :- X := Z / Y.  
c3: produto(X,Y,Z) :- Y := Z / X.
```

Se as variáveis X, Y e Z estiverem instanciadas, a cláusula c₁ verifica se o produto de X e Y é Z. Se as variáveis X e Y, mas não Z, estiverem instanciadas, a cláusula c₁ instancia Z com o produto de X e Y. Se alguma das variáveis X ou Y não estiver instanciada, a cláusula c₁ falha.

Se as variáveis Y e Z, mas não X, estiverem instanciadas, a cláusula c₂ instancia X com o quociente de Z por Y. Se alguma das variáveis Y ou Z não estiver instanciada, a cláusula c₂ falha.

Se as variáveis X e Z, mas não Y, estiverem instanciadas, a cláusula c₃ instancia Y com o quociente de Z por X. Se alguma das variáveis X ou Z não estiver instanciada, a cláusula c₃ falha e todo o procedimento acima falha.

Exemplo 11.4: Correção do programa PDBRAGAX

As cláusulas a seguir permitem corrigir o programa PDBRAGAX ao substituirem as cláusulas deste programa sobre os átomos "irmao" e "irma", com a codificação adicional do comando (infixo e predefinido) de comparação "==/", que retorna **SUCESSO** caso seus dois argumentos não sejam idênticos. Esta correção evita que uma consulta ao programa possa ter as variáveis X e Y instanciadas por um mesmo objeto (ou seja, evita que uma dada pessoa seja considerada irmã de si mesma).

```
irmao(X,Y) :- homem(X) &  
            pai(Pa,X) & mae(Ma,X) &  
            pai(Pa,Y) & mae(Ma,Y) &  
            X ==/ Y.  
  
irma(X,Y) :- mulher(X) &  
            pai(Pa,X) & mae(Ma,X) &  
            pai(Pa,Y) & mae(Ma,Y) &  
            X ==/ Y.
```

leia-se (primeira cláusula):

"para todo X, Y, Pa e Ma,
X é irmão de Y
se X for homem
e os pais de X forem Pa e Ma
e os pais de Y forem os mesmos Pa e Ma
e X não for idêntico a Y"

11.3 FACILIDADES EXTRALÓGICAS DE COMUNICAÇÃO

A comunicação de um programa Prolog com dispositivos de entrada e saída ocorre através de comandos extralógicos relativos a um determinado meio de entrada ou saída. Estes meios, referidos a seguir por *entrada corrente* ou *saída corrente*, são considerados neste texto como sendo o terminal, quando não explicitado de outra forma. A leitura ou impressão em outros meios implica em outras definições, cujas especificações variam nos diversos sistemas Prolog. Devido a isto, os detalhes relativos devem ser verificados no manual de utilização correspondente ao sistema em uso. De um modo geral, a definição de um meio de entrada e saída diferente do terminal ocorre através da codificação de cláusulas adicionais ou da utilização de argumentos adicionais nos comandos de leitura e impressão.

Segue uma descrição dos comandos de entrada e saída mais comuns, supondo-os dirigidos ao terminal. Observe que os comandos de comunicação não são afetados pelo procedimento de refutação Prolog (não sendo portanto afetados por retrocesso) sendo avaliados por rotinas separadas e então removidos (VM/Prolog ou MVS/Prolog permite, através da inclusão de um argumento extra, que comandos de leitura sejam passíveis de retrocesso em determinadas circunstâncias). Observe também que esta seção não incorpora comandos de comunicação com o sistema operacional, mas incorpora alguns comandos extralógicos de gerenciamento da área de trabalho Prolog.

Comando	Descrição
nl	Encerra a linha de impressão e posiciona para impressão em um novo registro da saída corrente.

tab(C)	Posiciona para impressão na coluna predefinida C da saída corrente. Em alguns sistemas Prolog, posiciona para impressão após saltar C espaços em branco no registro de saída corrente. C deve ser um número inteiro ou uma variável instanciada em um número inteiro.
read(T)	Lê o termo seguinte U definido na entrada corrente e unifica o mesmo com T. O termo U deve ser informado seguido por ponto, ou por branco e ponto se for número, que indicará o fim da entrada de dados.
readch(C)	Lê o carácter seguinte, definido na entrada corrente, e unifica o mesmo com C.
write(U)	Imprime a expressão U, seguida de ponto, na saída corrente e posiciona para impressão em um novo registro da saída corrente.
writes(U)	Imprime a expressão U na saída corrente e posiciona para impressão na coluna seguinte do mesmo registro.
writex(U)	Imprime a expressão U na saída corrente e posiciona para impressão na coluna seguinte do mesmo registro. Átomos são impressos sem aspas.
prst(X)	Imprime na saída corrente uma cadeia de caracteres que é o resultado da avaliação da expressão computável X e posiciona para impressão na coluna seguinte do mesmo registro. Constantes são impressas sem apóstrofos delimitadores.
save(X)	Armazena o estado corrente da área de trabalho Prolog no arquivo de nome X.
load(X)	Limpa a área de trabalho Prolog e carrega na mesma o arquivo de nome X, armazenado através de um prévio comando "save(X)", transformando-o na nova área de trabalho Prolog.
consult(X)	Carrega acumulativamente na área de trabalho Prolog as cláusulas Prolog armazenadas no arquivo de nome X. As

cláusulas carregadas serão prefixadas por X (nome do arquivo considerado). Cláusulas objetivo existentes no arquivo X são imediatamente executadas quando ocorre a submissão deste comando.

reconsult(X) Carrega na área de trabalho Prolog as cláusulas Prolog armazenadas no arquivo de nome X, eliminando previamente todas as cláusulas de prefixo X (nome do arquivo considerado) existentes nesta área. As cláusulas carregadas serão prefixadas por X. Cláusulas objetivo existentes no arquivo X são imediatamente executadas quando ocorre a submissão deste comando.

edconsult(X) Abre o arquivo X para edição na ATP e, em seguida, carrega na área de trabalho Prolog as cláusulas Prolog armazenadas no arquivo de nome X, eliminando previamente todas as cláusulas de prefixo X (nome do arquivo considerado) existentes nesta área. As cláusulas carregadas serão prefixadas por X. Cláusulas objetivo existentes no arquivo X são imediatamente executadas quando ocorre a submissão deste comando.

stop Encerra a ativação da máquina Prolog voltando o controle para o sistema monitor.

exit, fin Sinônimos para "stop".

restore(X) Sinônimo para "load(X)".

Quando repetido em uma mesma sessão Prolog, o comando "consult" tem efeito acumulativo, pois as cláusulas dos diversos arquivos "consultados" são acumuladas na área de trabalho Prolog. Os comandos "reconsult" e "edconsult" evitam os inconvenientes que podem advir do uso repetido do comando "consult". Observe que o comando "edconsult" é equivalente ao comando "reconsult" com a adição da facilidade que permite editar previamente o arquivo de nome X.

Sessão 11.1: Programa PDBRAGAX

Repete-se aqui a última sessão da seção 10.4 usando facilidades extralógicas de comunicação para imprimir o resultado da execução do programa PDBRAGAX, suposto carregado na área de trabalho Prolog. Observe que o usuário deve teclar ";" se desejar obter uma resposta correta adicional.

```

<- irma(X,Y) &
    writes(X) & tab(25) &
    writes(Y) & nl.
'Maria da Gloria'      'Pedro II'      ;
FALHA

<- tio(X,Y) &
    writes(X) & tab(25) &
    writes(Y) & nl.
miguel           'Pedro II'      ;
miguel           'Maria da Gloria' ;
FALHA

<- ancestral(X,'Pedro I') &
    writes(X) & nl.
'Joao VI'        ;
'Carlota Joaquina' ;
'Pedro III'       ;
'Maria I'         ;
FALHA

<- filho(X,'Carlota Joaquina') & pai(X,'Pedro II') &
    writes(X) & nl.
'Pedro I'        ;
FALHA

<- pai(X,'Pedro II') & ancestral('Pedro III',X) &
    writes(X) & nl.
'Pedro I'        ;
FALHA

<- mae(X,'Pedro I') & avoh(X,'Pedro II') &
    writes(X) & nl.
'Carlota Joaquina' ;
FALHA

```

```

<- pai(X,Y) & tio(miguel,Y) & irmao('Pedro II',Y) &
  writes(X) & tab(25) &
  writes(Y) & nl.
'Pedro I'          'Maria da Gloria' ; 
FALHA

<- pai(X,Y) & tio(Z,Y) & irmao('Pedro II',Y) &
  writes(X) & tab(25) &
  writes(Y) & tab(50) &
  writes(Z) & nl.
'Pedro I'          'Maria da Gloria'      miguel ;
FALHA

```

11.4 FACILIDADES EXTRALÓGICAS DE CONTROLE

11.4.1 Comando "cut"

O comando extralógico "cut" constitue-se no principal comando de controle ao permitir regular a execução de um programa Prolog pelo controle do procedimento de retrocesso. Segue uma descrição, alguns exemplos de utilização e uma sessão com este comando.

Comando	Descrição
	Comando extralógico sem argumentos, referido por "cut". Quando empregado em uma cláusula, comporta-se como uma fórmula atômica que é sempre satisfeita. Mas, se através do procedimento de retrocesso, a máquina Prolog tentar reavaliá-lo, a cláusula que contém o comando "cut" (chamada de <i>cláusula-pai</i>) FALHA, impedindo nova avaliação da mesma. Todas cláusulas em seqüência sobre o mesmo átomo no mesmo papel também falham nesta chamada sobre o átomo da cláusula-pai. Isto significa que o uso do comando "cut" em uma cláusula implica em um comprometimento com as unificações efetuadas na mesma até a fórmula atômica imediatamente à esquerda do comando "cut" (procedimento de poda heurística).

A utilização do comando "cut" em um programa permite ao programador alterar o mecanismo de controle do Prolog, aumentando a eficiência do programa pela supressão ("corte") parcial do procedimento de retrocesso.

Exemplo 11.5: Comando "cut"

(a) $c_1: p \leftarrow a \& b \& c \& d \& e \& / \& f.$

Se, selecionada para unificação, "f" falhar, não ocorre retrocesso sobre "e" e a cláusula-pai c_1 falha.

(b) $c_1: p \leftarrow a \& b \& / \& c \& d \& e \& / \& f.$

O mecanismo de retrocesso está livre para se mover entre "c" e "e" (ou seja, entre "cuts").

(c) $c_1: p \leftarrow a \& b \& c .$
 $c_2: b \leftarrow d \& e \& / \& f .$

Se, selecionada para unificação, "f" falhar, a cláusula-pai c_2 falha e o mecanismo de retrocesso do Prolog volta a selecionar "a" em c_1 em busca de uma unificação alternativa (dado que c_2 foi "chamada" por "b" de c_1).

(d) $c_1: p \leftarrow a .$
 $c_2: p \leftarrow b \& / \& c .$
 $c_3: p \leftarrow d \& e .$
 $c_4: p \leftarrow f .$
 $c_5: p \leftarrow g \& h .$

Com a codificação do comando "cut" na cláusula c_2 , tem-se:

- Se "b" for selecionada para unificação e for satisfeita e, em seguida, "c" falhar, não ocorre retrocesso sobre "b". A cláusula-pai c_2 falha e as cláusulas c_3 , c_4 e c_5 não mais serão chamadas pelo procedimento de refutação, nesta chamada de "p".
- Se "b" for selecionada para unificação e falhar, as cláusulas c_3 , c_4 e c_5 continuarão disponíveis para seleção pelo procedimento de refutação, nesta chamada de "p".

Exemplo 11.6: Comando "cut"

```

homem('Pedro III').
homem('Joao VI').
homem('Pedro I').
homem(miguel).
homem('Pedro II').
pai('Pedro III','Joao VI').
pai('Joao VI','Pedro I').
pai('Joao VI',miguel).
primogenito(X,Y) :- homem(X) & pai(Y,X) & !.

```

A codificação do comando "cut" na última cláusula garante obter apenas uma resposta ($X = 'Pedro I'$) para a consulta " $<- \text{primogenito}(X, 'Joao VI').$ ". Sem esta codificação duas respostas ($X = 'Pedro I'$ e $X = \text{miguel}$) poderiam ser obtidas para a mesma consulta, que não faz sentido face ao conceito de primogênito. Observe que, neste caso, a ordem das cláusulas unitárias sobre o átomo "pai" é importante para se obter corretamente o primogênito.

Exemplo 11.7: Operação "membro" definida com "cut"

```

c1: membro(X,X.* ) :- !.
c2: membro(X,*.Y) :- membro(X,Y).

```

Com a codificação do comando "cut" na cláusula c_1 , tem-se:

- uma cláusula objetivo definida para extrair os elementos de uma lista permitirá extrair apenas o primeiro elemento. Isto ocorrerá pois a cabeça da cláusula c_1 será escolhida com sucesso para unificação e o comando "cut" inibirá que uma solicitação de retrocesso, em busca de um segundo elemento da lista, escolha a cláusula c_2 .
- uma cláusula objetivo, consultando se um termo dado é elemento de uma lista dada, falha após atingir o primeiro elemento que satisfaz a consulta. Isto impossibilita atender a uma solicitação de resposta alternativa para verificar se existe mais de um tal elemento na lista.

Exemplo 11.8: Operação "concat" definida com "cut"

```
c1: concat(nil,L,L) <- /.
c2: concat(X,L1,L2,X,L3) <- concat(L1,L2,L3).
```

Com a codificação do comando "cut" na cláusula c_1 , inibe-se a possibilidade de decomposição de uma lista dada em todas as sublistas possíveis. A única decomposição possível, definida pela cláusula c_1 , será nas sublistas `nil` e lista dada.

Exemplo 11.9: Procedimento Prolog para cálculo de fatorial

```
c1: factorial(0,1) <- /.
c2: factorial(N,FATN) <- diff(N,1,N1) &
               factorial(N1,FATN1) &
               prod(FATN1,N,FATN).
```

A codificação do comando "cut" na cláusula c_1 evita que um pedido de resposta alternativa, improcedente neste caso, seja aceita e provoque um laço infinito.

Sessão 11.2: Comando "cut"

Considerando os procedimentos "membro", "concat" e "factorial", carregados na área de trabalho Prolog, como definidos nesta seção (ou seja, com a codificação do comando "cut" em suas cláusulas " c_1 "), tem-se:

```
<-membro(X, a. b. c. d. nil).
SUCESSO: X= a ;
FALHA

<-concat(X, Y, a. b. c. d. nil).
SUCESSO: X= nil
          Y= (a.b.c.d.nil) ;
FALHA

<-factorial(10,X) &
    prst('0 fatorial de 10 e'' ') &
    write(X).
0 fatorial de 10 e' 3628800 .
SUCESSO
```

```
<-fatorial(5,X).
SUCESSO: X= 120 ;
FALHA
```

Compare o resultado obtido para as duas primeiras cláusulas objetivo com os resultados obtidos com cláusulas objetivo idênticas nas sessões finais da seção 10.5.

O uso do comando "cut" pode alterar radicalmente a seqüência de controle de um programa Prolog, sendo, por este motivo, equivalente ao uso de comandos GOTO em programas escritos em linguagens procedimentais. Como GOTO, o "cut" é um comando poderoso mas indisciplinado. Portanto, a boa técnica de programação em Prolog indica que a utilização do comando "cut" ocorra de modo disciplinado e seja limitada a situações em que o seu efeito seja de compreensão imediata, como, por exemplo, na definição do comando "se-então-senão" (ver seção 11.7). A utilização disciplinada pode ocorrer também através da definição de um conjunto de rotinas de serviço que incluem o comando "cut" mas que permitam manter a interpretação declarativa de um programa Prolog (como exemplo, ver a implementação do conceito de negação por falha finita na seção 11.7).

Exemplo 11.10:

A seqüência de cláusulas mutuamente exclusivas:

```
b0 <- b1 & / & b2.
b0 <- b3.
```

é equivalente a: "se b_1 , então b_0 é b_1 e b_2 , senão, b_0 é b_3 "

Exemplo 11.11:

A seqüência de cláusulas a seguir:

```
c1: maximo(X,Y,X) <- ge(X,Y).
c2: maximo(X,Y,Y) <- lt(X,Y).
```

com leitura:

c₁: o máximo entre X e Y é X se X for maior ou igual a Y
 c₂: o máximo entre X e Y é Y se X for menor que Y

é equivalente a:

```
c1: maximo(X,Y,X) :- ge(X,Y) & !.
c2: maximo(X,Y,Y).
```

com leitura:

c₁: o máximo entre X e Y é X se X for maior ou igual a Y

c₂: o máximo entre X e Y é Y, caso contrário

O segundo programa é mais eficiente pois evita a computação redundante de "lt(X,Y)". Mas ele é menos claro na medida em que as cláusulas não podem mais serem corretamente compreendidas de modo individual, uma característica da interpretação semântica declarativa do Prolog.

Em resumo, o comando extralógico "cut" permite controlar o retrocesso através de:

- comprometimento com as unificações efetuadas em uma dada cláusula nas fórmulas atômicas que antecedem o comando "cut";
- impedimento de chamada para unificação de cláusulas sobre um dado átomo no mesmo papel, precedidas por uma cláusula sobre o mesmo átomo no mesmo papel que tenha falhado por ter ocorrido retrocesso sobre o comando "cut".

Algumas das formas usuais de emprego do comando "cut" são para:

- aumentar a eficiência de um programa Prolog;
- evitar a geração de respostas alternativas quando não forem significativas;
- indicar que a cláusula-pai (cláusula que contém o comando "cut") é insatisfável (ver, a seguir, combinação "cut/fail");
- estabelecer que a relação sendo computada é uma função no sentido matemático: ela fornece uma única resposta, não havendo sentido uma solicitação de retrocesso em busca de uma resposta alternativa (ver Exemplo 11.9).
- classificar objetos em categorias mutuamente exclusivas: uma vez obtida a classificação, não faz sentido que o mecanismo de retrocesso tente, pela ocorrência de alguma FALHA posterior, classificar o mesmo objeto em outra categoria.

11.4.2 Outros Comandos Extralógicos de Controle

Uma série de outros comandos considerados como extralógicos em alguns textos, na verdade, podem ser implementados por definição na própria linguagem Prolog. Apresentaremos a seguir um resumo, exemplos de utilização e duas sessões com alguns destes comandos.

Comando	Descrição
fail	Comando extralógico sem argumentos que sempre FALHA, forçando retrocesso imediato em busca de uma resposta alternativa.
true	Comando extralógico sem argumentos que é sempre satisfeito (oposto de "fail") mas FALHA em retrocesso.
repeat	Comando extralógico sem argumentos que é sempre satisfeito, mesmo em retrocesso.

O comando extralógico "fail" é tipicamente usado para solicitar respostas corretas alternativas ou provocar um laço que permita executar alguma seqüência repetitivamente.

O comando extralógico "repeat" quando selecionado para unificação em um processo de retrocesso gera outro ramo alternativo de avaliação: as fórmulas atômicas à direita do comando "repeat" na mesma cláusula são avaliadas como se não houvesse ocorrido retrocesso. Este comando apresenta a propriedade de não-determinismo, podendo ser usado para gerar uma seqüência infinita de respostas corretas.

Exemplo 11.12: Definição do comando "fail"

Este comando é definido pela inexistência, na área de trabalho Prolog, de qualquer cláusula cuja cabeça seja "fail()" ou "fail". Observe que qualquer fórmula atômica Prolog da forma "p()" com esta propriedade pode desempenhar o mesmo papel que o comando "fail".

Exemplo 11.13: Definição do comando "true"

Este comando pode ser definido adicionando-se a cláusula unitária Prolog "true()." ou "true." na área de trabalho Prolog. Observe que qualquer fórmula atômica Prolog da forma "p()" tal que "p()." seja a única cláusula na área de trabalho Prolog sobre "p" pode desempenhar o mesmo papel que o comando "true".

Exemplo 11.14: Definição do comando "repeat"

Este comando pode ser definido adicionando-se à área de trabalho Prolog as cláusulas:

```
c1: repeat.  
c2: repeat <- repeat.
```

Quando o comando "repeat" é avaliado pela primeira vez em um programa Prolog, a cláusula c_1 é ativada e satisfeita, permitindo a avaliação da fórmula atômica ou comando seguinte da cláusula objetivo corrente. Quando o comando "repeat" é avaliado em retrocesso em um programa Prolog, a cláusula c_2 é ativada e, recursivamente, ativa a cláusula c_1 que é satisfeita, permitindo a avaliação da fórmula atômica ou comando seguinte da cláusula objetivo corrente.

Exemplo 11.15: Comando "fail"

```
<-tio(X,Y) & prst(X) & tab(15) & prst(Y) & nl & fail.
```

Esta cláusula objetivo avalia inicialmente "tio(X,Y)" sobre um programa Prolog (como PDBRAGAX). Se esta avaliação for satisfeita, esta cláusula permite imprimir X (instanciado com o nome de tio) na coluna 1, Y (instanciado com o nome de sobrinho ou sobrinha) na coluna 15, saltar de linha e repetir este procedimento pois o comando "fail" falha provocando retrocesso. A cláusula falha quando esgotar todas as alternativas de respostas corretas associadas à fórmula atômica "tio(X,Y)", que representa a única possibilidade de retrocesso na cláusula objetivo acima, dado que os comandos "writes", "tab" e "nl" não são passíveis de retrocesso.

Sessão 11.3: Comando "fail"

Repete-se aqui a sessão relativa aos comandos de comunicação. Em adição às facilidades extralógicas de comunicação (com o comando "prst" substituindo o comando "writes" para, neste caso, permitir a impressão de constantes Prolog sem apóstrofos delimitadores), utiliza-se o comando extralógico de controle "fail", para executar o programa PDBRAGAX, suposto carregado na área de trabalho Prolog. O comando extralógico "fail" foi utilizado com a finalidade de obter automaticamente todas as respostas corretas para cada cláusula objetivo Prolog, evitando assim que o usuário tecle ";" para obter respostas adicionais.

```
<- irma(X,Y) &
    prst(X) & tab(25) &
    prst(Y) & nl & fail.
```

Maria da Gloria Pedro II
FALHA

```
<- tio(X,Y) &
    prst(X) & tab(25) &
    prst(Y) & nl & fail.
```

miguel Pedro II
miguel Maria da Gloria
FALHA

```
<- ancestral(X,'Pedro I') &
    prst(X) & nl & fail.
```

Joao VI
Carlota Joaquina
Pedro III
Maria I
FALHA

```
<- filho(X,'Carlota Joaquina') & pai(X,'Pedro II') &
    prst(X) & nl & fail.
```

Pedro I
FALHA

```
<- pai(X,'Pedro II') & ancestral('Pedro III',X) &
    prst(X) & nl & fail.
```

Pedro I
FALHA

```
<- mae(X,'Pedro I') & avoh(X,'Pedro II') &
   prst(X) & nl & fail.
```

Carlota Joaquina

FALHA

```
<- pai(X,Y) & tio(miguel,Y) & irmao('Pedro II',Y) &
   prst(X) & tab(25) &
   prst(Y) & nl & fail.
```

Pedro I

Maria da Gloria

FALHA

```
<- pai(X,Y) & tio(Z,Y) & irmao('Pedro II',Y) &
   prst(X) & tab(25) &
   prst(Y) & tab(50) &
   prst(Z) & nl and fail.
```

Pedro I

Maria da Gloria

miguel

FALHA

Sessão 11.4: Comandos "true" e "repeat"

Seja o seguinte programa Prolog carregado na ATP:

```
chama(a,b).
chama(a,c).
depende1(X,Y) <- chama(X,Y).
depende2(X,Y) <- true & chama(X,Y).
depende3(X,Y) <- repeat & chama(X,Y).
```

A sessão apresentada a seguir evidencia as diferenças operacionais existentes entre os comandos "true" e "repeat":

<-depende1(a,X).

SUCESSO: X= b ;
X= c ;

FALHA

<-depende2(a,X).

SUCESSO: X= b ;
X= c ;

FALHA

<-depende3(a,X).

SUCESSO: X= b ;

```
X= c ;  
X= b ;  
X= c ;  
X= b ;  
...
```

A combinação dos mecanismos extralógicos "cut" e "fail" (referida a seguir por combinação "cut/fail") em uma cláusula provoca, quando ativada, a FALHA da cláusula. A falha ocorre pois o comando "fail" provoca retrocesso sobre o comando "cut" que não pode ser reexecutado. Devido a isto, a cláusula falha e todas cláusulas posteriores sobre o mesmo símbolo predicativo não mais serão selecionadas pelo procedimento de refutação nesta chamada para unificação da cabeça da cláusula pai.

Exemplo 11.16: Combinação "cut/fail"

```
c1: p <- a.  
c2: p <- b & c & / & fail.  
c3: p <- d & e.  
c4: p <- f.  
c5: p <- g & h.
```

Se selecionadas para unificação "b" e "c" forem satisfeitas, a cláusula-pai c_2 falha e as cláusulas c_3 , c_4 e c_5 não mais serão selecionadas pelo procedimento de refutação nesta chamada de "p".

Exemplo 11.17: Combinação "cut/fail"

As cláusulas a seguir declaram e definem um operador infixo, denotado por "nao_unifica", que permite verificar se duas estruturas arbitrárias não são unificáveis:

```
op(nao_unifica,lr,50).  
X nao_unifica Y <- / & fail.  
* nao_unifica *.
```

Nota: Este operador é predefinido nos dialetos Prolog existentes (através do símbolo " \neq " ou " $=/$ ", em geral).

Exemplo 11.18: Combinações "repeat/fail" e "cut/fail"

A cláusula abaixo permite ler números e imprimir, para cada número lido, o triplo do seu valor. O comando "fail" força retrocesso automático sobre o comando "repeat", único comando passível de retrocesso na cláusula, que então evita repetir as mensagens iniciais do programa mas permite repetir a operação de leitura, teste, cálculo e impressão. Uma informação de "fim." permite encerrar a execução da cláusula através da combinação "cut/fail".

```
cubo <-
    nl &
    prst('Calculo do cubo de um numero')      &
    nl &
    prst('Para encerrar tecle: fim.')          &
    nl &
    prst('Entre um numero ')                   &
    prst('seguido de branco e ponto: ')        &
    nl &
    repeat &
        (read == fim & / & fail | Y := X * X * X) &
        prst('O cubo de ' || X || ' e' || Y)       &
        prst(' - entre outro numero ou: fim.')     &
    nl & fail.
```

Nota: o operador "|" designa o conectivo lógico "ou" e será discutido na seção 11.7.

11.5 FACILIDADES EXTRALÓGICAS DE PROCESSAMENTO DE CLÁUSULAS

Os comandos apresentados nesta seção permitem processar, estática ou dinamicamente, cláusulas existentes na área de trabalho Prolog. Recorde que uma cláusula definida é sobre um átomo "p", no papel de um símbolo predicativo *n*-ário, se e somente se a cabeça da cláusula for da forma "*p(t₁,...,t_n)*". Neste caso diz-se que "p" é o átomo da cláusula definida.

Comando	Descrição
call(C)	Avalia uma cláusula C, retornando SUCESSO se a avaliação de C for satisfeita.

addax(C)	Adiciona a cláusula C, na área de trabalho Prolog, após todas as cláusulas sobre o mesmo átomo no mesmo papel que o da cláusula C. Este comando não é passível de retrocesso.
addax(C,N)	Adiciona a cláusula C, na área de trabalho Prolog, na N-ésima posição entre as cláusulas sobre o mesmo átomo no mesmo papel que o da cláusula C. Este comando não é passível de retrocesso.
delax(H)	Remove, da área de trabalho Prolog, a primeira cláusula definida cuja cabeça unifica com a cláusula unitária H. Este comando não é passível de retrocesso.
delax(H,N)	Remove, da área de trabalho Prolog, a N-ésima cláusula definida cuja cabeça unifica com a cláusula unitária H. Este comando não é passível de retrocesso.
ax(H,R)	Recupera em R, da área de trabalho Prolog, a primeira cláusula definida cuja cabeça unifica com a cláusula unitária H.
ax(H,R,N)	Recupera em R, da área de trabalho Prolog, a N-ésima cláusula definida cuja cabeça unifica com a cláusula unitária H.
axn(P,A,R)	Recupera em R, da área de trabalho Prolog, a primeira cláusula sobre o átomo P no papel de um símbolo predicativo de aridade A.
axn(P,A,R,N)	Recupera em R, da área de trabalho Prolog, a N-ésima cláusula sobre o átomo P no papel de um símbolo predicativo de aridade A.
add(C)	Sinônimo para "addax(C,1)".
asserta(C)	Sinônimo para "addax(C,1)".
assertz(C)	Sinônimo para "addax(C)".

- retract(H)** Sinônimo para "delax(H)", porém passível de retrocesso.
- clause(H,R)** Sinônimo para "ax(H,R)".

Exemplo 11.19: Programa para listar cláusulas

O programa apresentado a seguir, suposto armazenado em um arquivo de nome "listac", permite listar cláusulas sobre átomos da ATP. Observe que o argumento do comando "prst" é a concatenação de uma constante Prolog com uma variável Prolog.

```
listac(X) :- prst('lista das clausulas sobre: '|| X) &
            nl & listac1(X).
listac1(X) :- nl & axn(X, *, R) & writes(R) & nl & fail.
```

Sessão 11.5: Processamento de cláusulas

```
<-addax(homem('Pedro I')) &
  addax(pai('Joao VI', 'Pedro I')) &
  addax(filho(X,Y) <- homem(X) & pai(Y,X)).
```

SUCESSO

```
<-filho(X,Y).
```

SUCESSO: X= 'Pedro I'
Y= 'Joao VI'

```
<-delax(filho(*,*)).
```

SUCESSO

```
<-filho(X,Y).
```

FALHA

```
<-pai(X,Y) & delax(pai(*,*)) & delax(homem(*)).
```

SUCESSO: X= 'Joao VI'
Y= 'Pedro I'

```
<-pai(X,Y).
```

FALHA

```
<-homem(X).
```

FALHA

Sessão 11.6: Processamento de cláusulas

```
<-ax(op(*,*,*),R) & writes(R) & nl & fail.  
/*
```

segue lista de todas as cláusulas da área de trabalho
que tenham aridade 3 e sejam sobre op
*/

```
<-ax(op(*,*,*),R,5) & writes(R) & nl & fail.  
op(=:=,lr,50)
```

FALHA

```
<-axn(op,*,R) & writes(R) & nl & fail.  
/*
```

segue lista de todas as cláusulas da área de trabalho
que tenham aridade qualquer e sejam sobre op
*/

```
<-axn(op,*,R,6) & writes(R) & nl & fail.  
op(=*=,lr,50)
```

FALHA

```
<-consult(listac).
```

SUCESSO

```
/*
```

carregado, a partir do arquivo de nome "listac",
o programa Prolog definido no exemplo anterior
*/

```
<-listac(op).
```

Lista das cláusulas sobre: op

```
/*
```

segue lista de todas as cláusulas da área de trabalho
que tenha aridade qualquer e sejam sobre op

```
*/
```

```

<-consult(pdbragax). /* carrega programa PDBRAGAX */
SUCESSO

<-listac(filho).
lista das clausulas sobre: filho

filho(V1,V2) <- homem(V1) & pai(V2,V1)
filho(V1,V2) <- homem(V1) & mae(V2,V1)
FALHA

```

Exemplo 11.20: Aprendizado de novos fatos

O programa codificado a seguir (Bharath [1986a]) representa uma base de fatos de estados e correspondentes capitais com capacidade de aprendizado de novos pares estado-capital interativamente. Observe que o comando extralógico "nonvar(X)", uma facilidade de verificação de tipos de dados, é satisfeito se X for uma variável instanciada ou X não for uma variável. Em outros dialetos (como o VM/Prolog ou MVS/Prolog) sua codificação pode ser diferente e o manual correspondente deverá ser consultado para acomodar esta possível diferença sintática.

```

estado_e_capital(rj,rio).
estado_e_capital(rn,nat).
estado_e_capital(rs,poa).
estado_e_capital(sp,sao).

estado_e_capital(Estado,Cidade) <-
    nonvar(Estado) &
    writex("Nao conheco a capital de ") &
    writex(Estado) &
    nl &
    writex("Favor me informar ") &
    nl &
    writex("(via sigla de 3 caracteres e ponto)")&
    nl &
    read(Cidade) &
    addax(estado_e_capital(Estado,Cidade),1).

```

```

estado_e_capital(Estado,Cidade) <-
    nonvar(Cidade) &
    writex("Nao conheco ") &
    writex("o estado cuja capital eh ") &
    writex(Cidade) &
    nl &
    writex("Favor me informar ") &
    nl &
    writex("(via sigla de 2 caracteres e ponto)")&
    nl &
    read(Estado) &
    addax(estado_e_capital(Estado,Cidade),1).

```

11.6 FACILIDADES EXTRALÓGICAS DE DEPURAÇÃO

Três comandos básicos, comuns aos sistemas Prolog existentes, permitem a depuração de cláusulas específicas em um programa Prolog. Esta seção descreve e apresenta exemplos destes comandos, conhecidos por "break", "spy" e "trace".

Observe que os comandos desta seção têm como argumento uma expressão do tipo "*p(*,...,*)*" onde "*p*" é um átomo. Esta expressão identifica a ocorrência de "*p*" como símbolo predutivo *n*-ário, onde *n* é definido pelo número de asteriscos da expressão. Esta identificação é necessária quando da submissão dos comandos apresentados, pois a área de trabalho Prolog poderá conter cláusulas sobre átomos idênticos em diferentes papéis. Observe também que, nas definições a seguir, o parâmetro "*T*", usado como argumento dos comandos de depuração, é uma chave que permite ativar (se não utilizada ou se *T* = *on*) ou desativar (se *T* = *off*) os comandos "break", "spy" e "trace".

Comando	Descrição
<code>break(p(*,...,*),T)</code>	Ativa o procedimento que permite parar a execução do programa sempre que uma fórmula atômica da forma $p(t_1, \dots, t_n)$, de mesma aridade que $p(*, \dots, *)$, for selecionada para unificação. Apresenta, no terminal, a cláusula Prolog escolhida para unificação permitindo interação com o usuário através de comandos predefinidos. Ou seja, o usuário pode, após análise da situação,

cancelar ou continuar a execução do programa, entre outras opções.

spy(p(*,...,*),T)

Ativa o procedimento que permite coletar, relativamente às fórmulas atômicas da forma $p(t_1, \dots, t_n)$, de mesma aridade que $p(*, \dots, *)$, informações sobre:

- aridade
- número de seleções para unificação
- número de seleções, por retrocesso, para unificação
- número de unificações bem sucedidas
- número de unificações mal sucedidas

trace(p(*,...,*),T)

Ativa o procedimento que permite rastreiar todas as seleções para unificação de fórmulas atômicas da forma $p(t_1, \dots, t_n)$, de mesma aridade que $p(*, \dots, *)$, durante a execução do programa.

As informações coletadas por "spy" são automaticamente impressas no final da execução do programa ou, dependendo do dialeto Prolog considerado, são armazenadas para impressão por solicitação através de um comando específico ("<-spy_display.", neste texto).

Se o rastreamento, definido por um comando "trace", estiver ativo para $p(*, \dots, *)$, sempre que uma fórmula atômica da forma $p(t_1, \dots, t_n)$, de mesma aridade que $p(*, \dots, *)$, for selecionada para unificação, a máquina Prolog informa ao usuário o tipo de seleção efetuada, que pode ser:

m: call --> p(t₁,...,t_n) Indica que a fórmula atômica $p(t_1, \dots, t_n)$ foi selecionada para unificação sobre alguma cláusula do programa Prolog.

m: redo --> p(t₁,...,t_n) Indica que a fórmula atômica $p(t_1, \dots, t_n)$ foi selecionada pelo mecanismo de retrocesso para unificação sobre alguma cláusula do programa Prolog.

m: exit --> p(u₁,...,u_n) Indica que p(u₁,...,u_n) é o resultado da unificação a que foi submetido a fórmula atômica p(t₁,...,t_n) selecionada pelo "call" ou "redo" correspondente (de mesmo "m").

m: fail --> p(t₁,...,t_n) Indica que a fórmula atômica p(t₁,...,t_n) não é unificável com a cabeça da cláusula do programa Prolog selecionada pelo "call" ou "redo" correspondente (de mesmo "m").

Observe que:

- o nível m de "call" e "exit" deve ser o mesmo;
- o nível m de "redo" e "exit" deve ser o mesmo;
- as comparações entre os termos t_i em "call" ou "redo" e u_i no "exit" correspondente permitem identificar as substituições efetuadas.

Resumindo, para o tipo de seleção efetuada, tem-se:

call	: indica seleção para unificação
redo	: indica seleção, por retrocesso, para unificação
exit	: indica unificação bem sucedida
fail	: indica unificação mal sucedida
m	: indica o número de ocorrência de um call, exit, redo ou fail

Sessão 11.7: Comando "spy"

Considere a operação "concat", carregada na área de trabalho Prolog, codificada através das cláusulas:

```

concat(nil,L,L).
concat(X,L1,L2,X,L3) :- concat(L1,L2,L3).

<-spy(concat(*,*,*)).          /* ativa spy           */
SUCESSO

<-concat(a.b.nil, c.d.nil, X). /* clausula objetivo */
SUCESSO:   X= (a.b.c.d.nil)

```

```

<-spy_display.          /* resultado do spy */

prefix predicate arity call exit redo fail

nil concat 3 3 3 0 0
SUCESSO

<-spy(concat(*,*),off). /* desativa spy */
SUCESSO

```

Sessão 11.8: Comando "spy"

Considere agora a operação "membro", definida em função de "concat" (vide a seguir), carregada na área de trabalho Prolog e codificada através da cláusula:

```

membro(X,L) :- concat(*,X,*),L.

<-spy(membro(*,*)).      /* ativa spy */
SUCESSO

<-spy(concat(*,*,*)).    /* ativa spy */
SUCESSO

<-membro(c, a. b. c. d. nil). /* clausula objetivo */
SUCESSO

<-spy_display.           /* resultado do spy */

prefix predicate arity call exit redo fail

nil concat 3 3 3 0 0
nil membro 2 1 1 0 0
SUCESSO

<-spy(membro(*,*),off). /* desativa spy */
SUCESSO

<-spy(concat(*,*,*),off). /* desativa spy */
SUCESSO

```

Sessão 11.9: Comando "trace"

Considere novamente a operação "concat" carregada na área de trabalho Prolog:

```

<-trace(concat(*,*,*)).          /* ativa trace      */
SUCESSO

<-concat(a.b.nil, c.d.nil, X).    /* clausula objetivo */

1: call ==> concat(a.b.nil, c.d.nil, V1).
2: call ==> concat(b.nil, c.d.nil, V1).
3: call ==> concat(nil, c.d.nil, V1).
3: exit ==> concat(nil, c.d.nil, c.d.nil).
2: exit ==> concat(b.nil, c.d.nil, b.c.d.nil).
1: exit ==> concat(a.b.nil, c.d.nil, a.b.c.d.nil).
SUCESSO:  X= (a.b.c.d.nil)

<-trace(concat(*,*,*),off).       /* desativa trace   */
SUCESSO

```

Sessão 11.10: Comando "trace"

Considere o programa PDIPA, cuja codificação simplificada é relembrada a seguir, carregado na área de trabalho Prolog. A ativação do comando de depuração "trace" para os símbolos predicativos "chama", "usa" e "depende" permite acompanhar as derivações efetuadas pela máquina Prolog (compare com a árvore de refutação apresentada em exemplo da seção 10.3):

```

/*
Programa PDIPA (simplificado)
*/
chama(a,b).
usa(b,e).

depende(X,Y) <- chama(X,Y).
depende(X,Y) <- usa(X,Y).
depende(X,Y) <- chama(X,Z) & depende(Z,Y).

```

```

<-trace(chama(*,*),on).          /* ativa trace */
SUCESSO

<-trace(usa(*,*),on).           /* ativa trace */
SUCESSO

<-trace(depende(*,*),on).       /* ativa trace */
SUCESSO

<-depende(W,e).                /* clausula objetivo */
/* compare secao 10.3 */

1: call ==> depende(V1,e).
2: call ==> chama(V1,e).
2: fail ==> chama(V1,e).
2: call ==> usa(V1,e).
2: exit ==> usa(b,e).          /* unifica V1 com b */
1: exit ==> depende(b,e).
SUCESSO

;

/* solicita resposta */
/* alternativa */

1: redo ==> depende(b,e).
2: redo ==> usa(b,e).
2: fail ==> usa(V1,e).
2: call ==> chama(V1,V2).
2: exit ==> chama(a,b).
2: call ==> depende(b,e).
3: call ==> chama(b,e).
3: fail ==> chama(b,e).
3: call ==> usa(b,e).
3: exit ==> usa(b,e).
2: exit ==> depende(b,e).
1: exit ==> depende(a,e).
SUCESSO

```

```

<-trace(chama(*,*),off).          /* desativa trace */
SUCESSO

<-trace(usa(*,*),off).          /* desativa trace */
SUCESSO

<-trace(depende(*,*),off).      /* desativa trace */
SUCESSO

```

Sessão 11.11: Comando "trace"

O programa "inteiro", apresentado a seguir, verifica se um número dado é inteiro e gera também números inteiros consecutivos. As ativações do comando "trace", a seguir, permitem entender as refutações efetuadas nestas duas situações.

```

<-consult(lista).
SUCESSO

```

```

<-consult(inteiro).
SUCESSO

```

```

<-listac(inteiro).
Lista das clausulas sobre inteiro:

```

```

inteiro(0).
inteiro(N) <- inteiro(N1) & sum(1,N1,N).
FALHA

```

```

<-inteiro(5).
SUCESSO

```

```

<-inteiro(X).
SUCESSO: X= 0 ;
SUCESSO: X= 1 ;
SUCESSO: X= 2 ;
SUCESSO: X= 3

```

```

<-trace(inteiro(*),on).          /* ativa trace */
SUCESSO

```

```

<-trace(sum(*,*,*),on).          /* ativa trace */
SUCESSO

```

```

<-inteiro(2).          /* clausula objetivo */

1: call ==> inteiro(2).
2: call ==> inteiro(V1).      /* unifica V1 com 0 */
2: exit ==> inteiro(0).
2: call ==> sum(1,0,2).
2: fail ==> sum(1,0,2).
2: redo ==> inteiro(0).
3: call ==> inteiro(V1).
3: exit ==> inteiro(0).      /* unifica V1 com 0 */
3: call ==> sum(1,0,V1).
3: exit ==> sum(1,0,1).
2: exit ==> inteiro(1).
2: call ==> sum(1,1,2).
2: exit ==> sum(1,1,2).
1: exit ==> inteiro(2).
SUCESSO

```

```

<-inteiro(X).          /* clausula objetivo */

1: call ==> inteiro(V1).
1: exit ==> inteiro(0).
SUCESSO: X= 0 ;

1: redo ==> inteiro(0).
2: call ==> inteiro(V1).      /* unifica V1 com 0 */
2: exit ==> inteiro(0).
2: call ==> sum(1,0,V1).
2: exit ==> sum(1,0,1).      /* unifica V1 com 1 */
1: exit ==> inteiro(1).
SUCESSO: X= 1 ;

```

```

1: redo ==> inteiro(1).
2: redo ==> sum(1,0,1).
2: fail ==> sum(1,0,V1).
2: redo ==> inteiro(0).          /* unifica V1 com 0 */
3: call ==> inteiro(V1).
3: exit ==> inteiro(0).
3: call ==> sum(1,0,V1).
3: exit ==> sum(1,0,1).          /* unifica V1 com 1 */
2: exit ==> inteiro(1).
2: call ==> sum(1,1,V1).          /* unifica V1 com 2 */
2: exit ==> sum(1,1,2).
1: exit ==> inteiro(2).

SUCESSO:   X= 2

<-trace(inteiro(*),off).          /* desativa trace */
SUCESSO

<-trace(sum(*,*,*),off).          /* desativa trace */
SUCESSO

```

11.7 PROGRAMAÇÃO NA METALINGUAGEM

Esta seção introduz o conceito de metaprograma, uma extensão bastante poderosa da linguagem Prolog que essencialmente permite definir expressões Prolog para gerar cláusulas. Esta extensão, em conjunto com as demais facilidades extralógicas, facilita a definição de programas aplicativos em Prolog e é fundamental para a implementação do conceito de negação por falha finita e para a simulação de outros conectivos e dos quantificadores.

Um primeiro exemplo ilustra os princípios básicos desta extensão.

Exemplo 11.21:

Considere o problema de introduzir o conectivo "ou" em Prolog, ou seja, de permitir disjunções de conjunções de fórmulas atômicas Prolog no corpo das cláusulas.

Para resolver este problema, observe inicialmente que o comportamento de "X ou Y" pode ser simulado pela tentativa de invocar X e, em caso de falha, invocar Y. Assim, uma chamada a "X ou Y" terá sucesso se a chamada a X tiver sucesso ou a chamada a

X fracassar finitamente e a chamada a Y tiver sucesso. As cláusulas abaixo simularão então o conectivo binário "ou":

$c_1: \text{ou}(X, Y) \leftarrow X.$
 $c_2: \text{ou}(X, Y) \leftarrow Y.$

Por exemplo, a submissão da consulta:

$c_3: \leftarrow \text{ou}(\text{p}(a) \& \text{q}(b), r(Z)).$

gerará inicialmente a cláusula:

$c_4: \leftarrow \text{p}(a) \& \text{q}(b).$

obtida de c_3 e c_1 com a substituição de X por " $\text{p}(a) \& \text{q}(b)$ " e Y por " $r(Z)$ ". Se a execução de c_4 falhar finitamente, o passo de retrocesso levará à geração de:

$c_5: \leftarrow r(Z).$

obtida de c_3 e c_2 novamente com a substituição de X por " $\text{p}(a) \& \text{q}(b)$ " e Y por " $r(Z)$ ". Note porém que, se c_4 divergir, c_5 nunca será gerada, mesmo que viesse a ter sucesso. Logo, c_1 e c_2 simulam apenas parcialmente a disjunção.

Para completar este exemplo, observe que a sintaxe da simulação do conectivo "ou" pode ser melhorada definindo-o como operador infixo da seguinte forma:

$c_0: \text{op}(\text{ou}, \text{rl}, 20).$
 $c_1: X \text{ ou } Y \leftarrow X.$
 $c_2: X \text{ ou } Y \leftarrow Y.$

Nota: Este operador é predefinido nos dialetos Prolog existentes (através do símbolo ";" ou "|", em geral).

Para que a solução apontada no exemplo seja possível, é necessário porém:

- permitir, no corpo das cláusulas, o uso de variáveis onde anteriormente ocorriam apenas fórmulas atómicas (como por exemplo o uso de X no corpo de c_1);

- permitir o uso de conjunções de fórmulas atômicas no lugar anteriormente reservado apenas para termos (como por exemplo o uso de "p(a) & q(b)" em c_3);
- estender o processo de unificação de tal forma que uma variável possa ser substituída por uma conjunção de fórmulas atômicas Prolog (como por exemplo a substituição de X por "p(a) & q(b)" na geração de c_4).

De uma forma mais rigorosa, programação na metalinguagem exige inicialmente uma redefinição da sintaxe de Prolog, acomodada através das seguintes definições:

Definição 11.1:

O conjunto das *expressões Prolog*, ou simplesmente das *expressões*, é o menor conjunto satisfazendo às seguintes condições:

- (i) toda variável Prolog é uma expressão Prolog;
- (ii) toda constante Prolog é uma expressão Prolog;
- (iii) se e_1, \dots, e_n são expressões Prolog e p é um átomo, então $p(e_1, \dots, e_n)$ é uma expressão Prolog, chamada de *expressão atômica*;
- (iv) se e_1, \dots, e_n são expressões atômicas ou variáveis Prolog então " $e_1 \& \dots \& e_n$ " é uma expressão Prolog, chamada de *conjunção Prolog*.

Exemplo 11.22:

Exemplos de expressões Prolog são:

- $e_1: p(X, f(Y) \& Z)$
 $e_2: p(X, Y) \& X$
 $e_3: p(X, f(a, h(Y \& X)) \& r(b))$

Em particular, note que em e_1 uma conjunção aparece como argumento do átomo p e em e_2 a variável X aparece no lugar de uma fórmula atômica. O terceiro exemplo, e_3 , apenas ilustra que conjunções poderão aparecer em um nível arbitrário de aninhamento, e não apenas no primeiro nível.

A cadeia abaixo, no entanto, não é uma expressão Prolog:

e₄: X(a,b,c)

pois a definição anterior, acompanhando VM/Prolog (ou MVS/Prolog), não permite a ocorrência de uma variável no lugar de uma átomo, ao contrário de alguns dialetos Prolog como micro-Prolog (Clark e McCabe [1984]).

A noção de cláusula Prolog é redefinida de tal forma que a cabeça de uma cláusula seja uma expressão atômica Prolog e o corpo seja uma conjunção de expressões atômicas ou variáveis Prolog.

Definição 11.2:

O conjunto das *cláusulas estendidas Prolog*, ou simplesmente das *cláusulas*, é o menor conjunto satisfazendo às seguintes condições:

- (i) se A é uma expressão atômica Prolog, então "A." é uma cláusula Prolog, chamada de *cláusula unitária estendida Prolog* (ou *cláusula terminal Prolog*);
- (ii) se B₁,...,B_n são expressões atômicas ou variáveis Prolog e A é uma expressão atômica Prolog então "A <- B₁ &...& B_n." é uma cláusula Prolog, chamada de *cláusula não-unitária estendida Prolog*, onde A é a *cabeça* e "B₁ &...& B_n" é o *corpo* da cláusula;
- (iii) se C₁,...,C_n são expressões atômicas ou variáveis Prolog, então "<- C₁ &...& C_n." é uma cláusula Prolog, chamada de *cláusula objetivo estendida Prolog*.

Exemplo 11.23:

Exemplos de cláusulas estendidas Prolog são:

- c₁: p(X,q(Y) & Z).
- c₂: p(X,Y) <- q(X) & Y.
- c₃: <- p(X,q(Y) & Z) & Z.

Note que nenhuma delas seria uma cláusula sintaticamente correta pela definição da seção 10.2.1. Note ainda que variáveis ocorrem no corpo de c₂ e c₃ diretamente no lugar anteriormente ocupado apenas por fórmulas atômicas.

As cadeias abaixo, no entanto, não são cláusulas estendidas Prolog:

$c_4: X.$

$c_5: X \leftarrow p(X, Y) \ \& \ Y$

pois a definição anterior não permite que a cabeça de uma cláusula estendida seja uma variável.

Uma vez redefinido o conceito de cláusula Prolog, a sintaxe dos *programas e consultas estendidos* Prolog segue como anteriormente. Porém, a semântica de programas e consultas estendidos necessita ajustes substanciais (o leitor poderá omitir a discussão sobre a semântica de programas estendidos em uma primeira leitura).

Como a sintaxe de programas e consultas estendidos foge bastante da sintaxe das linguagens de primeira ordem, a definição de uma semântica declarativa para esta extensão segue um caminho indireto, semelhante àquele da seção 10.3.1. Para tal, transformaremos inicialmente as expressões Prolog em listas Prolog com o auxílio de um átomo especial, "conj", para simular a conjunção. Por exemplo, a expressão " $p(a,b)$ " transforma-se na lista "(p . a . b . nil)" e a expressão " $r(p(a,b) \ \& \ X, q(b))$ " na lista "(r . ($conj$. (p . a . b . nil)). X . nil). (q . b . nil). nil)". Note que, desta forma, as conjunções Prolog transformam-se em termos e as variáveis Prolog passam novamente a ocorrer apenas no lugar de termos. Porém, as expressões atômicas Prolog também se transformam em termos e, portanto, não podem mais ser usadas diretamente para formar cláusulas. Para solucionar este último problema, introduzimos um segundo átomo especial, "vale", tal que " $vale((p.t_1. \dots . t_n).nil)$ " indica que a expressão atômica " $p(t_1, \dots , t_n)$ " deve ser entendida como verdadeira. Assim, uma cláusula da forma " $B_0 \leftarrow B_1 \ \& \dots \& B_n.$ " deve ser refraseada como " $vale(b_0) \leftarrow vale(b_1) \ \& \dots \& vale(b_n).$ ", onde b_i é a representação por lista da expressão B_i .

A definição abaixo captura estas observações.

Definição 11.3:

A função de canonização T , mapeando expressões Prolog em termos Prolog e cláusulas estendidas Prolog em cláusulas Prolog, é definida recursivamente da seguinte forma:

- (i) se " X " é uma variável Prolog, então $T("X") = "X"$;
- (ii) se " c " é uma constante Prolog, então $T("c") = "c"$;

(iii) se " $p(e_1, \dots, e_n)$ " é uma expressão atômica Prolog, então

$$T("p(e_1, \dots, e_n)") = "(p. T(e_1). \dots . T(e_n). nil)"$$

(iv) se " $e_1 \& \dots \& e_n$ " é uma conjunção Prolog, então

$$T("e_1 \& \dots \& e_n") = "(conj. T(e_1). \dots . T(e_n). nil)"$$

(v) se "A." é uma cláusula unitária Prolog, então

$$T("A.") = "vale(T(A).)"$$

(vi) se " $A \leftarrow B_1 \& \dots \& B_n.$ " é uma cláusula não-unitária Prolog então

$$\begin{aligned} T("A \leftarrow B_1 \& \dots \& B_n.") &= \\ "vale(T(A)) \leftarrow vale(T(B_1)) \& \dots \& vale(T(B_n))." \end{aligned}$$

(vii) se " $\leftarrow C_1 \& \dots \& C_n.$ " é uma cláusula objetivo Prolog, então

$$T("\leftarrow C_1 \& \dots \& C_n.") = "\leftarrow vale(T(C_1)) \& \dots \& vale(T(C_n))."$$

A transformação T permite então mapear um programa e uma consulta estendida Prolog em um programa e uma consulta de acordo com a definição antiga da seção 10.2.1. A transformação inversa T^{-1} por sua vez permite mapear respostas dadas em termos de listas representando expressões Prolog nas próprias expressões. Portanto, todo o desenvolvimento semântico da seção 10.3 aplica-se a programas e consultas estendidos via tais transformações, bastando acrescentar a cada programa cláusulas que captura o significado pretendido do átomo "conj".

Definição 11.4:

Sejam P e C um programa e uma consulta estendidas Prolog.

- (a) A *canonização* de C é a cláusula D resultante da aplicação de T a C .
- (b) A *canonização* de P é o programa R resultante da aplicação de T a cada cláusula de P e de acrescentar ao programa, nesta ordem, as cláusulas:

$c_1: \text{vale}((\text{conj. } X. \text{nil})) \leftarrow \text{vale}(X).$

$c_2: \text{vale}((\text{conj. } X. Y. \text{nil})) \leftarrow \text{vale}(X) \& \text{vale}((\text{conj. } Y. \text{nil})).$

- (c) Uma substituição $\beta = \{x_1/t_1, \dots, x_m/t_m\}$ de variáveis por expressões Prolog é uma *resposta correta* de C a P se e somente se $\theta = \{x_1/T(t_1), \dots, x_m/T(t_m)\}$ é uma resposta correta de D a R .

Exemplo 11.24:

Seja **P** o seguinte programa:

1. $p(p(a)).$
2. $p(a).$

e **C** a seguinte consulta:

3. $\neg p(X) \& X.$

A canonização de **P** é o programa **R** abaixo:

- 1'. $\text{vale}((\text{conj. } X. \text{ nil})) \leftarrow \text{vale}(X).$
- 2'. $\text{vale}((\text{conj. } X. Y. \text{ nil})) \leftarrow \text{vale}(X) \& \text{vale}((\text{conj. } Y. \text{ nil})).$
- 3'. $\text{vale}((p. (p. a. \text{ nil}). \text{ nil})).$
- 4'. $\text{vale}((p. a. \text{ nil})).$

e a canonização de **C** é a consulta **D** abaixo:

- 5'. $\neg \text{vale}((p. X. \text{ nil})) \& \text{vale}(X).$

Note que $\{X/(p. a. \text{ nil})\}$ é uma resposta correta de **D** a **R**. Logo, $\{X/p(a)\}$ será por definição uma resposta correta de **C** a **P**.

O tratamento das semânticas procedural e operacional pode naturalmente seguir o mesmo caminho. Porém, programas e consultas estendidas podem ser tratadas equivalente e diretamente através das seguintes modificações:

- a noção de unificação deve ser expandida para permitir a substituição de uma variável por uma expressão Prolog;
- o objeto selecionado em cada passo de uma refutação passa a ser uma expressão Prolog.

Quando uma variável é utilizada com fins explícitos de ser substituída por uma expressão Prolog que não seja um termo Prolog, ela passa a ser chamada de *metavariável* e o programa de *metaprograma*.

Exemplo 11.25:

Este exemplo ilustra como a máquina Prolog trabalha com metavariáveis. Considere o seguinte programa:

1. $p(p(a)).$
2. $p(a).$

e a seguinte consulta:

3. $\neg p(X) \& X.$

Com as modificações indicadas acima, a máquina Prolog produzirá a seguinte resposta (correta):

SUCESSO: $X = p(a)$

De fato, a máquina gerará a seguinte seqüência de cláusulas:

3. $\neg p(X) \& X.$
4. $\neg p(a). \quad .1, 3 \text{ com } \{X/p(a)\}$
5. $\square \quad .2, 4$

Seguem-se exemplos mais realistas de programação na metalinguagem.

Exemplo 11.26:

- (a) O programa a seguir lista todas as unificações de um dado termo X que satisfazem a uma conjunção de expressões P , nas quais X supostamente ocorre:

```
quais(X,P) :- P & nl & prst('Resposta: ') &
              writes(X) & fail.
quais(X,P) :- nl & prst('Fim das Respostas') & nl.
```

- (b) O programa a seguir adiciona uma dada cláusula na área de trabalho Prolog, na frente de todas cláusulas sobre o mesmo átomo no mesmo papel, se o usuário confirmar a inserção da mesma. A confirmação será solicitada somente se a cláusula não puder ser provada (como determina o uso de metavariável na primeira cláusula) a partir das cláusulas já existentes na área de trabalho.

```

confirme(C) :- C & / .
confirme(C) :- nl & writex("Confirme se") &
             nl & nl &
             writex(C) & nl & nl &
             writex("e' verdade (sim./nao.):") &
             nl &
             read(sim) &
             addax(C,1).

```

Nota: o comando extralógico "cut" na primeira cláusula evita que a segunda cláusula seja selecionada para unificação caso a primeira tenha sido satisfeita e ocorra uma solicitação de resposta alternativa.

Sessão 11.12:

Sejam "quais" e "confirme" os procedimentos definidos no exemplo anterior e "pdbragax" o programa referente a relações familiares na dinastia de Bragança, estudado no capítulo anterior.

```
<-consult(pdbragax).
SUCESSO
```

```
<-consult(quais).
SUCESSO
```

```
<-quais(X,filho('Pedro I',X)).
```

Resposta:

'Joaoo VI'

'Carlota Joaquina'

Fim das Respostas

```
<-consult(confirme).
SUCESSO
```

<-confirmefilho('Pedro I','Joao da Silva')).

Confirme se

filho('Pedro I','Joao da Silva')

e' verdade (sim./nao.):

nao.

FALHA

Não tendo sido possível provar a solicitação efetuada, se "'Pedro I'" foi filho de "'Joao da Silva'", o programa pediu para confirmar este fato. Com a resposta negativa, a cláusula não foi acrescentada na área de trabalho Prolog.

Passemos agora ao importante problema de simular os conectivos lógicos diretamente com o auxílio de metavariáveis.

Inicialmente, a negação pode ser fácil e eficientemente simulada simplesmente invertendo-se as respostas Prolog de SUCESSO e FALHA, através das seguintes cláusulas:

$n_1: \text{nao}(P) \leftarrow P \& / \& \text{fail}.$

$n_2: \text{nao}(P).$

ou, equivalentemente, declarando-se o operador prefixo " \neg ":

$n_0: \text{op}(" \neg ", \text{prefix}, 40).$

$n_1: \neg P \leftarrow P \& / \& \text{fail}.$

$n_2: \neg P.$

Estas cláusulas devem ser entendidas da seguinte forma:

- a cláusula n_0 apenas declara o operador " \neg ".
- a cláusula n_1 tenta provar $\neg P$ através da tentativa de estabelecer P . Se isto ocorrer, a combinação "cut/fail" retorna FALHA para $\neg P$.
- a cláusula n_2 retorna SUCESSO para " $\neg P$ " caso a cláusula n_1 falhe em estabelecer P .

Exemplo 11.27: Negação por Falha Finita

Suponha a área de trabalho Prolog carregada com a definição da negação por falha finita. A cláusula objetivo:

```
<- ~casado(joao,maria).
```

resultará em SUCESSO, de acordo com a implementação do operador “~” apresentada, se “casado(joao,maria)” não puder ser provada a partir das cláusulas da área de trabalho Prolog.

A implementação do operador Prolog “~” apresentada acima corresponde à regra da negação por falha finita e, portanto, simula a negação lógica apenas parcialmente. Em primeiro lugar, a chamada P da cláusula n₁ acima pode divergir e, portanto, deixar indeterminado o valor de “~P”. Em segundo lugar, a implementação é sensível à instanciação das variáveis e, portanto, depende incorretamente da ordem das fórmulas atômicas no corpo de uma cláusula, como ilustra o seguinte exemplo.

Exemplo 11.28: Negação por Falha Finita (Kowalski [1983])

Suponha a área de trabalho Prolog carregada com a definição do operador “~” e com as seguintes cláusulas:

```
c1. gosta(maria,joao).  
c2. deseja(maria,bicicleta).
```

Se submetermos a consulta:

```
<- ~deseja(X,Y) & gosta(maria,X).
```

obteremos FALHA como resposta, pois “deseja(X,Y)” resulta em SUCESSO, já que “deseja(maria,bicicleta)” existe na área de trabalho, o que implica que “~deseja(X,Y)” resulta em FALHA.

Porém, se submetermos a consulta:

```
<- gosta(maria,X) & ~deseja(X,Y).
```

obteremos SUCESSO como resposta, pois “gosta(maria,X)” resulta em SUCESSO, já que “gosta(maria,joao)” existe na área de trabalho, e “~deseja(joao,Y)” também resulta em SUCESSO, pois “deseja(joao,Y)” resulta em FALHA, já que não há nenhuma cláusula desta forma na área de trabalho.

Em geral, a implementação do operador “ \neg ” se comportará corretamente se supusermos que “ $\neg P$ ” significa “ $\neg(\exists x_1 \dots \exists x_n(P))$ ” ou, equivalentemente, “ $\forall x_1 \dots \forall x_n(\neg P)$ ” onde x_1, \dots, x_n são todas as variáveis livres de P . Se, por outro lado, a leitura pretendida para “ $\neg P$ ” for “ $\exists x_1 \dots \exists x_n(\neg P)$ ”, então a implementação produzirá resultados incorretos se alguma das variáveis x_1, \dots, x_n for instanciada durante a avaliação de P , como no exemplo acima, ou se comportará corretamente, se nenhuma das variáveis x_1, \dots, x_n for instanciada durante a avaliação de P .

Para evitar esta interpretação errônea, certas implementações de Prolog bloqueiam a avaliação de “ $\neg P$ ” até que todas as variáveis de P estejam instanciadas. Esta estratégia pode gerar assim uma reordenação do corpo de uma cláusula, fugindo da ordem padrão de seleção das fórmulas atômicas do corpo de uma cláusula (esquerda para a direita).

Observar finalmente que o operador “ \neg ” permite muitas vezes melhorar a legibilidade de programas Prolog ao substituir o comando extralógico “cut”.

A simulação dos outros conectivos lógicos através de metavariáveis segue o estilo da simulação do conectivo “ou”, repetida abaixo:

```
o0: op(ou,r1,20).
o1: X ou Y <- X.
o2: X ou Y <- Y.
```

Observe que este conectivo não é necessário em Prolog desde que cada ocorrência do mesmo pode ser substituída pela codificação de uma cláusula Prolog adicional. Em alguns casos, no entanto, a implementação deste conectivo permite melhorar a eficiência de um programa Prolog.

Exemplo 11.29:

A cláusula:

```
c1: p <- a & b ou c & d.
```

é equivalente às cláusulas:

```
c2: p <- a & b & d.
c3: p <- a & c & d.
```

Note porém que a primeira codificação é mais eficiente pois, na segunda, "a" será avaliada uma segunda vez se "b" falhar.

Como um outro exemplo de simulação de conectivos, as duas cláusulas abaixo:

```
s1: se(C,E,S) :- C & / & E.
s2: se(C,E,S) :- S.
```

simulam o conectivo ternário "se-então-senão". De fato, uma chamada "se(C,E,S)" resultará em SUCESSO, se:

- ao invocar s_1 , a chamada C resultar em SUCESSO, e em seguida a chamada E também resultar em SUCESSO;
- ao invocar s_1 a chamada C falhar finitamente e, ao invocar s_2 em seguida, a chamada S resultar em SUCESSO;

e resultará em FALHA, se:

- ao invocar s_1 , a chamada C resultar em SUCESSO, e em seguida a chamada E resultar em FALHA, pois o "cut" bloqueia o retrocesso;
- ao invocar s_1 a chamada C resultar em FALHA e, ao invocar s_2 em seguida, a chamada S resultar em FALHA.

Observe que a sintaxe da simulação do conectivo "se-então-senão" pode ser melhorada codificando-se dois operadores, "entao" e "senao", da seguinte forma:

```
s1: op(entao,r1,38).
s2: op(senao,r1,34).
s3: C entao E senao S :- C & / & E.
s4: C entao E senao S :- S.
```

Note que, como a prioridade do "entao" é maior do que a prioridade do "senao", uma chamada da forma "C entao E senao S" de fato terá a leitura pretendida.

Com o auxílio da negação por falha finita é possível simular ainda uma forma do quantificador universal. De fato, recorde inicialmente que a fórmula:

$$\forall x_1 \dots \forall x_n (P[x_1, \dots, x_n] \rightarrow Q[x_1, \dots, x_n])$$

onde x_1, \dots, x_n é uma lista das variáveis livres de P e Q, é equivalente a:

$$\neg \exists x_1 \dots \exists x_n (P[x_1, \dots, x_n] \wedge \neg Q[x_1, \dots, x_n])$$

Assim, o seguinte esquema de cláusulas:

$$\text{paratodo}(P, Q) \leftarrow \neg(P \wedge \neg Q).$$

simulará corretamente a primeira fórmula, se todas as variáveis de Q ocorrerem em P, para evitar interpretações errôneas da simulação da negação.

Para completar esta seção, observe que programação na metalinguagem permite definir comandos que geram, por retrocesso, todos objetos que satisfazem uma dada cláusula, acumulando-os em uma lista. Estes comandos, que são, em geral, predefinidos nos diversos dialetos Prolog, desempenham um papel importante em codificação Prolog, pois o mecanismo de retrocesso destrói instanciações de variáveis que podem ser essenciais em operações recursivas sobre estruturas de dados. Eles diferem na aceitação de quantificadores existenciais, na remoção de elementos duplicados e na classificação da lista resultante. Segue uma descrição de alguns destes comandos (na notação apresentada a seguir, X deve ser uma variável ocorrendo na cláusula C):

Comando	Descrição
<code>setof(X,C,L)</code>	L é a lista (sem repetições) classificada de todos objetos X que satisfazem a cláusula C.
<code>bagof(X,C,L)</code>	L é a lista (com possíveis repetições) não-classificada de todos objetos X que satisfazem a cláusula C.
<code>findall(X,C,L)</code>	L é a lista de todos objetos X que satisfazem a cláusula C. Todas variáveis de C não ocorrendo em X são tratadas como existencialmente quantificadas. A lista não é classificada e pode conter ou não (dependendo da implementação) repetição de elementos.

No dialeto VM/Prolog (ou MVS/Prolog) o comando "compute" desempenha o papel destes comandos com dois argumentos adicionais. Sua sintaxe e descrição é:

Comando	Descrição
<code>compute(T,X,C,V,L)</code>	L é a lista (se "T" estiver instanciada com "list") ou o conjunto (se "T" estiver instanciada com "set") de todos objetos X que satisfazem a cláusula C.

O parâmetro "V", que não será detalhado, é uma lista que permite particionar o resultado em um conjunto de resultados através de solicitação de retrocesso. Se esta lista for instanciada com `nil` não ocorre particionamento do resultado.

Codificações destes comandos, utilizando-se metavariáveis, podem ser encontradas em Bratko [1986], Li [1984] ou Kluzniak e Szpakowicz [1985]. Uma implementação de "setof" em Prolog pode ser encontrada em Pereira e Porto [1981]. Segue um exemplo de implementação do comando "findall" com repetição e outro exemplo sem repetição. Este comando difere do comando "bagof" no sentido que *todos* objetos X são coletados independentemente de possíveis diferentes soluções para variáveis não compartilhadas com X na cláusula C.

Exemplo 11.30: Uma implementação do comando "findall" com repetição

```

findall(X,C,L) :- C & addax(item(X)) & fail.
findall(X,C,L) :- coleta(L) & /.

coleta(X,Y) :- delax(item(X)) & coleta(Y).
coleta(nil).
```

Todas as soluções para a expressão C são geradas por retrocesso forçado. Cada solução gerada é adicionada na ATP para não ser perdida quando a próxima solução for atingida. Após todas as soluções terem sido geradas e armazenadas na ATP, elas são coletadas em uma lista e apagadas da ATP. Para tal, a expressão "item(X)" simula uma pilha. Quando todas as soluções tiverem sido coletadas, a pilha torna-se vazia.

Exemplo 11.31: Uma implementação do comando "findall" sem repetição

```

findall(X,C,L) :- C & salva_item(item(X)) & fail.
findall(X,C,L) :- coleta(L) & /.

coleta(X,Y) :- delex(item(X)) & coleta(Y).
coleta(nil).

salva_item(X) :- X & /.
salva_item(X) :- addax(X).

```

Em relação a implementação anterior, o procedimento sobre o átomo "salva_item", que substitui o comando "addax" na primeira cláusula, evita adicionar na ATP uma solução gerada que já exista na mesma.

Sessão 11.13: Comando "findall"

As consultas desta sessão foram efetuadas sobre as cláusulas unitárias básicas abaixo supostas carregadas na ATP:

```

pai('Pedro III','Joao VI').
pai('Joao VI','Pedro I').
pai('Joao VI',miguel).
pai('Pedro I','Pedro II').
pai('Pedro I','Maria da Gloria').

<-findall(X, pai('Pedro I',X), L) & writes(L) & nl.

SUCESSO:   L= ('Pedro II'.'Maria da Gloria'.nil)

<-findall(X, pai(Y,X), L) & writes(L) & nl.

SUCESSO:   L= ('Joao VI'.'Pedro I'.miguel.
              'Pedro II'.'Maria da Gloria'.nil)

<-findall(X/Y, pai(X,Y), L) & writes(L) & nl.

SUCESSO:   L= ('Pedro III'/'Joao VI'.
              'Joao VI'/'Pedro I'.
              'Joao VI'/miguel.
              'Pedro I'/'Pedro II'.
              'Pedro I'/'Maria da Gloria'.nil)

```

```
<-findall(X,Y,nil, pai(X,Y), L) & writes(L) & nl.  
SUCESSO: L= (('Pedro III','Joao VI'.nil).  
             ('Joao VI','Pedro I'.nil).  
             ('Joao VI'.miguel.nil).  
             ('Pedro I','Pedro II'.nil).  
             ('Pedro I','Maria da Gloria'.nil))
```

NOTAS BIBLIOGRÁFICAS

Kowalski [1981] apresenta algumas facilidades extralógicas da linguagem Prolog. Este artigo se preocupa principalmente em propor um uso disciplinado para as facilidades extralógicas apresentadas, através de encapsulamentos das mesmas em características de alto nível que permitam estender as primitivas lógicas da linguagem ou suas facilidades de controle. Em outro artigo, Kowalski [1983] discute novamente aspectos relativos as facilidades extralógicas da linguagem Prolog. Warren [1982] examina duas possíveis extensões, consideradas de alta-ordem (no sentido informal de pertinente a funções, relações, conjuntos,...), para a linguagem Prolog. Clark [1978] e Lloyd [1984] discutem o conceito de negação por falha em detalhe e Reiter [1978, 1984] aborda, por sua vez, a hipótese do mundo fechado. Sterling e Shapiro [1986] apresentam diversos capítulos e Walker et alli [1987] diversas subseções relativos às facilidades extra e metalógicas da linguagem Prolog.

Campbell [1984], Kluzniak [1985] e Van Caneghem [1986] contém discussões sobre implementação de Prolog.

CAPÍTULO 12: EXEMPLOS DO USO DE PROLOG

As duas primeiras seções deste capítulo apresenta dois exemplos detalhados do uso de Prolog. O primeiro deles refere-se ao problema genérico de geração de planos, enquanto que o segundo mostra um modo de associar Bancos de Dados e Inteligência Artificial pelo desenvolvimento de aplicações de bancos de dados regidas por regras que disciplinam sua atualização. A terceira seção conclui o capítulo com uma comparação de Prolog com LISP e Pascal.

Este capítulo só é recomendado para o nível avançado de leitura.

12.1 GERAÇÃO DE PLANOS

O problema de geração de planos consiste em obter uma seqüência de operações que possa levar um estado corrente de um certo universo a um estado objetivo.

Um estado deve ser entendido neste caso como uma coleção de fatos. Isto permite definir as operações através dos:

- fatos que adicionam;
- fatos que removem;
- pre-condições para sua aplicação válida.

As operações são entendidas como funções que incluem um estado como um dos argumentos e retornam um estado como seu valor.

O que torna o problema interessante é que diversas operações são realizáveis mas não sabemos quais as que levariam ao estado objetivo. Além disso, se uma operação adequada a essa finalidade não pode ser executada porque uma ou mais pre-condições não valem no estado corrente, admitiremos que essas pre-condições podem ser fixadas como sub-objetivos, isto é, procuraremos atingir um estado intermediário (ou seqüência de estados intermediários) até finalmente chegar ao estado objetivo. Como os estados intermediários devem por sua vez ser atingidos pelas operações disponíveis, o plano necessário para chegar ao estado objetivo consistirá em geral de uma seqüência de várias aplicações das operações.

Consideremos um universo em que os seguintes fatos são de interesse:

<code>caixa_em(x,s)</code>	no estado <code>s</code> , uma caixa encontra-se em uma posição <code>x</code>
<code>por_cima(s)</code>	no estado <code>s</code> , um macaco está sobre a caixa
<code>tem_bananas(s)</code>	no estado <code>s</code> , o macaco está de posse de um cacho de bananas

Suporemos, neste universo limitado a uma sala, que um cacho de bananas pende do teto em uma posição `c`. No estado corrente, o macaco não está de posse das bananas e elas estão fora de seu alcance. Se subir na caixa poderá alcançá-las, mas a caixa está em uma posição `d` diferente de `c`. O estado objetivo é simplesmente caracterizado pelo fato de o macaco passar a ter as bananas.

As operações que o macaco pode executar e que são diretamente relevantes ao problema são:

<code>empurre(x,s)</code>	a partir do estado <code>s</code> , empurrar a caixa para a posição <code>x</code> ;
<code>suba(s)</code>	a partir do estado <code>s</code> , subir na caixa;
<code>pegue(s)</code>	a partir do estado <code>s</code> , estender a mão para pegar as bananas.

Segue-se uma definição axiomática das operações que, ao mesmo tempo, indica seus efeitos (fatos adicionados ou removidos) e suas pre-condições, sem as quais os efeitos não se produzem (o símbolo predicativo "igual" denota a igualdade):

(1) $\forall s (\text{por_cima}(\text{suba}(s)))$

("em todo estado obtido pela aplicação da operação "suba" a qualquer estado s , o macaco estará sobre a caixa, ou seja, a operação, "suba" não tem pre-condições")

(2) $\forall x \forall y \forall s (\text{caixa_em}(x,s) \wedge \neg \text{igual}(x,y) \wedge \neg \text{por_cima}(s) \rightarrow \text{caixa_em}(y, \text{empurre}(y,s)))$

("se em um estado s a caixa estiver numa posição x diferente de outra posição y e o macaco não estiver sobre a caixa, então a caixa estará na posição y em todo estado obtido pela aplicação da operação "empurre", com argumento y , aplicada a s ")

(3) $\forall s (\text{por_cima}(s) \wedge \text{caixa_em}(c,s) \rightarrow \text{tem_bananas}(\text{pegue}(s)))$

("se em um estado s o macaco estiver sobre a caixa e esta estiver na posição c , então o macaco estará de posse das bananas no estado obtido pela aplicação da operação "pegue" a s ")

Deveríamos com esses axiomas poder demonstrar o teorema:

$\exists s (\text{tem_bananas}(s))$

onde o estado objetivo s , que é qualquer estado em que o macaco tenha as bananas, seria denotado por alguma seqüência de operações (plano) apto a atingí-lo, seqüência essa a ser obtida no curso da demonstração do teorema.

Entretanto os três axiomas mostrados não são suficientes, em virtude de uma dificuldade geral que complica o processo de geração de planos, denominado problema do entorno ("frame problem"). Em resumo, não basta indicar os efeitos de uma operação para definir o novo estado; é preciso indicar também o que permanece inalterado do estado anterior, sendo assim "copiado" na formação do novo estado. O inconveniente é que o número de axiomas adicionais para esta finalidade tende a crescer de forma exponencial.

Para os fins imediatos do exemplo dado, basta um desses axiomas:

(4) $\forall x \forall s (\text{caixa_em}(x,s) \rightarrow \text{caixa_em}(x,\text{suba}(s)))$

("se em um estado s , a caixa está na posição x , então estará nessa posição em todo estado obtido pela aplicação da operação "suba" a s . Em outras palavras: a operação "suba" não afeta o fato "caixa_em"").

Ilustraremos agora como a resposta computada por uma dedução pode ser interpretada como um plano para o macaco apanhar as bananas a partir de um dado estado inicial.

Considere a seguinte dedução linear abaixo a partir das cláusulas (1) a (9) e iniciando na cláusula em (9), onde:

- as cláusulas em (1) a (4) são uma representação clausal dos axiomas;
- as cláusulas em (5) a (7) caracterizam o estado inicial, que é denotado pela constante s_0 (a cláusula (7) não é na verdade necessária neste caso);
- a cláusula em (8) indica que as posições c e d são diferentes;
- a cláusula em (9) é a negativa do teorema, acrescida do literal de resposta, como explicado no Capítulo 8.

1. $\text{por_cima}(\text{suba}(s))$
2. $\neg \text{caixa_em}(x,s) \text{ igual}(x,y) \text{ por_cima}(s) \text{ caixa_em}(y,\text{empurre}(y,s))$
3. $\neg \text{por_cima}(s) \neg \text{caixa_em}(c,s) \text{ tem_bananas}(\text{pegue}(s))$
4. $\neg \text{caixa_em}(x,s) \text{ caixa_em}(x,\text{suba}(s))$
5. $\text{caixa_em}(d,s_0)$
6. $\neg \text{por_cima}(s_0)$
7. $\neg \text{tem_bananas}(s_0)$
8. $\neg \text{igual}(d,c)$
9. $\neg \text{tem_bananas}(s) \text{ r}(s)$
10. $\neg \text{por_cima}(s) \neg \text{caixa_em}(c,s) \text{ r}(\text{pegue}(s))$. 9a, 3c
11. $\neg \text{caixa_em}(c,\text{suba}(s)) \text{ r}(\text{pegue}(\text{suba}(s)))$. 10a, 1a
12. $\neg \text{caixa_em}(c,s) \text{ r}(\text{pegue}(\text{suba}(s)))$. 11a, 4b
13. $\neg \text{caixa_em}(x,s) \text{ por_cima}(s) \text{ igual}(x,c)$
 $\text{r}(\text{pegue}(\text{suba}(\text{empurre}(c,s))))$. 12a, 2d
14. $\text{por_cima}(s_0) \text{ igual}(d,c) \text{ r}(\text{pegue}(\text{suba}(\text{empurre}(c,s_0))))$. 13a, 5a
15. $\text{igual}(d,c) \text{ r}(\text{pegue}(\text{suba}(\text{empurre}(c,s_0))))$. 14a, 6a
16. $\text{r}(\text{pegue}(\text{suba}(\text{empurre}(c,s_0))))$. 15a, 8a

Como a resposta computada por esta dedução é a substituição $\beta = \{s/\text{pegue}(\text{suba}(\text{empurre}(c,s_0)))\}$, a conclusão (óbvia neste caso simples) é de que o macaco deverá realizar a seqüência de operações:

empurre(c), suba, pegue

aplicadas sobre o estado inicial s_0 , para apanhar as bananas.

A implementação em Prolog é imediata. O programa segue abaixo.

```
/*
  Primeira Solucao em Prolog
*/
por_cima(suba(X)).

caixa_em(d,s0).
caixa_em(X,empurre(X,Y)) <- caixa_em(Z,Y) & ne(X,Z) &
                           \> por_cima(Y) & /.
caixa_em(X,suba(Y)) <- caixa_em(X,Y).

tem_bananas(pegue(X)) <- por_cima(X) & caixa_em(c,X).
```

A segunda cláusula exprime o único fato acerca do estado corrente s_0 necessário neste caso, já que o fato do macaco não se encontrar sobre a caixa e o fato de não ter as bananas resultam da negação por falha.

Feita a consulta:

<- tem_bananas(X).

resulta a resposta:

SUCESSO: X = pegue(suba(empurre(c,s0)))

Além desta primeira solução do problema, é bem conhecida a alternativa de se recorrer à Lógica de Segunda Ordem para superar o problema do entorno. Essencialmente podemos simular Lógica de Segunda Ordem em Prolog utilizando um símbolo predutivo "vale" e considerando os antigos símbolos preditivos "por_cima", "caixa_em" e "tem_bananas" agora como símbolos funcionais, do mesmo modo como vimos tratando as

operações. Informalmente, uma afirmação X vale em qualquer dos seguintes casos:

- se X for declarada como um fato, será entendido que X vale em s_0 ;
- se X puder ser adicionada por uma operação O e se a aplicação de O for válida (pre-condições satisfeitas) em um estado s , então X vale no estado obtido aplicando O a s ;
- se X valer em um estado s e a operação não remover X , então X vale no estado obtido aplicando O a s (previsão para o problema do entorno).

Antes de apresentar o programa, algumas observações são pertinentes. O programa contém uma parte que não depende do problema específico (aqui o problema do macaco), sendo portanto aplicável a outros casos de geração de planos. O programa utiliza o procedimento "inv" para inversão de listas, descrito na seção 10.5, e adota a notação de Edinburgh para listas.

```
/*
  Segunda Solucao em Prolog
 */

/*
  Parte Geral
 */

plano(X,Y) <- vale(X,Z) & inv(Z,Y).

vale(X,[s0])  <- fato(X) & /.
vale(X,[Y|Z]) <- adicionado(X,Y) & valido(Y,Z).
vale(X,[Y|Z]) <- var(Y) & / & fail.
vale(X,[Y|Z]) <- \removido(X,Y) & vale(X,Z).
```

```

/*
  Parte Especifica
*/

adicionado(tem_bananas, pegue).
adicionado(por_cima, suba).
adicionado(caixa_em(X), empurre(X)).

removido(caixa_em(X), empurre(Y)) <- ne(X,Y).

valido(pegue, X)      <- vale(por_cima, X) &
                        vale(caixa_em(c), X) & /.
valido(empurre(X), Y) <- vale(caixa_em(Z), Y) & ne(Z,X) &
                        not vale(por_cima, Y) & /.
valido(suba, X)        <- /.

/*
  Estado Corrente
*/
fato(caixa_em(d)).

```

Nota: var(Y) será satisfeita se Y for uma variável.

A terceira cláusula do procedimento Prolog "vale" evita que a quarta cláusula, correspondente ao problema do entorno, seja invocada com uma variável não instanciada (que deveria indicar o nome de uma operação). Sem a inclusão desta cláusula, uma chamada ao procedimento resultaria em um laço infinito.

Fazendo a consulta:

```
<- plano(tem_bananas, X).
```

obtemos:

SUCESSO: X = [s0, empurre(c), suba, pegue]

É interessante notar o estilo mais intuitivo de exibir o plano obtido, sob a forma de uma lista representando a seqüência de operações, em vez do formato anterior de composição de funções.

Na seção seguinte veremos um exemplo que tem algumas semelhanças com este por também envolver uma simulação de Lógica de Segunda Ordem. O objetivo é no entanto diferente: os fatos que caracterizam um universo passam a constituir um banco de dados. Em vez de geração de planos, o problema passa a ser como aplicar as operações de modo a atualizar o banco de dados de forma correta.

12.2 DESENVOLVIMENTO DE APLICAÇÕES DE BANCOS DE DADOS

Esta seção mostra um modo de associar Bancos de Dados e Inteligência Artificial pelo desenvolvimento de aplicações de bancos de dados regidas por regras que disciplinam sua atualização. Essas regras incorporam, de forma declarativa, o conhecimento de especialistas na área de aplicação. Esta seção discute ainda o conceito de bancos de dados temporais e apresenta um exemplo elementar utilizando Prolog em combinação com o sistema SQL/DS.

Recorde que o conteúdo do banco de dados em dado momento constitui um *estado* do banco de dados. Porém, nem todos os estados são válidos, mas apenas aqueles que satisfazem certas regras denominadas *restrições de integridade estáticas*.

As *operações de atualização* transformam em outro o estado corrente do banco de dados. Uma mudança de estado se denomina uma *transição*. Nem todas as transições entre estados válidos são por sua vez válidas, mas apenas as que obedecem às *restrições de integridade de transição* que forem especificadas.

No projeto de banco de dados é fundamental impor uma disciplina que garanta a preservação das restrições de integridade, quer estáticas quer de transição. Uma estratégia particularmente eficiente consiste em limitar as operações de atualização a um elenco predefinido, em que cada operação inclui *precondições* (assertivas) e pode provocar a execução de outras operações, mecanismo conhecido como *gatilho*. As precondições e gatilhos devem ser definidos de tal forma que, em conjunto, mantenham a integridade.

Quando falha uma precondição de uma operação, fica estabelecido que a operação não é executada; se qualquer das operações de uma seqüência encadeada pelo mecanismo de gatilhos falha, todas as operações da seqüência são desfeitas. Uma tal seqüência de operações, em que ou todas

são executadas com sucesso ou não se produz nenhum efeito, denomina-se uma *transação*.

Consideraremos que um banco de dados é *temporal* se os fatos nunca são, em princípio, removidos e sempre vem acompanhados de um *selo temporal* indicando o instante em que passaram a valer. Em bancos de dados temporais podemos assim consultar não apenas se um fato vale mas quando vale. Além disso, torna-se possível introduzir restrições de integridade de transição que se refiram explicitamente a tempo.

Há mais de um modo de implementar bancos de dados temporais. Utilizaremos uma organização bastante simples, na qual também nos ocuparemos da questão da integridade. Na realidade, a preservação de restrições de integridade de transição pressupõe, em geral, alguma forma de banco de dados temporal.

A organização adotada é *orientada para a atualização*, no sentido de que, em vez de armazenarmos os fatos, registramos que operações de atualização foram executadas e quando foram. Para isso, a cada operação de atualização faremos corresponder uma tabela, cujas tuplas registram as listas de argumentos com que a operação foi chamada. Cada tupla incluirá ainda o selo temporal, registrando o momento de execução da operação, sob a forma de data (ano/mês/dia) concatenada com instante no dia (hora/minuto/segundo). Essas informações são obtidas pela leitura do relógio interno da máquina ao ser iniciada a execução da operação.

Mas só armazenaremos uma tal tupla se as precondições estabelecidas se verificarem. Além disso, exigiremos que a operação seja *produtiva*, isto é, nenhum dos fatos que ela deveria adicionar já valem e todos os que deveria remover não valem. Finalmente, adicionaremos gatilhos a algumas operações.

Os fatos serão então avaliados segundo a regra:

- um fato F vale em um momento indicado T se alguma operação O capaz de adicionar F tiver sido executada em um momento T', sendo T' anterior ou simultâneo a T, e nenhuma operação O' capaz de remover F tiver sido executada entre T' e T.

Claramente a estratégia que escolhemos terá significado prático somente se combinar Programação em Lógica com um sistema de gerência de bancos de dados. Tendo apenas Programação em Lógica, ficaríamos limitados a "simulações" na própria memória principal. Por outro lado,

tendo apenas um sistema de gerência de bancos de dados, não teríamos como incluir, de forma declarativa, as regras correspondentes a precondições, gatilhos, etc.

Vamos considerar um banco de dados exemplo em que os fatos são

- cursos são oferecidos
- alunos tomam cursos

As restrições de integridade são:

Restrição Estática:

r_1 : um aluno só pode tomar um curso que esteja sendo oferecido

Restrições de Transição:

r_2 : um novo curso só pode ser oferecido até 15 de março do ano corrente;

r_3 : um aluno não pode retornar a um curso em que já tenha estado antes;

r_4 : nenhum curso que estiver sem alunos após 20 de março pode continuar a ser oferecido;

r_5 : o número de cursos feitos por um aluno não pode decrescer.

Suponhamos que somente as operações abaixo sejam consideradas necessárias:

- oferecer um curso
- matricular um aluno em um curso
- transferir um aluno de um curso para outro
- cancelar um curso

Além dessas operações, teremos comandos para inicializar e terminar uma sessão, por motivos que ficarão claros mais adiante.

Note que não foi prevista (supostamente por não ser julgada necessária) uma operação para um aluno abandonar um curso. Por esse motivo, a restrição r_5 é trivialmente mantida, não nos obrigando a impor precondições nem associar gatilhos às operações. Entretanto as demais restrições exigem precondições ou gatilhos, conforme descrito a seguir.

A operação "oferecer" tem como precondição só poder ser executada até a data limite e a operação "matricular", a condição do curso ser oferecido e não ter sido anteriormente tomado pelo aluno.

A operação "transferir", além de precondições análogas às de "oferecer" e "matricular", tem associado o seguinte gatilho: se um aluno A é transferido de um curso C para outro qualquer, sendo A o único aluno tomado C e sendo a data corrente posterior a 20 de março, o curso C deverá ser automaticamente cancelado e o usuário simplesmente avisado disso.

Esta providência não basta, no entanto, para garantir r₄ pois a simples ocorrência da data limite poderia tornar necessário o cancelamento de um curso que até então permanecesse (validamente) sem alunos. Para cobrir este caso, temos de associar um gatilho também ao comando de inicialização, o que é suficiente se admitirmos que cada sessão de uso do banco de dados se passa dentro de um único dia.

Finalmente, a operação "cancelar" poderia ter como precondição que nenhum aluno esteja tomando o curso ou como gatilho (solução pela qual optamos neste exemplo) que após sua execução todos os alunos que estivessem tomando o curso fossem transferidos para outros cursos. Naturalmente este gatilho não deve ser totalmente automático, convindo estabelecer que o usuário seja consultado para indicar, para cada um desses alunos, para qual curso deverá ser transferido.

A implementação descrita abaixo utiliza uma combinação dos sistemas VM/PROLOG e SQL/DS. Nela distinguem-se a parte geral destinada a uma classe ampla de aplicações, e a parte específica que deve ser elaborada para cada aplicação. Os símbolos predicativos fundamentais da parte geral são "inicie", "termine", "execute" e "vale", com a seguinte interpretação pretendida:

inicie()	inicializa uma sessão, invocando gatilhos se for o caso
termine()	encerra uma sessão, devolvendo o controle ao sistema operacional
execute(0)	executa uma operação 0 predefinida
vale(F,T)	apura a ocorrência de fatos em um instante T

Além destes, usaremos, entre outros, os seguintes símbolos predicativos auxiliares: "agora", "antes", "no_maximo", "produtivo". O significado pretendido desses símbolos é:

agora(T) retorna em T o momento corrente, obtido do relógio interno
antes(T,U) é verdadeiro se T é anterior a U
no_maximo(T,U) é verdadeiro se T e U coincidem ou se T é anterior a U
produtivo(X,T) determina se a execução de uma operação X é produtiva no sentido discutido antes

```
/*
  Parte Geral
*/
```

```

inicie() <-
  agora(T) &
  gatilho(inicie(),T).

termine() <-
  rexvar('FLAG','F') &
  fin().

execute(X) <-
  agora(T) &
  (exec(X,T) & / |
   sql('rollback work',*) & fail).

exec(X,T) <-
  agora(T) &
  valido(X,T) &
  produtivo(X,T) &
  inc_tempo(X,T,0) &
  insira(0) &
  agora(V) &
  gatilho(X,V).

vale(X,T) <-
  adicionado(X,Y) &
  inc_tempo(Y,V,0) &
  questione(0) &
  no_maximo(V,T) &
  - (removido(X,Z) &
     inc_tempo(Z,W,P) &
     questione(P) &
     antes(V,W) &
     no_maximo(W,T)).

```

```

agora(T) <- suspend() &
    rexvar('DATA',X) &
    rexvar('HORA',W) &
    substring(X,Y,0,5) &
    substring(X,Z,6,2) &
    T := Z || '/' || Y || ' ' || W.

antes(X,Y) <-
    (var(X) | var(Y)) & / & fail.
antes(X,Y) <-
    lt(X,Y).

no_maximo(X,Y) <-
    X = Y & / |
    lt(X,Y).

produtivo(X,T) <-
    paratodo(adicionado(Z,X), \+ vale(Z,T)) &
    paratodo(removido(W,X),vale(W,T)).

questione(X) <-
    crie_cons(X,Y,Z) &
    sql(Y,Z,*).

crie_cons([F|A],Y,W) <-
    prep_cons(A,W) &
    Y := 'select * from ' || F.

prep_cons([],[]).
prep_cons([X|Y],[X1|Y1]) <-
    (var(X) & X1 = X & / |
     stringp(X) & X1 = X & / |
     st_to_at(X1,X)) &
    prep_cons(Y,Y1).

insira(X) <-
    crie_ins(X,Y) &
    sql(Y,*).

crie_ins([F|A],Y) <-
    inc_sep(A,Z) &
    cat(Z,W) &

```

```

Y := 'insert into ' || F || ' values(' || W || ')'.

inc_sep([X],['''',X,'''']). 
inc_sep([X|Y],['''',X, '''', '|Z])
  <- inc_sep(Y,Z).

inc_tempo(X,T,0) <-
  X =.. [F|A] &
  0 = [F,T|A]. 

cat(X,Y) <- cat1(X,Z) & st_to_li(Y,Z).

cat1([X|Y],Z) <-
  (stringp(X) & W = X & / |
   st_to_at(W,X)) &
  st_to_li(W,L) &
  cat1(Y,U) &
  concat(L,U,Z).
cat1([],[]).

paratodo(X,Y) <-
  -(call(X) & - call(Y)). 

quais(X,Y) <-
  call(Y) &
  nl & prst('Resposta: ') &
  writes(X) & fail.
quais(X,Y) <-
  nl & prst('Fim das respostas') & nl & nl.

```

A parte específica contém a especificação de cada operação de atualização, através dos procedimentos Prolog:

- adicionado
- removido
- valido
- gatilho

onde, para cada operação, são indicados que fatos ela pode adicionar ou remover e quais precondições e gatilhos estão associados a ela.

O programa Prolog correspondente à parte específica segue abaixo:

```

/*
Parte Especifica
*/

adicionado(oferecido(X),ofereca(X)).
adicionado(toma(X,Y),matricule(X,Y)).
adicionado(toma(X,Z),transfira(X,Y,Z)).

removido(oferecido(X),cancele(X)).
removido(toma(X,Y),transfira(X,Y,Z)).

valido(ofereca(X),T) <-
    substring(T,V,3,5) &
    antes(V,'03/15').
valido(matricule(X,Y),T) <-
    vale(oferecido(Y),T) &
    \+ vale(toma(X,Y),V).
valido(transfira(X,Y,Z),T) <-
    vale(oferecido(Z),T) &
    \+ vale(toma(X,Z),V).
valido(cancele(X),T).

gatilho(inicie(),T) <-
    substring(T,V,3,5) &
    antes(V,'03/20') & / |
    paratodo(vale(oferecido(Z),T) & \+ vale(toma(W,Z),T),
              exec(cancele(Z),T) &
              M := 'curso ' || Z || ' cancelado' &
              nl & prst(M) & nl).
gatilho(ofereca(X),T).
gatilho(matricule(X,Y),T).
gatilho(transfira(X,Y,Z),T) <-
    \+ vale(oferecido(Y),T) & / |
    substring(T,V,3,5) &
    antes(V,'03/20') & / |
    vale(toma(S,Y),T) & \+ (S = X) & / |
    M := 'curso ' || Y || ' cancelado' &
    & nl & prst(M) & nl &
    exec(cancele(Y),T).
gatilho(cancele(X),T) <-
    nl & paratodo(vale(toma(Y,X),T),
                  W := 'indique outro curso para ' || Y &

```

```
prst(W) & nl &
read(Z) &
exec(transfira(Y,X,Z),T)).
```

Como exemplo, suponhamos que ao inicio de uma sessão realizada a 15 de abril sejam verdadeiros os fatos:

- o curso c_1 é oferecido
- o curso c_2 é oferecido
- o curso c_3 é oferecido
- o aluno João toma o curso c_1
- a aluna Maria toma o curso c_1
- o aluno Luiz toma o curso c_2

A execução do comando

```
<- inicie.
```

resultará no cancelamento do curso c_3 , que permanece sem alunos após a data limite de 20 de março (restrição r_4). O comando

```
<- execute(cancel(c1)).
```

resulta em ser o usuário solicitado a indicar novos cursos para João e para Maria (c_2 em ambos os casos, por ser a única alternativa de curso oferecido). Com isso, os dois alunos são transferidos e o curso é cancelado. Para termos os fatos correntes executamos

```
<- agora(T) & quais(X,vale(X,T)).
```

obtendo, sucessivamente, as informações:

- o curso c_2 é oferecido
- o aluno Luiz toma o curso c_2
- o aluno João toma o curso c_2
- a aluna Maria toma o curso c_2

Entretanto os estados anteriores do banco de dados continuam acessíveis. Podemos, por exemplo recuperar o histórico escolar de João (em que cursos ele está ou esteve matriculado e quando os iniciou) por:

```
<- quais([C,T],vale(toma(joao,C),T)).
```

o que nos daria como resposta c_1 e c_2 seguidos, respectivamente, do selo temporal associado à matrícula de João em c_1 e à sua transferência para c_2 .

As tabelas ao final da sessão figuram abaixo. Note que, em virtude da disciplina adotada, as tuplas em cada tabela estão classificadas pelo selo temporal (atributo st).

OFEREÇA	
<i>st</i>	<i>curso</i>
86/02/20 14:30:51	c_1
86/02/21 14:30:47	c_2
86/02/27 16:25:03	c_3

MATRICULE		
<i>st</i>	<i>aluno</i>	<i>curso</i>
86/03/03 15:20:23	joao	c_1
86/03/03 15:30:04	maria	c_1
86/03/03 15:35:32	luiz	c_2

TRANSFIRA			
<i>st</i>	<i>aluno</i>	<i>curso1</i>	<i>curso2</i>
86/04/15 11:27:55	joao	c_1	c_2
86/04/15 11:27:55	maria	c_1	c_2

CANCELE	
<i>st</i>	<i>curso</i>
86/04/15 11:26:30	c_3
86/04/15 11:27:55	c_1

Para completar esta seção, observamos que a pesquisa em sistemas especialistas de bancos de dados está apenas começando. A ligação de Programação em Lógica e aplicações de bancos de dados de grande porte envolve problemas de eficiência que devem ser resolvidos antes de seu uso poder generalizar-se. Existe ainda, entre outros, o problema de projetar interfaces para os usuários, que sejam fáceis de usar e escondam a complexidade desses sistemas.

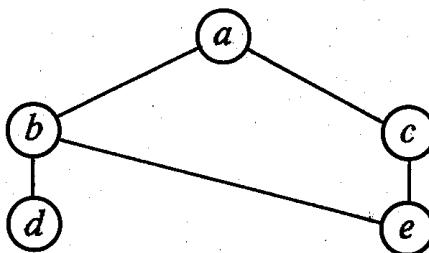
12.3 COMPARAÇÃO DE PROLOG COM OUTRAS LINGUAGENS

Esta seção apresenta, através de um exemplo, uma breve comparação entre Prolog e as linguagens LISP e Pascal. A finalidade da comparação não é ressaltar as vantagens de Prolog, embora o exemplo escolhido a favoreça. Na verdade, é fácil achar exemplos em que LISP (ex: cálculo de fatorial) ou Pascal (ex: resolução de sistemas de equações lineares) é mais apropriado do que as duas outras linguagens. Assim, o objetivo desta seção é apenas apontar algumas diferenças fundamentais no estilo de programar em cada paradigma. Alertamos o leitor para o fato de que programas diferentes dos nossos poderiam ser elaborados para o mesmo exemplo, com os quais, naturalmente, seria possível sugerir outros pontos relevantes à comparação.

LISP é um exemplo bem conhecido do paradigma de programação funcional, que engloba as linguagens baseadas essencialmente na definição e avaliação de funções. Já Pascal representa o paradigma de programação imperativa ou procedimental, correspondente às linguagens que oferecem ao programador um elenco de *estruturas de controle* e *estruturas de dados*. Além disso, tais linguagens incluem, como característica fundamental do modelo de Von Neumann, o *comando de atribuição*.

O exemplo consiste em percorrer um grafo dirigido acíclico em profundidade ("depth-first search" ou, abreviadamente, "dfs"). Um percurso em profundidade, a partir de um nó x qualquer, consiste em visitar um nó y adjacente a x , selecionado arbitrariamente, e iniciar em seguida um percurso em profundidade, recursivamente, a partir de y ; somente após esgotar a possibilidade avançar em profundidade no grafo, retorna-se para considerar como alternativas os caminhos através dos outros nós adjacentes a x .

Utilizaremos para ilustração o seguinte grafo:



Um percurso em profundidade neste grafo, a partir do nó *a*, visitaria os nós, por exemplo, na ordem $\langle a, b, d, e, c, e \rangle$. Como não estamos marcando os nós já visitados, o nó *e* será visitado duas vezes, já que existem dois caminhos para atingí-lo.

Um programa em Prolog implementando percurso em profundidade é apresentado abaixo. A fórmula atômica `dfs(X,Y)` deve ser lida como "partindo de *X*, *Y* é atingível pelo percurso em profundidade". Com esse entendimento, é claro que um nó é atingível a partir dele mesmo; além disso, a partir de *X* atingimos um nó *Y* se *X* for adjacente a algum nó *Z*, do qual por sua vez *Y* possa ser atingido.

```

/*
  Pesquisa em Profundidade em Prolog
*/
dfs(X,X).
dfs(X,Y) <-
  adj(X,Z) &
  dfs(Z,Y).
  
```

Note que o caso em que *X* é imediatamente adjacente a *Y* é coberto através das duas cláusulas (*Z* e *Y* podem corresponder a um mesmo nó). O procedimento Prolog `adj` é constituído de cláusulas básicas que enumeram as arestas do grafo:

```
/*
  Relação de Adjacência do Grafo em Prolog
*/
adj(a,b).
adj(a,c).
adj(b,d).
adj(b,e).
adj(c,e).
```

Para obter os nós atingíveis a partir de "a" em dfs, basta escrever:

```
<- dfs(a,N).
```

e entrar com ";" repetidamente até obter todos os valores para N:

```
SUCESSO: N= a ;
SUCESSO: N= b ;
SUCESSO: N= d ;
SUCESSO: N= e ;
SUCESSO: N= c ;
SUCESSO: N= e ;
```

Analizando o programa, verificamos que o prosseguimento do percurso a partir de um nó "mais abaixo" é explicitamente indicado pela chamada recursiva na segunda cláusula de dfs. A posterior consideração de outro nó adjacente, no entanto, não está explícita; ela é consequência do mecanismo de retrocesso, inerente ao processador Prolog e invocado por ";".

Embora tais mecanismos implícitos sejam responsáveis pela simplicidade do programa, eles trazem uma séria dificuldade à sua compreensão. Se examinarmos a semântica declarativa do programa, podemos apenas concluir que o predicado dfs exprime a relação de conectividade, por essencialmente exprimir o fecho transitivo da relação de adjacência; nada podemos concluir sobre a ordem em que os nós são obtidos. Sob o ponto de vista da semântica declarativa teríamos que admitir que as duas definições de dfs a seguir são equivalentes à primeira, por representarem simples trocas de ordem ou das cláusulas ou dos literais do corpo de uma das cláusulas:

```
/*
  Formas Alternativas de Pesquisa em Prolog
*/
dfs_a(X,Y) <-
  adj(X,Z) &
  dfs_a(Z,Y).
dfs_a(X,X).

dfs_b(X,X).
dfs_b(X,Y) <-
  dfs_b(Z,Y) &
  adj(X,Z).
```

A semântica operacional de `dfs`, `dfs_a` e `dfs_b` entretanto é diferente. Apenas `dfs` implementa o percurso em profundidade corretamente. A execução de

`<- dfs_a(a,X).`

resulta em:

```
SUCESSO: N= d ;
SUCESSO: N= e ;
SUCESSO: N= b ;
SUCESSO: N= e ;
SUCESSO: N= c ;
SUCESSO: N= a ;
```

ou seja, é "adiada" a visita aos nós para o retorno das chamadas recursivas. Já o comportamento de `dfs_b` diverge de `dfs` de forma ainda mais radical: o grafo é percorrido em amplitude ("breadth-first search"), após o que a execução entra em um laço infinito em consequência da chamada recursiva à esquerda na segunda cláusula. Para entender a diferença entre `dfs` e `dfs_b`, temos de considerar que, em Prolog, apesar dos antecedentes de uma cláusula serem processados da esquerda para a direita, as alternativas dos antecedentes mais à direita são exploradas primeiro.

Passemos agora à implementação em LISP, que pode ser feita através das duas funções mutuamente recursivas dadas abaixo:

```

/*
  Pesquisa em Profundidade em LISP.
*/
dfs[x] = cons[x;dfs1[1adj[x]]]

dfs1[z] = [null[z] -> nil;
           t -> append[dfs[car[z]];dfs1[cdr[z]]]]

```

Note de início que em LISP não temos relações, mas apenas funções, as quais só podem retornar um resultado para cada argumento. Isso nos leva a colocar o resultado de `dfs` sob a forma de uma lista em que os nós aparecem na ordem desejada. Em palavras, `dfs` e `dfs1` exprimem:

"O resultado de `dfs` de x é a lista cujo primeiro elemento é x e o restante é obtido aplicando `dfs1` à lista dos nós adjacentes a x ; o resultado de `dfs1`, se a lista de nós adjacentes é vazia, é também a lista vazia, senão é o resultado da concatenação da lista obtida aplicando `dfs` ao primeiro elemento da lista ("mergulho" em profundidade) com a lista obtida por `dfs1` sobre os elementos restantes (exploração dos demais nós adjacentes)".

A chamada `a dfs[a]` retorna `(a b d e c e)`.

Agora a relação de adjacência também deve conformar-se ao estilo de programação funcional. A função `ladj`, dado um nó `x`, retorna a lista dos nós adjacentes a `x`. Se `x` for "b", por exemplo, será retornada a lista (`d e`). A definição de `ladj` envolve a função auxiliar `ladj1`, que é chamada tendo como argumentos o próprio nó `x` e a *lista de adjacências* do grafo. Esta consiste de uma sub-lista para cada nó, contendo o nó seguido dos nós adjacentes a ele. A função `ladj1` percorre a lista até achar a sub-lista iniciada por `x`, retornando então a sua continuação.

```

/*
  Funções Auxiliares para Definição do Grafo em LISP
*/
ladj[x] = ladj1[x; ((a b c)(b d e)(c e)(d)(e))]

ladj1[x; z] = [eq[x; caar[z]] -> cdar[z];
                t -> ladj1[x; cdr[z]]]

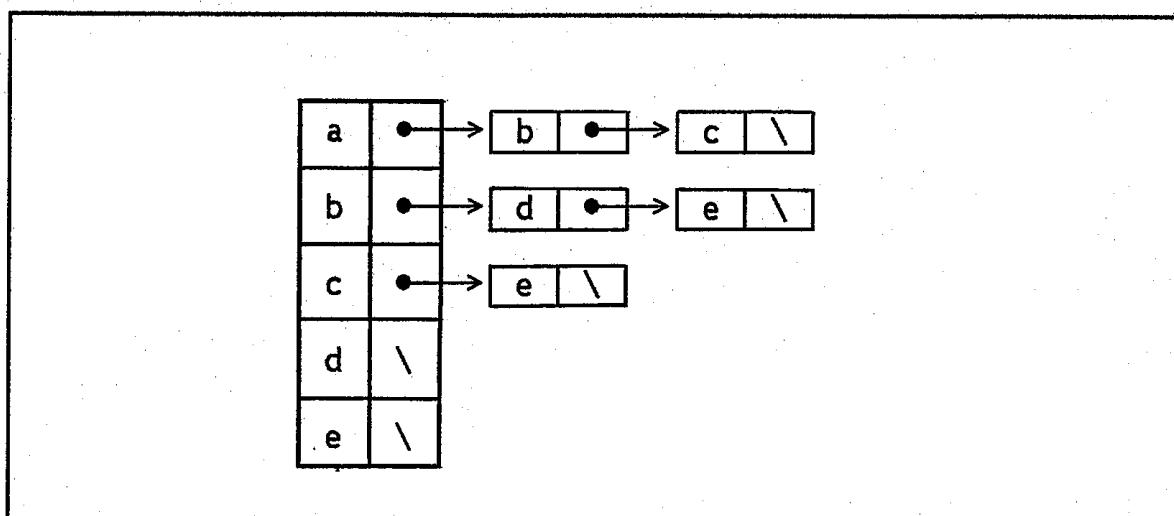
```

Em Pascal, dfs pode ser expressa de forma conveniente como um procedimento recursivo, em que os nomes dos nós são impressos na ordem desejada.

```
(*  
  Pesquisa em Profundidade em Pascal  
*)  
procedure dfs(x: no);  
  var y: no;  
  l: listanos;  
begin  
  writeln(x);  
  l := ladj(x);  
  while escolheadj(l,y) do  
    dfs(y)  
end;
```

Como em Prolog e LISP, a chamada recursiva efetua o aprofundamento no grafo. A exploração de nós adjacentes alternativos é feita agora por uma estrutura de controle iterativa (comando `while`.).

Uma estrutura de dados, ilustrada na figura abaixo, implementa a lista de adjacências.



A função `ladj`, como em LISP, retorna a lista dos nós adjacentes ao nó indicado:

```

function ladj(x: no): listanos;
var i: 1..nnos;
begin
  for i := 1 to nnos do
    if g[i].nomeno = x
      then ladj := g[i].segnos
end;

```

Bem mais complexo é o caso de `escolheadj`, que é uma função com efeitos colaterais. Se a lista `l` de nós adjacentes dada é vazia, a função retorna "false". Caso contrário, atribui a seu segundo argumento, `y`, o nome do primeiro nó da lista e o retira dela (ou seja, reduz a lista `l` à sua continuação), já preparando sua chamada seguinte dentro do escopo do `while`, e retorna `true`.

```

function escolheadj(var l: listanos; var y: no): boolean;
begin
  if l = nil then escolheadj := false
  else begin
    y := l↑.nomeno;
    l := l↑.segnos;
    escolheadj := true
  end
end;

```

Segue-se o programa em que se inserem esses procedimentos e funções, as declarações dos tipos de dados e a chamada ao procedimento `dfs`. Deixaremos de incluir a construção do grafo `g`, sobre o qual será aplicado `dfs`; essa tarefa poderia ser feita por procedimentos adicionais cujo detalhamento não é relevante aqui.

```
program p(input,output);
const nnos = 5;
type no = char;
listanos = ^elemento;
elemento = record
    nomeno: no;
    segnos: listanos
end;
grafo = array[1..nnos] of elemento;
var g: grafo;
t,v: listanos;
function ladj ...
...
end;
function escolheadj ...
...
end;
procedure dfs ...
...
end;
begin
... construcao do grafo g ...
dfs(a)
end.
```

Estamos agora em condições de efetuar uma comparação entre os três programas para dfs.

Natureza de dfs

- em Prolog: predicado
- em LISP : função
- em Pascal: procedimento

Representação dos dados

- em Prolog: cláusulas básicas
- em LISP : lista como argumento de função
- em Pascal: estrutura de dados

Forma do resultado

- em Prolog: valores assumidos por variável da consulta
- em LISP : lista retornada como valor da função
- em Pascal: valores impressos pelo procedimento

Exploração de alternativas

- em Prolog: retrocesso
- em LISP : recursão
- em Pascal: iteração

Argumentos no momento da chamada

- em Prolog: constantes e variáveis
- em LISP : constantes
- em Pascal: constantes e endereços de variáveis

Quanto ao último item, cabe notar a diferença entre Prolog e LISP, proveniente do mecanismo de unificação do Prolog, permitindo que ambos os termos a serem unificados contenham variáveis, enquanto que em LISP ocorre a substituição de parâmetros por constantes (mecanismo herdado do lambda-cálculo). Em Pascal, temos duas formas de passagem de parâmetro - por valor ou por endereço - permitindo esta a execução de atribuições para modificar valores de variáveis.

Talvez a diferença mais interessante entre Prolog e as duas outras linguagens seja a forma de definir os predicados através de um número arbitrário de cláusulas separadas, contrastando com a definição monolítica das funções e procedimentos de LISP e Pascal. Essa forma, que é originária da própria notação de cláusulas, assemelha-se também às *regras de produção*, que constituem um estilo de uso comum em Inteligência

Artificial, tendo ainda influenciado linguagens de programação como o SNOBOL. A separação das cláusulas aumenta a facilidade de estender, reduzir ou modificar as definições e, por outro lado torna o controle da execução menos estruturado.

Nos programas que mostramos em LISP, e mais ainda em Pascal, vimos a intromissão crescente de detalhes referentes à execução do algoritmo ("como fazer", ao invés de apenas "o que fazer"). Procuramos esconder ao máximo esses detalhes, colocando-os, na medida do possível, fora do programa principal (constituído pelas funções dfs e dfs1 em LISP, e pelo procedimento dfs em Pascal). Essa idéia de esconder detalhes se relaciona com o conceito de *tipos abstratos de dados*. Podemos considerar que funções como ladj, ladj1 em LISP e ladj e escolheadj em Pascal implementam operações sobre o tipo abstrato *grafo*. Essa associação de um tipo de dados com suas operações é mais nítido em linguagens como MODULA (extensão do Pascal), CLU, MESA, ALPHARD, etc.

NOTAS BIBLIOGRÁFICAS

O problema de geração de planos é tratado em Nilsson [1982] e Warren [1974]. A segunda referência contém um algoritmo bastante geral, capaz de gerar planos que atendam a uma conjunção de objetivos que interfiram entre si, caso em que o algoritmo tem de encontrar uma ordem em que os objetivos possam ser atingidos. Bancos de dados temporais são tratados em Bubenko [1977] e Kowalski e Sergot [1985], sendo que a primeira referência introduz (sem essa denominação) organizações orientadas para a atualização. Bancos de Dados e Programação em Lógica são relacionados em Smith [1986]. O sistema SQL/DS é descrito no manual SQL/Data System Application Programming (IBM [1983]), e a interface com VM/Prolog em Chang e Walker [1986]. O fundamento teórico e os aspectos metodológicos de nossa abordagem estão em Furtado e Neuhold [1986] e Veloso e Furtado [1985].

APÊNDICE I: COMANDOS EXTRA-LÓGICOS E OPERADORES PREDEFINIDOS DE PROLOG

Este apêndice resume alguns comandos extralógicos e operadores predefinidos da linguagem IBM PROLOG, que corre sob o sistema operacional VM (VM/Prolog (IBM [1985])) ou MVS (MVS/Prolog (IBM [1986])). Estes programas são similares apresentando essencialmente a mesma sintaxe e os mesmos comandos e operadores predefinidos. Uma lista completa dos comandos destes dialetos Prolog pode ser encontrada nos manuais correspondentes.

Estes comandos e operadores estendem a linguagem Prolog básica para permitir comunicação, gerenciamento da área de trabalho, controle de execução, processamento de cláusulas, depuração de programas e implementação da regra de negação por falha finita. Permitem também efetuar as operações aritméticas usuais, operações de comparação e de identificação de estruturas arbitrárias.

Os comandos podem ser inseridos normalmente em uma cláusula no papel de fórmulas atômicas e os operadores predefinidos podem ser codificados apenas dentro das expressões computáveis. Eles serão apresentados através de tabelas que descrevem a sintaxe de cada comando ou operador, seguida de um resumo da interpretação pretendida e de sinônimos em alguns casos. O manual do particular dialeto Prolog considerado deverá ser consultado para acomodar as diferenças sintáticas existentes em relação aos comandos e operadores apresentados.

Para facilidade de referência diremos "símbolo funcional" (abreviado a seguir por "sfun") em substituição à "átomo no papel de um símbolo funcional" e "símbolo predicativo" (abreviado a seguir por "spred") em substituição à "átomo no papel de um símbolo predicativo".

Abreviações Adotadas

ATP	Área de Trabalho Prolog
cadc	Cadeia de caracteres
car	Carácter
dig	Dígito
exp	Expressão
expc	Expressão computável
lcar	Lista de caracteres
ldig	Lista de dígitos
nint	Número inteiro
nflu	Número em ponto flutuante
num	Número
sfun	Símbolo funcional
spred	Símbolo predicativo
sss	Se e somente se
tfun	Termo funcional
var	Variável
vv	Vice-versa

Símbolos Sintáticos nos Diversos Dialetos Prolog

<i>Adotado</i>	<i>Outros Dialetos</i>	<i>Significado</i>
\wedge	,	Conectivo lógico "e": conjunção
$ $;	Conectivo lógico "ou": disjunção
\neg	not ou \neg	Conectivo lógico "não": negação
\leftarrow	$:$ ou if	Conectivo lógico "se": implicação
\leftarrow	$:$ ou ?	Caracteriza cláusula objetivo Prolog
*	-	Variável anônima Prolog
/	!	Comando extralógico "corte"
;	;	Comando para solicitar solução alternativa
.	.	Indica término de cláusula Prolog
.	ou !	Símbolo funcional construtor de listas
	!!	Operador de concatenação

Comandos Aritméticos	
sum(X,Y,Z)	Tenta resolver a equação $Z = X + Y$
diff(X,Y,Z)	Tenta resolver a equação $Z = X - Y$
prod(X,Y,Z)	Unifica Z com o produto de X e Y
quot(X,Y,Z)	Unifica Z com o quociente de X e Y
rem(X,Y,Z)	Unifica Z com o resto da divisão de X por Y

Nota: X, Y e Z devem ser números com X e Y previamente instanciados nos comandos prod, quot e rem.

Comandos de Comparação	
eq(X,Y)	Retorna SUCESSO sss $X = Y$
ne(X,Y)	Retorna SUCESSO sss $X \neq Y$
le(X,Y)	Retorna SUCESSO sss $X \leq Y$
lt(X,Y)	Retorna SUCESSO sss $X < Y$
ge(X,Y)	Retorna SUCESSO sss $X \geq Y$
gt(X,Y)	Retorna SUCESSO sss $X > Y$

Nota: X e Y devem ser constantes Prolog

Comandos de Comparação de Termos	
X = Y	Retorna SUCESSO sss X unificar com Y
X =/ Y	Retorna SUCESSO sss X não unificar com Y
X == Y	Retorna SUCESSO sss X for idêntico a Y
X ==/ Y	Retorna SUCESSO sss X não for idêntico a Y
X =*= Y	Retorna SUCESSO sss for possível renomear as vars de X e Y de modo que X e Y tornem-se idênticos
X =:= Y	Retorna SUCESSO sss X e Y forem árvores finitas e se X unificar com Y com o teste de ocorrência (requisição de teste de ocorrência)

Operadores Válidos em Expressões Computáveis	
X - Y	Retorna a diferença entre X e Y
X + Y	Retorna a soma de X e Y
X * Y	Retorna o produto de X e Y
X / Y	Retorna o quociente de X e Y
max(X,Y)	Retorna o máximo entre X e Y
min(X,Y)	Retorna o mínimo entre X e Y
- X	Retorna a negação de X
abs(X)	Retorna o valor absoluto de X
sin(X)	Retorna o seno de X (X em radianos)
cos(X)	Retorna o cosseno de X (X em radianos)
exp(X)	Retorna o número e elevado a X
log(X)	Retorna o logaritmo de X na base e
pi()	Retorna o número π
len(X)	Retorna o comprimento da avaliação de X
X Y	Retorna a concatenação de X com Y
upper(X)	Retorna a avaliação de X em letras maiúsculas
lower(X)	Retorna a avaliação de X em letras minúsculas
strip(X)	Retorna a avaliação de X sem brancos
substring(X,Y,Z)	Retorna a subcadeia resultante da avaliação de X, iniciando na posição Y com comprimento Z

Comandos de Comunicação	
nl	Encerra a linha de impressão e posiciona para impressão em uma nova linha da saída corrente
tab(C)	Posiciona para impressão na coluna C da saída corrente
read(T)	Lê o termo seguinte, seguido de ponto, na entrada corrente e unifica o mesmo com T
readch(C)	Lê o car seguinte na entrada corrente e unifica o mesmo com C
write(U)	Imprime exp U, seguida de ponto, e posiciona para impressão em uma nova linha da saída corrente
writes(U)	Imprime exp U e posiciona para impressão na coluna seguinte da saída corrente
writex(U)	Imprime exp U e posiciona para impressão na coluna seguinte da saída corrente (átomos são impressos sem aspas)
prst(X)	Imprime a cadeia resultante da avaliação da expc X e posiciona para impressão na coluna seguinte da saída corrente (constantes são impressas sem apóstrofos)

Nota: Estes comandos de comunicação não são passíveis de retrocesso.

Comandos de Gerenciamento da Área de Trabalho Prolog (ATP)	
new	Restaura o estado da ATP para o estado de inicialização (só primitivas Prolog)
dump	Lista todas cláusulas da ATP
save(X)	Armazena o estado corrente da ATP no arquivo de nome X
load(X)	Limpa a ATP e carrega o arquivo de nome X (criado através de comando "save")
consult(X)	Carrega acumulativamente na ATP as cláusulas Prolog armazenadas no arquivo de nome X
reconsult(X)	Carrega na ATP as cláusulas Prolog armazenadas no arquivo de nome X eliminando previamente todas cláusulas de prefixo X
edconsult(X)	Abre o arquivo X para edição na ATP e em seguida carrega na ATP as cláusulas Prolog armazenadas no arquivo de nome X eliminando previamente todas cláusulas de prefixo X (como "reconsult" mas com capacidade de edição)
fin	Encerra a ativação do interpretador Prolog voltando o controle para o sistema monitor

Comandos de Controle	
/	Comprometimento com as substituições efetuadas até a fórmula atômica imediatamente anterior (corte)
fail	Provoca retrocesso imediato
true	Retorna SUCESSO (FALHA em retrocesso)
repeat	Retorna SUCESSO repetidamente

Comandos de Processamento de Cláusulas	
call(C)	Avalia a cláusula C retorna SUCESSO se C for satisfeita
addax(C)	Adiciona a cláusula C na ATP após todas cláusulas sobre o mesmo átomo no mesmo papel que o da cláusula C
addax(C,N)	Adiciona a cláusula C na ATP na N-ésima posição entre as cláusulas sobre o mesmo átomo no mesmo papel que o da cláusula C
delax(H)	Remove da ATP a primeira cláusula definida cuja cabeça unifica com a cláusula unitária H
delax(H,N)	Remove da ATP a N-ésima cláusula definida cuja cabeça unifica com a cláusula unitária H
ax(H,R)	Recupera em R a primeira cláusula definida da ATP cuja cabeça unifica com a cláusula unitária H
ax(H,R,N)	Recupera em R a N-ésima cláusula definida da ATP cuja cabeça unifica com a cláusula unitária H
axn(P,A,R)	Recupera em R a primeira cláusula da ATP sobre o átomo P no papel de um spred de aridade A
axn(P,A,R,N)	Recupera em R a N-ésima cláusula da ATP sobre o átomo P no papel de um spred de aridade A

Nota: os comandos "addax" e "delax" não são passíveis de retrocesso.

Comandos de Depuração	
sp	suspende execução do programa (comando imediato)
break(p(*,..))	Ativa o procedimento que permite parar a execução da fórmula atômica p(*,..) com aridade definida pelo total de *'s
break(p(*,..),on)	Idêntico comando anterior (ativa break)
break(p(*,..),off)	Desativa break para p(*,..)
break(p(*,..),X)	Retorna em X estado do break para p(*,..)
spy(p(*,..))	Ativa o procedimento que permite coletar estatísticas da fórmula atômica p(*,..) com aridade definida pelo total de *'s
spy(p(*,..),on)	Idêntico comando anterior (ativa spy)
spy(p(*,..),off)	Desativa spy para p(*,..)
spy(p(*,..),X)	Retorna em X estado do spy para p(*,..)
spy_display	Lista tabela com estatísticas das fórmulas atômicas ativadas através de spy
trace(p(*,..))	Ativa o procedimento que permite rastrear a fórmula atômica p(*,..) com aridade definida pelo total de *'s
trace(p(*,..),on)	Idêntico comando anterior (ativa trace)
trace(p(*,..),off)	Desativa trace para p
trace(p(*,..),X)	Retorna em X estado do trace para p

Comandos de Identificação de Tipos de Dados	
atom(U)	Retorna SUCESSO sss U é átomo
atomic(U)	Retorna SUCESSO sss U é átomo ou número
charp(U)	Retorna SUCESSO sss U é car ou dig entre 0 e 9
digit(N)	Retorna SUCESSO sss N é dig entre 0 e 9
letter(C)	Retorna SUCESSO sss C é um único car
inf(X)	Retorna SUCESSO sss X é árvore infinita
noinf(X)	Retorna SUCESSO sss X não é árvore infinita
int(U)	Retorna SUCESSO sss U é nint
floatp(U)	Retorna SUCESSO sss U é nflu
numb(U)	Retorna SUCESSO sss U é num
skel(F)	Retorna SUCESSO sss F é tfun
stringp(U)	Retorna SUCESSO sss U é cadc
var(U)	Retorna SUCESSO sss U é var não-instanciada

Comandos de Processamento de Termos	
arg(I,F,X)	Retorna SUCESSO sss o I-ésimo argumento do tfun F unificar com X
arg(0,F,X)	Retorna SUCESSO sss o sfun do tfun F unificar com X
F = .. L	Converte a lista L no tfun F ou vv. O primeiro elemento de L é o sfun de F e os demais elementos de L são os argumentos de F. F e/ou L devem ser instanciados (sinônimo: cons(F,L)).

Comandos de Conversão	
st_to_at(C,A)	Converte cada C para átomo ou número A e vv C ou A devem ser instanciados
fl_to_int(Z,I)	Converte nflu Z para nint N e vv Z ou N devem ser instanciados
st_to_li(C,L)	Converte cada C em lcar ou ldig L e vv C ou L devem ser instanciados
st_to_nb(C,N)	Converte cada C para num N e vv C ou N devem ser instanciados

Comandos de Processamento de Cadeia de Caracteres	
stconc(X,Y,Z)	Unifica Z com concatenação de X e Y
stlen(X,Y)	Unifica Y com o comprimento da cadeia X
substring(C,S,I,L)	Unifica S com uma subcadeia da cadeia C com início no índice I e comprimento L

Outros Comandos e Operadores Extra-Lógicos	
copy(E1,E2)	Copia exp E1 em exp E2 renomeando vars de E2
listvar(E,CJ)	Unifica CJ com o conjunto das vars da exp E
op(A,T,P)	Declara átomo A como um operador do tipo T com prioridade P
rlen(F,A)	Retorna SUCESSO se A unificar com a aridade do tfun F
skel(AT,A,F)	Constrói tfun F com sfun AT e aridade A ou retorna sfun AT e aridade A do tfun F
random(M,N,X)	Unifica X com um número aleatório entre os inteiros M e N, com M < N
X := E	Avalia expc E (obtendo cada e/ou num) e unifica o resultado com X
¬X	Retorna FALHA se E não for expc (sinônimo: "is") Retorna SUCESSO sss X retornar FALHA Retorna FALHA sss X retornar SUCESSO

Nota: o comando "copy" é útil para visualizar unificações viáveis de variáveis em expressões, sem ser necessário instanciá-las.

APÊNDICE II: EXERCÍCIOS DE PROLOG

II.1 ENUNCIADO DOS EXERCÍCIOS

Este apêndice enuncia exercícios a serem resolvidos usando-se a linguagem Prolog. Observe que, de um modo geral, os exercícios são passíveis de terem mais de uma solução.

1. Sabe-se que:

Ismael e Isac descendem de Abraão
Esaú e Jacó descendem de Isac

Criar uma Base de Fatos Prolog com estes dados acrescida com as cláusulas não-unitárias Prolog que permitam responder às questões:

- a) Quais são os descendentes de Abraão?
- b) Quem é descendente de quem?

2. Sabe-se que:

País	População (milhões hab.)	Área (km ²)
Brasil	130	9
Estados Unidos	230	9
India	548	3
China	800	12

Criar uma Base de Fatos Prolog com estes dados acrescida por uma cláusula não-unitária Prolog que permita calcular a densidade populacional dos países considerados.

3. Escrever um programa Prolog, que permita responder a uma questão, em um certo contexto, com uma sentença vaga, porém relativa a questão efetuada. Exemplos:

questão: voce gosta de estudar?

resposta: eu não gosto de estudar!

questão: voce fala inglês?

resposta: eu falo português!

questão: voce normalmente diz bom dia?

resposta: eu normalmente não digo bom dia!

questão: voce é um elefante?

resposta: eu não sou um elefante!

4. Dados os seguintes eventos históricos:

Data Evento

1500 Descobrimento do Brasil

1822 Independência do Brasil

1889 Proclamação da República

1960 Inauguração de Brasília

1964 Início do Governo Militar

1985 Início da Nova República

1986 Início do Plano de Estabilização Econômica

1986 Término do Plano de Estabilização Econômica

1987 Moratória Técnica dos Juros da Dívida Externa

Criar uma Base de Fatos Prolog com estes dados e efetuar consultas sobre a mesma.

5. Escrever uma única cláusula não-unitária Prolog que permita calcular o valor médio entre 2 valores.

6. Escrever um programa Prolog que permita calcular o máximo divisor comum de 2 números inteiros dados.

7. Construir uma Base de Fatos usando o seguinte vocabulário:

Nome de objetos:	Londres	França
	Paris	Italia
	Roma	Nigeria
	Lagos	Europa
	Reino Unido	Africa
	Angola	Luanda

Nomes de relações: `capital_de`, `pais_em`

Expressar, sobre esta Base de Fatos, a seguinte consulta:

- a) Que capitais estão na Europa? e na África? e na Ásia?
 - b) Que pares de cidades são capitais de países na Europa?
 - c) Que pares de cidades são capitais de países em um mesmo continente?
8. Definir uma fórmula atômica, cujo símbolo predicativo tenha nome VERIFICA, com 2 argumentos: nome do estudante e nome do professor. Esta fórmula atômica deve permitir verificar a existência de um professor que lecione dois diferentes cursos a um mesmo estudante em uma mesma sala de aula.

A Base de Fatos deve conter (invente dados para que tal ocorra):

- a) Informações sobre estudantes e seus cursos definidas por fórmulas atômicas da forma:
`estudante(Nome_do_estudante,Nome_do_curso)`
- b) Informações de professores e seus cursos definidas por fórmulas atômicas da forma:
`professor(Nome_do_professor,Nome_do_curso)`
- c) Informações de dias da semana e de salas onde ocorrem os cursos, definidas por fórmulas atômicas da forma:
`curso(Nome_do_curso,Dia_da_semana,Nome_da_sala)`

9. Considerando as operações "consec" e "concat" conforme definidas na seção 10.5, desenhar as árvores de refutação completas para as consultas:

a) $\leftarrow \text{consec}(2, X, 1 . 2 . 3 . 2 . 4 . \text{nil}).$

b) $\leftarrow \text{concat}(X, Y, \text{a.b.c.d.nil}).$

10. Escrever uma operação Prolog que permita obter a soma dos N primeiros números naturais.

11. Escrever as operações Prolog para imprimir uma lista nas seguintes condições:

a) um elemento por linha.

b) todos elementos em uma mesma linha (para listas com 5 elementos no máximo).

c) lista com sublistas: um elemento por linha com cada sublista impressa com um deslocamento de 3 posições para a direita.

12. Escrever uma operação Prolog que permita encontrar um elemento com propriedade p em uma lista dada.

13. Escrever uma operação Prolog que permita encontrar o primeiro elemento com propriedade p em uma lista dada.

14. Escrever uma operação Prolog que permita encontrar o n-ésimo elemento de uma lista dada.

15. Escrever uma operação Prolog que permita calcular o comprimento de uma lista.

16. Escrever uma operação Prolog que implemente a concatenação de uma mesma lista 3 vezes.

17. Escrever uma operação Prolog que implemente a concatenação de 3 listas.

18. Escrever uma operação Prolog, em função da operação "concat", que implemente a inversão de uma lista dada.

19. Escrever uma operação Prolog que implemente a permutação de uma lista dada.
20. Escrever uma operação Prolog que implemente a inserção de um elemento dado em uma lista classificada dada, em posição tal que a classificação da lista seja mantida.
21. Escrever uma operação Prolog que permita remover todas ocorrências de um elemento em uma lista.
22. Escrever uma operação Prolog que permita verificar se um termo dado é membro de uma lista dada ou de qualquer sublistas da lista dada.
23. Escrever uma operação Prolog que permita particionar uma lista em duas sublistas de aproximadamente mesmo comprimento.
24. Escrever uma operação Prolog que permita obter o valor máximo de uma lista numérica.
25. Escrever uma operação Prolog que permita obter a soma dos elementos de uma lista numérica.
26. Escrever uma operação Prolog que permita particionar uma lista dada em duas, tais que os elementos da primeira sejam menor ou igual a um número n dado e os elementos da segunda sejam maior que n , preservando a ordem original dos elementos da lista.
27. Escrever um programa Prolog que permita extrair todas variáveis de um termo, juntando-as em uma lista.
28. Escrever um programa Prolog que permita verificar se um conjunto é subconjunto de outro.
29. Escrever um programa Prolog que permita verificar se a intersecção de dois conjuntos não é vazia.
30. Escrever um programa Prolog que permita verificar se dois conjuntos são disjuntos.
31. Escrever um programa Prolog que permita verificar se dois conjuntos são iguais.

32. Escrever um programa Prolog que contenha definidas as principais operações sobre conjuntos.
33. Escrever um programa Prolog que permita calcular a N-ésima potência de um número.
34. Escrever um programa Prolog que permita encontrar um caminho entre dois nós de um grafo orientado.
35. Simule, utilizando metavariables, a conjunção, a disjunção, a implicação e o quantificador existencial.
36. Implemente métodos de ordenação em Prolog.

II.2 SOLUÇÃO DE EXERCÍCIOS SELECIONADOS

Este seção apresenta a solução de alguns dos exercícios enunciados no Apêndice II.1, codificados na linguagem IBM Prolog (VM/Prolog [IBM 1985] ou MVS/Prolog [IBM 1986]) adotando-se a notação de Marseille para listas. Com pequenas alterações de sintaxe, as codificações apresentadas podem ser convertidas para outros dialetos Prolog.

/*

Ex. 1: descendencia

Formulas atomicas adotadas / interpretações:

descende(X,Y): X descende de Y

descendente(X,Y): X é descendente de Y

Obs:

a. system clear): comando que permite limpar a tela

b. Execução (observação válida também para exercícios 4 e 7):

<- q1a.

<- q1b.

ou diretamente com:

<- corpo da cláusula q1a

<- corpo da cláusula q1b

*/

descende(ismael,abraao).

descende(isac,abraao).

```
descende(esau, isac).
descende(jaco, isac).
```

```
descendente(X,Y) <- descende(X,Y).
descendente(X,Y) <- descendente(X,Z) & descendente(Z,Y).
```

```
q1a <- system(clear) &
      descendente(X,abraao) &
      writes(X) & nl & fail.
```

```
q1b <- system(clear) &
      descendente(X,Y) &
      tab(10) & writes(X) &
      tab(20) & writes(Y) & nl & fail.
```

/*

Ex. 2: calculo de densidade populacional

Formulas atomicas adotadas / interpretacoes:

pop(X,Y): a populacao de X e' Y

area(X,Y): a area de X e' Y

densidade(X,Y): a densidade de X e' Y

*/

```
pop(brasil,108).
```

```
pop(usa,203).
```

```
pop(india,548).
```

```
pop(china,800).
```

```
area(brasil,9).
```

```
area(usa,9).
```

```
area(india,3).
```

```
area(china,12).
```

```
densidade(X,Y) <- pop(X,P) & area(X,A) & quot(P,A,Y).
```

```
q2 <- system(clear) &
      densidade(X,Y) &
      prst('A densidade populacional de: ') &
      writes(X) & prst(' e'' ') & writes(Y) &
      nl & fail.
```

/*

Ex. 3: resposta a uma questão com uma sentença relativa
 Formulas atomicas adotadas / interpretacoes:
 altere(E1,E2): argumento E2 é o argumento E1 alterado
 responda(L1,L2): lista L2 é a resposta para lista L1
 */

```

altere(voce,eu).
altere(fala,falo).
altere(e,nao.sou).
altere(diz,nao.digo).
altere(gosta,nao.gosto).
altere(frances,ingles).
altere(ingles,portugues).
altere(portugues,alemao).
altere(alemao,latin).
altere(latin,russo).
altere(russo,japones).
altere(japones,castelhano).
altere(castelhano,grego).
altere(grego,frances).
altere(?,?).
altere(X,X).
responda(nil,nil).
responda(H,T,X,Y) :- altere(H,X) & responda(T,Y).
```

/*

Ex. 4: eventos históricos
 Formula atomica adotada / interpretacao:
 evento(Ano,Fato): evento Fato ocorreu no ano Ano
 */

```

evento(1500,'Descobrimento do Brasil').
evento(1822,'Independencia do Brasil').
evento(1889,'Proclamacao da Republica').
evento(1960,'Inauguracao de Brasilia').
evento(1964,'Inicio do Governo Militar').
evento(1985,'Inicio da Nova Republica').
evento(1986,'Inicio do Plano de Estabilizacao Economica').
evento(1986,'Fim do Plano de Estabilizacao Economica').
evento(1987,'Moratoria Tecnica dos Juros da Dvida Externa').
evento(*,desconhecido).
```

/* exemplo: que evento importante ocorreu em 1889? */

```

q4a <- evento(1889,X) & writes(X) & nl.

/* entrada do ano do evento pelo terminal via dialogo */
q4b <- prst('Entre com o ano do evento: ') & nl &
      read(X) & nl & evento(X,Y) &
      writes(Y) & nl.

/*
Ex. 5: valor medio entre dois numeros
Formula atomica adotada / interpretacao:
  valor_medio(X,Y): valor medio de X e Y
*/
valor_medio(X,Y) <- prst('O valor medio entre '
                          '|| X || ' e ' || Y || ' e' '
                          '|| (X + Y)/2) &
                          nl.

/*
Ex. 6: maximo divisor comum de 2 numeros inteiros
Formula atomica adotada / interpretacao:
  mdc(X,Y,D): maximo divisor comum entre X e Y e' D
                (X e Y inteiros positivos)
  1. Se X = Y, D= X
  2. Se X < Y, D= mdc entre X e Y1 = Y - X
  3. Se X > Y, D= mdc entre Y e X, conforme 2
*/
mdc(X,X,X) <- / .
mdc(X,Y,D) <- lt(X,Y) & / &
               Y1 := Y - X &
               mdc(X,Y1,D).
mdc(X,Y,D) <- mdc(Y,X,D).

/*
Ex. 7: relacoes entre paises e cidades
Formulas atomicas adotadas / interpretacoes:
  capital_de(Cidade,Pais): Cidade e' capital de Pais
  pais_em(Pais,Continente): Pais esta em Continente

```

*/

```

capital_de(londres,reino_unido).
capital_de(paris,franca).
capital_de(roma,italia).
capital_de(lagos,nigeria).
capital_de(luanda,angola).

pais_em(reino_unido,europa).
pais_em(franca,europa).
pais_em(italia,europa).
pais_em(nigeria,africa).
pais_em(angola,africa).

q7aa <- capital_de(X,Y) & pais_em(Y,europa) &
      writes(X) & nl & fail.

q7ab <- capital_de(X,Y) & pais_em(Y,africa) &
       writes(X) & nl & fail.

q7ac <- capital_de(X,Y) & pais_em(Y,asia) &
       writes(X) & nl & fail.

q7b <- capital_de(X,Y) & capital_de(Z,W) &
      pais_em(Y,europa) & pais_em(W,europa) &
      ne(X,Z) &
      tab(10) & writes(X) & tab(20) & writes(Z) & nl & fail.

q7c <- capital_de(X,Y) & capital_de(Z,W) &
      pais_em(Y,Continente) & pais_em(W,Continente) &
      ne(X,Z) &
      tab(10) & writes(X) & tab(20) & writes(Z) & nl & fail.

```

/*

Ex. 10: somatoria de 1 a N
 Formula atomica adotada / interpretacao:
 somat(N,SOMA): soma dos N primeiros inteiros

*/

```

somat(N,1)    <- 1e(N,1) & /.
somat(N,SOMA) <- diff(N,1,N1)      &
                  somat(N1,SOMA1)    &
                  sum(N,SOMA1,SOMA).

```

/*

Ex. 11: impressao de listas

a) um elemento por linha - solucao 1

Formula atomica adotada / interpretacao:

impr_lista(L,N): imprime Lista L com Deslocamento N

*/

impr_lista(nil).

impr_lista(H,T) <- writes(H) & nl & impr_lista(T).

/* exemplo: <-impr_lista(a.b.c.d.nil). */

/*

a) um elemento por linha - solucao 2

Obs: operacao membro definida a seguir

exemplo:

<-membro(X,a.b.c.d.nil) & writes(X) & nl & fail.

*/

/*

b) todos elementos impressos em uma mesma linha

(para listas com 5 elementos no maximo)

Formula atomica adotada / interpretacao:

imprime(L,N): imprime Lista L com Deslocamento N

*/

imprime(nil,*).

imprime(H,T,N) <- tab(N) & writes(H) &

sum(N,5,M) &

imprime(T,M).

/* exemplo: <-imprime(a.b.c.d.nil,0). */

/*

c) lista com sublistas (impressas indentadas)

Formula atomica adotada / interpretacao:

pp(L,N): imprime Lista L com Deslocamento N
 $*/$

pp(H,T,I) \leftarrow sum(I,3,J) & pp(H,J) & ppx(T,J).
 pp(X,I) \leftarrow tab(I) & writes(X) & nl.

ppx(nil,*).
 ppx(H,T,I) \leftarrow pp(H,I) & ppx(T,I).

$/*$ exemplo: \leftarrow pp(a.b.(b1.b2.nil).c.(c1.c2.nil).d.nil) $*/$

$/*$

Ex. 12: elemento com propriedade p em uma lista
 Formulas Atomicas adotadas / interpretacoes:
 elem_propriedade(E,L): E e' um elemento da lista L
 com propriedade p
 p(Prop): p e' uma propriedade Prop
 $*/$

elem_propriedade(H,H,*) \leftarrow p(H).
 elem_propriedade(H,* .T) \leftarrow elem_propriedade(H,T).

$/*$ 2a. solucao: usa "membro" definido a seguir $*/$

elem_propriedade_s2(H,Lista) \leftarrow membro(H,Lista) & p(H).

$/*$

Ex. 13: 1o. elemento com propriedade p em uma lista
 Formulas atomicas adotadas / interpretacoes
 elem_propriedade(E,L): E e' o 1o. elemento da lista L
 com propriedade p
 p(Prop): p e' uma propriedade Prop
 $*/$

elem1_propriedade(H,H,*) \leftarrow p(H) & /.
 elem1_propriedade(H,* .T) \leftarrow elem1_propriedade(H,T).

$/*$ 2a. solucao: usa "membro" definido a seguir $*/$

elem1_propriedade_s2(H,Lista) \leftarrow membro(H,Lista) & p(H) & /.

/*

Ex. 14: n-esimo elemento de uma lista dada

Formula atomica adotada:

elem_n(Num_elemento,Lista_dada,Nesimo_elemento)

*/

elem_n(1,X, * ,X).

elem_n(N, * ,Y,X) <- elem_n(M,Y,X) & sum(1,M,N).

/*

Ex. 15: comprimento de uma lista

Formula atomica adotada / interpretacao:

cpto(L,N): comprimento da lista L e' N

*/

cpto(nil,0).

cpto(H,T,N) <- cpto(T,N1) &

N := N1 + 1.

/*

Ex. 16: concatenar 3 vezes uma mesma lista

Formula atomica adotada / interpretacao:

triplo(L,LLL): LLL e' a concatenacao de L tres vezes

*/

triplo(L,LLL) <- concat(L,L,LL) & concat(L,LL,LLL).

concat(nil,L,L).

concat(X,L1,L2,X,L3) <- concat(L1,L2,L3).

/*

Ex. 17: concatenar 3 listas

Obs: usa "concat" ja definido anteriormente

Formula Atomica adotada / interpretacao:

concat3(L1,L2,L3,L): L e' a concatenacao de L1,L2 e L3

*/

concat3(L1,L2,L3,L) <- concat(L1,L2,LL) & concat(LL,L3,L).

/*

Ex. 18: inverte os elementos de uma lista

Obs: usa "concat" ja definido anteriormente
Formula atomica adotada / interpretacao:
ninv(L1,L2): L2 e' a lista inversa de L1

* /

ninv(nil,nil).

`ninv(X,L1,L) :- ninv(L1,L2) & concat(L2,X,nil,L).`

/* 2a. solucao (sem usar concat - mais eficiente) */

```
inv(X,Y) :- invertir(X,nil,Y).
```

inverte(nil,L,L).

inverte(X. L1, LA, L) :- inverte(L1, X. LA, L).

/*

Ex. 19: permuta os elementos de uma lista

Obs: usa "concat" ja definido anteriormente

Formula atomica adotada / interpretacao:

`permuta(L1,L2)`: L2 e' uma permutacao da lista L1

* /

permuta(nil,nil).

permuta(L,H,T) :-

concat(V,H,U,L) &

`concat(V,U,W)` &

permuta(W,T).

14

Ex. 20: insere elemento em lista classificada

Formula atomica adotada / interpretacao:

insere(E,L1,L): L e' a lista L1 com E inserido de forma a manter a classificacao.

* /

insere(X, nil, X)

`insere(X,Y | X Y |)` \Leftarrow $X \leq Y$

insere(X, Y, L, E, X, Y, L) :- $X \neq Y$, ...
insere(X, Y, L1, Y, L2) :- **insere(X, L1, L2)**

/*

Ex. 22: verifica se um elemento e' membro de uma lista ou de qualquer sublistas de uma lista dada

Formula atomica adotada / interpretacao:

membrot(X,L): X e' um elemento de L ou de
qualquer sublistas de L

/*

membrot(X,X.*).

membrot(X,*.Y) <- membrot(X,Y).

membrot(X,(Y.Z).*) <- membrot(X,Y.Z).

/*

Ex. 23: particiona uma lista em duas sublistas de
aproximadamente mesmo comprimento

Formula atomica adotada / interpretacao:

divlista(L,L1,L2): L1 e L2 sao divisoes de
aproximadamente mesmo tamanho da lista L

/*

divlista(nil,nil,nil).

divlista(X.nil,X.nil,nil).

divlista(X.Y.L,X.L1,Y.L2) <- divlista(L,L1,L2).

/*

Ex. 24: elemento de maior valor em uma lista numerica

Formula atomica adotada / interpretacao:

maxlista(L,Max): Max e' o elemento de maior valor da lista L

/*

maxlista(X.Y.Z,Max) <- ge(X,Y) & maxlista(X.Z,Max).

maxlista(X.Y.Z,Max) <- maxlista(Y.Z,Max).

maxlista(X.Y.nil,X) <- ge(X,Y) & / .

maxlista(X.Y.nil,Y) <- / .

/*

Ex. 25: soma dos elementos de uma lista numerica

Formula atomica adotada / interpretacao:

somalista(L,S): S e' a soma dos elementos da lista L

/*

somalista(nil,0).

somalista(H.T,S) <- somalista(Y,S1) &
S := H + S1.

/*

Ex. 26: particionamento de uma lista

Formula atomica adotada / interpretacao:

particiona(N,L,L1,L2): lista L particiona-se em
L1 com elementos =
N e
L2 com elementos >
N

*/

particiona(N,H,T,H,U1,U2) <- le(H,N) & particiona(N,T,U1,U2).

particiona(N,H,T,U1,H,U2) <- particiona(H,T,U1,U2).

particiona(*,nil,nil,nil).

/*

Ex. 28: verifica se um conjunto e' subconjunto de outro

Obs: conjuntos definidos por listas

Formula atomica adotada / interpretacao:

subcjto(C1,C2): C1 e' sub-conjunto de C2

*/

subcjto(nil,C).

subcjto(E,C1,C2) <- membro(E,C2) & subcjto(C1,C2).

/*

Ex. 29: verifica se dois conjuntos se interceptam

Obs: conjuntos definidos por listas

Formula atomica adotada / interpretacao:

intercepta(C1,C2): C1 e C2 se interceptam

*/

intercepta(C1,C2) <- membro(E,C1) & membro(E,C2) & /.

/*

Ex. 30: verifica se dois conjuntos sao disjuntos

Obs: conjuntos definidos por listas

Formula atomica adotada / interpretacao:

disjunto(C1,C2): C1 e C2 sao conjuntos disjuntos

*/

```
disjunto(C1,C2) <- membro(E,C1) & ~membro(E,C2).
~X <- X & / & fail.
~X.
```

/* 2a. solucao: */

```
disjunto_s2(C1,C2) <- intercepta(C1,C2) & / & fail.
disjunto(*,*).
```

/*

Ex. 31: verifica igualdade de 2 conjuntos

Obs: conjuntos definidos por listas

Formulas atomicas adotadas / interpretacoes:

icjto(C1,C2) : conjuntos C1 e C2 sao iguais
 ilista(L1,L2): listas L1 e L2 sao iguais
 remove(X,L1,L2): remove elemento X da lista L1 dando
 como resultado lista L2

*/

```
icjto(X,X) <- /.
icjto(X,Y) <- ilista(X,Y).
```

ilista(nil,nil).

ilista(X,L1,L2) <- remove(X,L2,L3) & ilista(L1,L3).

remove(X,X,Y,Y).

remove(X,*.L1,*.L2) <- remove(X,L1,L2).

/*

Ex. 32: operacoes sobre conjuntos definidos por listas

Formulas atomicas adotadas / interpretacoes:

membro(E,C): E e' um elemento do conjunto C
 uniao(C1,C2,C): C e' uniao dos conjuntos C1 e C2
 inter(C1,C2,C): C e' a interseccao dos conjuntos C1 e C2

*/

```
membro(X,X,*).
membro(X,*.Y) <- membro(X,Y).
```

uniao(nil,C,C).

uniao(X,C1,C2,C3) <- membro(X,C2) & / & uniao(C1,C2,C3).

uniao(X,C1,C2,X,C3) <- uniao(C1,C2,C3).

```
inter(nil,C,nil).
inter(X,C1,C2,X,C3) <- membro(X,C2) & / & inter(C1,C2,C3).
inter(X,C1,C2,C3)  <- inter(C1,C2,C3).
```

/*

Ex. 33: N-esima potencia de um numero

Formula atomica adotada / interpretacao:

expo(X,N,R): R é X elevado a N (inteiro)

*/

```
expo(X,0,1) <- / .
expo(X,N,R) <- N1 := N - 1  &
               expo(X,N1,R1) &
               R := R1 * X .
```

REFERÊNCIAS BIBLIOGRÁFICAS

- Apt, K.R. e M.H. van Emden [1982].** "Contributions to the Theory of Logic Programming", *J. ACM* 29:3 (julho 1982), 841-862.
- Battani, G. e H. Méloni [1973].** "Interpréteur du language de programmation Prolog", Groupe d'Intelligence Artificielle, Université d'Aix-Marseille, Marseille, France.
- Bell, J.L. e M. Machover [1977].** *A Course in Mathematical Logic*, North-Holland Publishing Company.
- Bharath, R. [1986a].** "Expand your System's Knowledge Base", Computer Language (Agosto 1986).
- Bharath, R. [1986b].** *An Introduction to Prolog*, TAB Books, Inc.
- Bowen, D.L. (ed.) [1981].** *DECsystem-10 Prolog User's Manual*, Department of Artificial Intelligence, University of Edinburgh, Edinburgh, Scotland.
- Bowen, K.A. e R.A. Kowalski [1982].** "Amalgamating language and metalanguage in Logic Programming", em *Logic Programming*, K.L. Clark e S.-A. Tärnlund (eds.), Academic Press.
- Brainerd, W.S. e L.H. Landweber [1974].** *Theory of Computation*, John Wiley & Sons.
- Bratko, I. [1986].** *Prolog Programming for Artificial Intelligence*, Addison-Wesley Publishing Company.
- Bubenko, J.A., Jr. [1977].** "The temporal dimension in information modelling", em *Architectures and Models in Data Base Management Systems*, G.M. Nijssen (ed.), North-Holland Publishing Company, 93-118.
- Burnham, W.D. e A.R. Hall [1985].** *Prolog Programming and Applications*, Macmillan Publishing Company.

- Campbell, J.A. (ed.) [1984].** *Implementations of PROLOG*, Ellis Horwood, Ltd.
- van Caneghem, M. [1986].** *L'Anatomie de Prolog*, Paris, Interéditions.
- Casanova, M.A. e C.M.O. Moura [1986].** "Designing database applications in Logic Programming", Anais do IFIP '86 World Computer Congress, North-Holland Publishing Company, 235-240.
- Chang, C.C. e H.J. Keisler [1973].** *Model Theory*, North-Holland Publishing Company.
- Chang, C-L. e R. C-T. Lee [1973].** *Symbolic Logic and Mechanical Theorem Proving*, Academic Press.
- Chang, C-L. e A. Walker [1986].** "PROSQL: a Prolog programming interface with SQL/DS", em *Expert Database Systems*, L. Kerschberg (ed.), The Benjamin/Cummings Publishing Company, 233-246.
- Church, A. [1941].** *The Calculi of Lambda Conversion*, Princeton University Press.
- Clark, K.L. [1978].** "Negation as Failure", em *Logic and Databases*, H. Gallaire e J. Minker (eds.), Plenum Press.
- Clark, K.L. [1982].** "An introduction to Logic Programming", em *Introductory Readings in Expert Systems*, D. Michie (ed.), Gordon and Breach Science Publishers.
- Clark, K.L. e K.G. McCabe [1984].** *micro-PROLOG: Programming in Logic*, Prentice-Hall, Inc.
- Clocksin, W.F. e C.S. Mellish [1984].** *Programming in Prolog* (2^a ed.), Springer- Verlag.
- Coelho, H., J.C. Cotta e L.M. Pereira [1980].** *How to solve it with Prolog* (2^a ed.), Laboratório Nacional de Engenharia Civil, Lisboa, Portugal.
- Cohen, J. [1985].** "Describing Prolog by its Interpretation and Compilation", Comm. ACM 28:12 (dez. 1985).
- Colmerauer, A. [1985].** "Prolog in 10 figures", Comm. ACM 28:12 (dez. 1985).
- Colmerauer, A., H. Hanoui, P. Roussel e R. Pasero [1973].** "Un Systeme de Communication Homme-Machine en Français", Groupe d'Intelligence Artificielle, Université d'Aix-Marseille, Marseille, France.
- Cordeiro, J.L. [1985].** *Anatomia do Prolog*, Tese de Mestrado, Instituto Militar de Engenharia, Rio de Janeiro, Brasil (junho 1985).
- Covington, M. [1986].** "Programming in Logic", PC TECH Journal (dez./jan. 1985).

- Cuadrado, C.Y. e J.L. Cuadrado [1985].** "Prolog goes to work", Byte (agosto 1985).
- Davis, R.E. [1985].** "Logic Programming and Prolog: A Tutorial", IEEE Software (set. 1985).
- Davis, M. e H. Putnam [1960].** "A computing procedure for quantification theory", J. ACM 7:3 (julho 1960), 201-215.
- De Long, H. [1970].** *A Profile of Mathematical Logic*, Addison-Wesley Publishing Company.
- Dreben, B. e W.D. Goldfarb [1979].** *The Decision Problem: Solvable Classes of Quantificational Formulas*, Addison-Wesley Publishing Company.
- Dwork, C., P.C. Kanellakis e J.C. Michell [1984].** "On the sequential nature of unification", J. Logic Programming 1:1 (junho 1984), 35-50.
- van Emden, M.H. e R.A. Kowalski [1976].** "The semantics of predicate logic as a programming language", J. ACM 23:4 (out. 1976), 733-742.
- Enderton, H.B. [1972].** *A Mathematical Introduction to Logic*, Academic Press.
- Ferguson, R. [1981].** "Prolog - A step toward the ultimate computer language", Byte (nov. 1981).
- Fleisig, S., D. Loveland, A.K. Smiley III e D.L. Yarmush [1969].** "An implementation of the model elimination proof procedure", J. ACM 21:1 (jan. 1974), 124-139.
- Furtado, A.L. e E.J. Neuhold [1986].** *Formal Techniques for Data Base Design*, Springer-Verlag.
- Gabbay, D.M. e M.J. Sergot [1986].** "Negation as Inconsistency-I", J. Logic Programming 3:1 (abril 1986), 1-36.
- Gallaire, H., J. Minker e J-M. Nicolas [1984].** "Logic and Databases: A deductive approach", ACM Computing Surveys 16:2 (junho 1984), 153-185.
- Garavaglia, S. [1987].** *Prolog: Programming Techniques and Applications*, Harper and Row Publishing Company.
- Giannesini, F., H. Kanoui, R. Pasero e M. van Caneghem [1986].** *Prolog*, Addison- Wesley Publishing Company.
- Gilmore, P.C. [1960].** "A proof method for quantification theory: its justification and realization". IBM J. Res. Develop. 4 (jan. 1960), 28-35.
- Green, C. [1969].** "Application of theorem proving to problem solving", Anais do First Int'l. Joint Conf. on Artificial Intelligence, 219-239.

- Guttag, J.V. [1976].** "Abstract data types and the development of data structures", ACM SIGPLAN Notices 8:2.
- Hammond, P. e M. Sergot. [1985].** *apes: Augmented Prolog for Expert Systems - Reference Manual*, Logic Based Systems Ltd., Richmond, Surrey, England.
- Hayes, P.J. [1973].** "Computation and Deduction", Anais do 2nd Mathematical Foundations of Computer Science Symp., Czechoslovak Academy of Sciences, 105-118.
- Hill, R. [1974].** "LUSH-Resolution and its Completeness", DCL Memo 78, Department of Artificial Intelligence, University of Edinburgh, Edinburgh, Scotland.
- Hogger, C.J. [1984].** *Introduction to Logic Programming*, Academic Press.
- Horowitz, E. e S.Sahni [1976].** *Fundamentals of Data Structures*, Computer Science Press.
- IBM [1983].** *SQL/Data System Application Programming*, doc. IBM SH24-5018.
- IBM [1985].** *VM/Programming in Logic - Program Description and Operations Manual*, doc. IBM SB11-6374.
- IBM [1986].** *MVS/Programming in Logic - Program Description and Operations Manual*, doc. IBM SH40-0030.
- Jensen, K. e N. Wirth [1978].** *PASCAL - User Manual and Report*, Springer-Verlag.
- Kleene, S.C. [1952].** *Introduction to Metamathematics*, North-Holland Publishing Company.
- Kluzniak, F. e S. Szpakowicz [1985].** *Prolog for Programmers*, Academic Press.
- Kowalski, R.A. [1972].** "The Predicate Calculus as a Programming Language", Anais do International Symposium and Summer School on Mathematical Foundations of Computer Science, Jablona near Warsaw, Poland.
- Kowalski, R.A. [1974].** "Predicate Logic as a Programming Language", Anais do IFIP '74 World Congress, North-Holland Publishing Company, 689-574.
- Kowalski, R.A. [1978].** "Logic for Data Description", em *Logic and Databases*, H. Gallaire e J. Minker (eds.), Plenum Press.
- Kowalski, R.A. [1979a].** *Logic for Problem Solving*, Artificial Intelligence Series, Vol. 7, Elsevier-North Holland.
- Kowalski, R.A. [1979b].** "Algorithm = Logic + Control", Comm. ACM 22:7 (julho 1979), 424-436.

- Kowalski, R.A. [1981].** "Logic as a data base language", Anais do Advanced Seminar on Theoretical Issues on Databases, Cetrano, Italy.
- Kowalski, R.A. [1983].** "Logic Programming", Anais do IFIP '83 World Congress, North-Holland Publishing Company, 133-145.
- Kowalski, R.A. e P. Haynes [1969].** "Semantic trees in automatic theorem proving", em *Machine Intelligence 4*, B. Meltzer e D. Michie (eds.), American Elsevier, 87-101.
- Kowalski, R.A. e D. Kuehner[1971].** "Linear resolution with selection function", *Artificial Intelligence 2*, 227-260.
- Kowalski, R.A. e M. Sergot [1984].** "A Logic-based Calculus of Events", Tech. Rep. Department of Computing, Imperial College, London, England (nov. 1984).
- Kunifugi, S. e H. Yokota [1982].** "PROLOG and Relational Databases for Fifth Generation Computer Systems", Anais do Workshop for Logical Bases for Data Bases, Toulouse, France (dez. 1982).
- Lewis, H.R. [1979].** *Unsolvable Classes of Quantificational Formulas*, Addison-Wesley Publishing Company.
- Li, D. [1984].** *A Prolog Database System*, Research Studies Press, Letchworth, England.
- Lloyd, J.W. [1983].** "An introduction to deductive database systems", The Australian Computer Journal, 15:2 (maio 83).
- Lloyd, J.W. [1984].** *Foundations of Logic Programming*, Springer-Verlag.
- Lloyd, J.W. e R.W. Topor [1985a].** "A basis for deductive database systems I", Tech. Rep. 85/1, Dept. of Computer Science, Univ. of Melbourne, Victoria, Australia.
- Lloyd, J.W. e R.W. Topor [1985b].** "A basis for deductive database systems II", Tech. Rep. 85/6, Dept. of Computer Science, Univ. of Melbourne, Victoria, Australia.
- Loveland, D.W. [1968].** "Mechanical theorem-proving by model elimination", J. ACM 15:2 (abril 1968), 236-251.
- Loveland, D.W. [1969a].** "A simplified format for the model elimination theorem-proving procedure", J. ACM 16:3 (julho 1969), 349-363.
- Loveland, D.W. [1969b].** "Theorem-provers combining model elimination and resolution", em *Machine Intelligence 4*, B. Meltzer e D. Michie (eds.), Edinburgh University Press, 73-86.

- Loveland, D.W. [1970].** "A linear format for resolution" em *Symposium on Automatic Demonstration*, Lecture Notes in Mathematics 125, Springer-Verlag, 147-163.
- Loveland, D.W. [1972a].** "A unifying view of some linear Herbrand procedures", J. ACM 19:2 (abril 1972), 366-384.
- Loveland, D.W. [1978].** *Automated Theorem Proving: a Logical Basis*, North-Holland Publishing Company.
- Low, R. [1985].** "Reconhecimento de operadores em Prolog", 2º Simpósio Brasileiro de Inteligência Artificial, INPE, S.J. Campos, Brasil (nov. 1985).
- Luckham, D. [1970].** "Refinement theorems in resolution theory" em *Symposium on Automatic Demonstration*, Lecture Notes in Mathematics 125, Springer-Verlag, 163-191.
- Machtey, M. e P. Young [1978].** *An Introduction to the General Theory of Algorithms*, North-Holland Publishing Company.
- Manna, Z. [1974].** *Mathematical Theory of Computation*, McGraw-Hill Book Company.
- Marcus, C. [1986],** *Prolog Programming: Applications for Database Systems, Expert Systems and Natural Languages Systems*, Addison-Wesley Publishing Company.
- Martelli, A. e U. Montanari [1982].** "An efficient unification algorithm", ACM Transactions on Programming Languages and Systems 4:2 (abril 1982), 258-282.
- Moto-Oka, T. (ed.) [1982].** "Fifth Generation Computer Systems", Anais do Int. Conf. on Fifth Generation Computer Systems, JIPDEC, North-Holland Publishing Company.
- Naish, L. [1982].** "An Introduction to MU-Prolog", Tech. Rep. 82/2, Dept. of Computer Science, Univ. of Melbourne, Victoria, Australia.
- Nicolas, J-M. e K. Yasdanian [1983].** "An outline of DBGEN: a deductive DBMS", Anais do IFIP '83 World Congress, North-Holland Publishing Company, 711-717.
- Parker, D.S. et alli [1984].** "Logic Programming and Databases", Painel do First Int'l Workshop on Expert Database Systems, Kiawah Island, South Caroline, USA (out. 1984).
- Paterson, M.S. e M.N. Wegman [1978].** "Linear unification", J. Comput. Syst. Sci. 16:2 (abril 1978), 158-167.
- Pereira, L.M., F.C.N. Pereira e D.H.D. Warren [1979].** *User's Guide to DECsystem-10 Prolog (Provisional Version)*, Department of Artificial Intelligence, University of Edinburgh, Edinburgh, Scotland.

- Pereira, L.M. e A. Porto [1981].** "All Solutions", Logic Programming Newsletter 2, 9-10.
- Poe, M.D. et alli [1984].** "Bibliography on Prolog and Logic Programming", J. Logic Programming 1:1 (junho 1984), 81-142.
- Prawitz, D. [1960].** "An improved proof procedure", Theoria 26, 102-139.
- Reiter, R. [1971].** "Two results on ordering for resolution with merging and linear format", J. ACM 18:4 (out. 1971), 630-646.
- Reiter, R. [1978].** "On Closed World Databases", em *Logic and Databases*, H. Gallaire and J. Minker (eds.), Plenum Press.
- Reiter, R. [1984].** "Towards a logical reconstruction of relational database theory", em *On Conceptual Modelling*, M.L. Brodie, J. Mylopoulos e J.W. Schmidt (eds.), Springer-Verlag, 191-238.
- Roberts, G. [1977]** *Waterloo Prolog User's Manual*, University of Waterloo, Waterloo, Canada.
- Robinson, J.A. [1965].** "A machine oriented logic based on the resolution principle", J. ACM 12 (jan. 1965), 23-41.
- Robinson, J.A. [1968].** "The generalized resolution principle", em *Machine Intelligence* 3, B. Meltzer e D. Michie (eds.), American Elsevier, 77-94.
- Robinson, J.A. [1983].** "Logic Programming - Past, Present and Future", New Generation Computing 1, 107-124.
- Rogers, J.B. [1986].** *A Prolog Primer*, Addison-Wesley Publishing Company.
- Roussel, P. [1975].** *Prolog: Manuel de Reference et d'Utilization*, Groupe d'Intelligence Artificielle, Université d'Aix-Marseille, Marseille, France.
- Shoenfield, J.R. [1967].** *Mathematical Logic*, Addison-Wesley Publishing Company.
- Smith, J.M. [1986].** "Logic programming and databases", em *Expert Database Systems*, L. Kerschberg (ed.), The Benjamin/Cummings Publishing Company, 3-15.
- Smullyan, R.M. [1971].** *First-Order Logic*, Springer-Verlag.
- Smullyan, R.M. [1978].** *What's the Name of this Book?*, Prentice-Hall, Inc.
- Sowa, J.F. [1982].** "A Prolog to Prolog", (Unpublished manuscript), IBM Systems Research Institute, New York, USA.
- Sterling, L. e E. Shapiro [1986].** *The Art of Prolog*, The MIT Press.
- Stickel, M.E. [1984].** "A PROLOG technology theorem prover", New Generation Computing 2, 371-383.

- van Heijenoort, J. (ed.) [1967].** *From Frege to Gödel: A Source Book in Mathematical Logic*, Harvard University Press.
- Veloso, P.A.S. e A.L. Furtado [1985].** "Towards simpler and yet complete formal specifications", em *Information Systems Theoretical and Formal Aspects*, A. Sernadas, J. Bubenko e A. Olive (eds.), North-Holland Publishing Company, 175-189.
- Vieira, N.J. [1986].** *SAFO2: Manual do Usuário* (Versão Preliminar), Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro, Rio de Janeiro, Brasil.
- Walker, A. (ed.) [1987].** *Knowledge Systems and Prolog*, Addison-Wesley Publishing Company.
- Warren, D.H.D. [1974].** "WARPLAN: a system for generating plans" Memo 76, Department of Artificial Intelligence, Universidade de Edinburgh, Edinburgh, Scotland.
- Warren, D.H.D [1979].** "Prolog on the DECsystem-10", em *Expert Systems in the Micro Electronic Age*, D. Michie (ed.), Edinburgh University Press.
- Warren, D.H.D. [1982]** "Higher-order extensions to PROLOG: are they needed?", em *Machine Intelligence 10*, J.E. Hayes, D. Michie and Y-H. Pao, (eds.), John Wiley & Sons.
- Wilson, W.G. [1986].** "Prolog for applications programming", IBM Systems Journal 25:2, 190-206.
- Winston, P.H. e B.K.P. Horn [1984].** *LISP*, Addison-Wesley Publishing Company.
- Wos, L., J.A. Robinson e D.F. Carson [1965].** "Efficiency and completeness of the set of support strategy in theorem proving", J. ACM 12:4 (out. 1965), 698-709.
- Zimbarg, J. [1973].** *Introdução à Lógica Matemática*, 9º Colóquio Brasileiro de Matemática, Instituto de Matemática Pura e Aplicada - CNPq, Poços de Caldas, Brasil (julho 1973).

ÍNDICE REMISSIVO

A

alfabeto 38
 básico Prolog 289
 de primeira ordem 30, 66,
 172, 196
 proposicional 15
Área de Trabalho Prolog 313
árvore
 de EMF-refutação 188
 de LSD(f)-refutação 202
 de LSDNF(f)-refutação 213
 de RL-refutação 163
 semântica 92
átomo 50, 67, 289
atribuição de valores-verdade 20
avaliação 38
axioma 44, 56, 144

B

base de Herbrand 84

C

cadeia 14, 173
 admissível 179
 auxiliar 180
 canonização 186
 celula inicial 183

contração 174
elementar 173
extensão 174
extensão plena 186
linguagem 173
pai 180
preadmissível 179
redução 174
redução plena 186
canonização de um programa
 Prolog 378
célula inicial 183
cláusula 67
 auxiliar 155
 canonização 162
 definida 196
 definida Prolog 292
 derivada 132
 entrada 132
 f-extensão 198, 213
 fator 129
 Horn 197
 inicial 155
 instância básica 86
 linguagem 67
 não-unitária estendida
 Prolog 376
 não-unitária Prolog 292, 320
 objetivo 197
 objetivo estendida Prolog 376
 objetivo Prolog 292, 318
 pai 155

- pseudo-definida 209
- pseudo-Horn 210
- pseudo-objetivo 209
- resolvente 130
- resolvente canônico 162
- sobre um átomo 292
- subjuga 148
- subjuga por substituição 150
- tautologia 148
- unitária básica Prolog 315
- unitária estendida Prolog 376
- unitária Prolog 291
- comandos de comunicação
 - consult(X) 346
 - edconsult(X) 347
 - exit 347
 - fin 347
 - load(X) 346
 - nl 345
 - prst(X) 346
 - read(T) 346
 - readch(C) 346
 - reconsult(X) 347
 - restore(X) 347
 - save(X) 346
 - stop 347
 - tab(C) 346
 - write(U) 346
 - writes(U) 346
 - writex(U) 346
- comandos de controle
 - "cut" 349
- comandos de depuração
 - break(p(*,...,*),T) 365
 - spy(p(*,...,*),T) 366
 - trace(p(*,...,*),T) 366
- comandos para processamento de cláusulas
 - addax(C) 361
 - addax(C,N) 361
 - ax(H,R) 361
- ax(H,R,N) 361
- call(C) 360
- delax(H) 361
- delax(H,N) 361
- comandos para programação na metalinguagem 386
 - bagof(X,C,L) 386
 - compute(T,X,C,V,L) 387
 - findall(X,C,L) 386
 - setof(X,C,L) 386
- compacidade 58
- completude
 - método da eliminação de modelos fraca 183
- método da resolução-LSDNF 222
- método de resolução
 - linear 159
- método de resolução-LSD 207
- sistema axiomático 57
- sistema formal da eliminação de modelos 178
- sistema formal da resolução 139
- conectivos 15
- conjunção 35
- conjunto de discordia 117
- conjunto de suporte 153
- conjunto quase-definido (de cláusulas) 197
- conjunto quase-pseudo-definido (de cláusulas) 210
- conseqüência lógica 21, 42
- constante
 - primeira ordem 30
 - Prolog 289, 314
- consulta
 - a um programa em cláusulas 231

- a um programa em cláusulas definidas 259
a um programa em cláusulas pseudo-definidas 274
Prolog 293
contração 174, 176
correção
método da resolução-LSDNF 221
método de resolução-LSD 207
sistema axiomático 57
sistema formal da eliminação de modelos 178
sistema formal da resolução 136
- D**
- dedução 144
admissível 145
EM-dedução 176
EM-dedução linear fraca 180
EMF-dedução 180
linear 154
linear de entrada 155
LSD(f)-dedução 199
LSDNF(f)-dedução 215
R-dedução 132
RL-dedução 154
sistema *AX* 57
definição 45
disjunção 35
- E**
- espaço de busca 145
estrutura 36
estrutura de Herbrand 83
expressão 112, 342
computável Prolog 342
Prolog 375
simples 112
extensão 176
governada por f 198
plena 186
- F**
- f-extensão 213
facilidades extralógicas de controle
fail 355
repeat 355
true 355
fator 129
fecho existencial 35
fecho universal 35
floresta
de EMF-refutação 188
de RL-refutação 163
forma normal conjuntiva 35
forma normal prenex 35
fórmula
atômica 31
atômica Prolog 291
matriz 35
prefixo 35
primeira ordem 31, 39
proposicional 15
representação clausal 69
satisfatível 42
tautologicamente
equivalente 21
válida 42
verdadeira 42
função de avaliação 20, 38, 39
função de seleção 183, 191, 198
função de Skolem 71

G

generalização 55
geração de planos 391

H

H-interpretação 90
H-modelo 90
Hipótese do Mundo Fechado 217

I

implicação lógica 21, 42
instância 113
instância básica 86
instanciação 113
interpretação 36
interpretador padrão Prolog 302

L

Lema da Promoção 136
linguagem
 das cláusulas de Horn 197
 das cláusulas
 pseudo-Horn 210
 de cadeias 173
 de cláusulas 67
 definição 14
 primeira ordem
 semântica 36
 sintaxe 29
 proporcional
 semântica 19
 sintaxe 14
Teoria das Listas 51
Teoria dos Conjuntos 33

LISP

408
lista 50
 cabeça 54, 325
 cauda 54, 325
 notação de Edinburgh 325
 notação de Marseille 325
 notação por colchete em

Prolog 325
 notação por ponto em
 Prolog 325
 operações básicas em
 Prolog 327
 operações em Prolog 331

literal

66
 básico 84
 complementar 67
 negativo 66
 positivo 66
 puro 147
 resolvido 130, 173

M

maquina Prolog 302, 308, 311
matriz 35
metaprograma 379
metavariável 379
método da tabela-verdade 24
método de dedução 145
metodo de eliminação de modelos
 fraca 182
método de resolução
 com conjunto de suporte 153
 com função de seleção 183
 com função de seleção para
 conjuntos
 quase-definidos 201
 linear 156
 linear com conjunto de suporte
 inicial 157

por saturação 146
 por saturação com filtragem 148
 resolução-LSD(f) 201
 resolução-LSDNF(f) 216
 modelo 21, 43, 44
 modelo de Herbrand 84
 Modus Ponens 55

N

negação por falha finita 215
 nó
 fracasso 163
 sucesso 163

O

operador Prolog 296, 297

P

Pascal 408
 prefixo 35
 primitivas Prolog 313
 problema
 co-problema 62
 complementar 62
 decidível 60
 Implicação Lógica 60
 indecidível 60
 Insatisfatibilidade 60, 97
 parcialmente decidível 60
 reduzível 62
 Validade 60
 problema do entorno ("frame problem") 393
 procedimento
 decisão 60

decisão parcial 60
 dedução 145
 Herbrand 97
 Prolog 294
 redução 62
 refutação 96, 146
 refutação com filtragem 149
 refutação linear 161
 refutação por saturação 146
 programa
 canonização 378
 em cláusulas 231
 em cláusulas definidas 259
 em cláusulas
 pseudo-definidas 274
 Prolog 293
 regularização 300

Q

quantificador 30
 escopo 34

R

ramo
 fracasso 163
 sucesso 163
 redução 176
 redução plena 186
 refinamento 145
 refutação
 EM-refutação 176
 EM-refutação linear
 fraca 181
 EMF-refutação 181
 linear 155
 linear de entrada 155
 LSD(f)-refutação 200

LSDNF(f)-refutação 216
 R-refutação 132
 RL-refutação 155
 regra de inferência 55, 144
 contração 176
 extensão 176
 f-extensão 199, 215
 fatoração 132
 Modus Ponens 55
 negação por falha finita 215, 383
 redução 176
 resolução 131
 regularização de um programa Prolog 300
 renomeação 112, 129
 representação clausal
 conjunto de fórmulas 77
 fórmula 69
 resolução 131
 resolvente 129, 130
 binário 129
 canônico 162
 resposta 231, 259
 alternativa em Prolog 319
 computada 243, 270, 276
 correta 232, 260, 275
 genérica 231, 259
 H-correta 262
 mais geral 244
 simples 231, 259

S

semântica do modelo
 mínimo 262
 sentença 34, 43
 símbolo

funcional 30
 lógico 30
 não-lógico 30
 predutivo 30
 proposicional 15
 sistema axiomático 55
 sistema formal 144
 eliminação de modelos 176
 resolução 131
 resolução com função de seleção para conjuntos quase-definidos 199
 resolução para conjuntos quase-pseudo-definidos 214
 Skolemização 71
 subfórmula 15
 subjugamento 148
 substituição 35, 112
 básica 112
 composição 113
 simples 112
 vazia 112

T

tautologia 21, 22, 55, 148
 teorema 57
 teoria 44
 termo
 funcional Prolog 290, 291, 314
 Herbrand 83
 primeira ordem 30, 39
 Prolog 290, 314
 teste de ocorrência 117, 123

U

u.m.g 114
unificação 119, 124
unificador 114
unificador mais geral 114
universo 36
universo de Herbrand 83

V

variável
amarrada 34
anônima Prolog 289
livre 34
primeira ordem 30
Prolog 289, 314
variante 161
variante canônica 161

