DISSERTAÇÃO DE MESTRADO

COMBINANDO AVALIAÇÃO PARCIAL E INTRODUÇÃO AUTOMÁTICA DE CONTROLE PARA AUMENTAR A EFICIÊNCIA DE SISTEMAS ESPECIALISTAS

CLÁUDIA KIMIE YAMAKI

ORIENTADOR: PROF. Dr. ANTONIO EDUARDO COSTA PEREIRA

CO-ORIENTADORA: PROFa. Dra. ARIADNE MARIA BRITO RIZZONI CARVALHO

IMECC - UNICAMP

1990

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

Este exemplar corresponde à redação final da tese devidamente corrigida, defendida por CLÁUDIA kIMIE YAMAKI e aprovada pela Comissão Julgadora.

Antonio Eduardo Costa Pereira.

Prof. Dr Antônio Eduardo Costa Pereira
(Orientador)

Ariadne Maria Brito Rizzoni Carvalho

Profa. Dra. Ariadne Maria B. R. Carvalho
(Co-orientadora)

Campinas, 30 de Maio de 1990.

Aos meus pais

Hideo e Kiwako

AGRADECIMENTOS

A UNICAMP - Universidade Estadual de Campinas, pela oportunidade que me foi dada para realização deste trabalho.

Ao Prof. Dr Antônio Eduardo Costa Pereira, pela orientação, dedicação, contribuição e incentivo.

A Prof.Dra Ariadne Maria Brito Rizzoni Carvalho, pelas sugestões, críticas, boa vontade e interesse demonstrado.

Ao Paulo Gomide Cohn, pela atenção e horas dedicadas as nossas discussões que foram de fundamental importância.

A CAPES e CNPq pela bolsa de estudos concedida no decorrer do curso.

Aos meus queridos pais, irmãos e sobrinhos, pelo carinho e constante apoio e incentivo.

Aos amigos do curso, em especial a Sílvia, Henrique, Carlos, Antônio, Mônica e Alfonso, pela grande amizade e coleguismo demonstrados.

Finalmente, agradeço à todos os amigos que me acompanharam nesta fase, pelo incentivo e estímulo que sempre recebi.

RESUMO

Apresentamos uma pesquisa realizada com o objetivo de criar uma nova técnica a ser utilizada para melhorar a eficiência de sistemas especialistas baseados em representação por regras. A técnica proposta é a combinação da avaliação parcial e introdução automática de controle em um único engenho de inferência. Trataremos das dificuldades de se introduzir controle explícito em sistemas especialistas. Mostraremos alguns algoritmos baseados em avaliação parcial que aumentam em muito a eficiência destes sistemas. Demonstraremos com um caso simples a nova técnica proposta.

ABSTRACT

A research for the creation of a new technique to improve the efficiency of expert systems, based on representation by rule, is presented. The new technique, which is proposed, combines partial evaluation and the automatic introduction of control in a single inference engine. We will deal with difficulties to introduce explicit control in expert systems, and we will show some algorithms based on partial evaluation, which increases the efficience of these systems. Using a simple case we will describe the new proposed technique.

INDICE

INTRODUÇÃO

Na Inteligência Artificial, o controle efetivo de inferência é um problema muito crítico. Até mesmo para problemas relativamente simples, se a inferência não for direcionada, pode ocorrer uma explosão combinatória no número de diferentes estratégias possíveis.

Para contornar esta explosão combinatória, a maioria dos sistemas especialistas desenvolvidos usam informação de controle dependente do domínio. Por exemplo, executam as condições na ordem em que aparecem nas premissas das regras e usam as regras na ordem em que aparecem na base de conhecimentos.

Há vários problemas com este tipo de controle. Neste caso, temos que a melhor ordem para resolver uma conjunção depende da questão envolvida e, portanto, é frequentemente inconveniente ou impossível dar toda informação de controle necessário para um domínio. Um outro problema é que a parte "inteligente" do sistema, ou seja, a escolha da estratégia a ser utilizada é fornecida pelo engenheiro do conhecimento (pessoa responsável para alimentar a base de conhecimentos). Seria interessante que o controle fosse independente do domínio pois, desta forma, o próprio especialista seria responsável pela construção e manutenção da base de conhecimentos.

O objetivo deste trabalho é apresentar uma nova abordagem, independente do domínio, para o problema da explosão combinatória, aumentando consequentemente a eficiência dos sistemas especialistas baseados em representação por regras. A técnica proposta é a combinação da avaliação parcial e introdução do controle para engenhos de inferência com encadeamento regressivo.

Embora estas técnicas já tenham sido estudadas no contexto de sistemas especialistas, não conhecemos nenhum trabalho que procurasse combiná-las.

A abordagem utilizada é a introdução automática do controle através do conhecimento que o sistema possui sobre objetos e sobre a área de atuação, obtidos a partir da avaliação parcial da base de conhecimentos. O resultado são as regras da base de conhecimentos com as premissas ordenadas para execução.

Temos que todo sistema especialista introduz controle na base de conhecimentos. Entretanto, os usuais o fazem na fase da consulta. A técnica proposta tentará fazê-lo na fase da aquisição. Desta forma, o sistema ficará muito mais rápido na fase da consulta, pois parte das operações necessárias nesta fase já terão sido realizadas durante a aquisição.

Devido ao recente interesse por Inteligência Artificial na maioria dos cursos de Ciência de Computação, acreditamos que pelo menos parte dos interessados nesta tese desconhecem o assunto em questão. Desta maneira, começaremos o capítulo 1 definindo conceitos elementares sobre sistemas especialistas, restringiremos a noção de inteligência para nosso contexto, discutiremos a representação do conhecimento usando regras, mostraremos o funcionamento de sistemas especialistas que partem de hipóteses e tentam demonstrar sua validade (Provadores) usando método de inferência baseado no modus ponens.

No capítulo 2 discutiremos a avaliação parcial, veremos alguns trabalhos de outros autores e mostraremos alguns algoritmos que aumentam em muito a eficiência dos sistemas especialistas.

No capítulo 3 trataremos das dificuldades de se introduzir automaticamente controle explícito em sistemas especialistas, discutiremos alguns trabalhos publicados e demonstraremos, com um caso simples, a maneira de se combinar controle com avaliação parcial.

No capítulo 4 apresentaremos as conclusões obtidas e algumas propostas para futuras pesquisas.

CAPÍTULO 1

SISTEMAS ESPECIALISTAS

1.1 DEFINIÇÃO E CARACTERÍSTICAS

A Inteligência Artificial (I.A) surgiu em meados da década de 50, como um ramo da Ciência da Computação com o propósito principal de entender a inteligência humana e desenvolver sistemas que, com base em seus conhecimentos, desempenhem tarefas intelectualmente difíceis.

Dentre os tópicos de pesquisa da Inteligência Artificial, destaca-se o estudo dos sistemas especialistas, os quais têm demonstrado resultados práticos bastante úteis.

Os sistemas especialistas (S.E.) são programas que comportam-se como os especialistas humanos em domínios específicos do conhecimento. A maioria dos problemas em um domínio de especialidade não tem uma solução baseada em algoritmos devido a complexidade do contexto, das informações incompletas e inexatas. Portanto, os sistemas especialistas, trabalhando em áreas restritas do conhecimento, são capazes de resolver problemas para os quais não existem algoritmos que levem forçosamente à solução. Porém, isto não significa que os sistemas especialistas não sejam baseados em algoritmos, e sim, que seus algoritmos podem não encontrar a solução do problema proposto. No pior caso,

podem até encontrar soluções erradas.

Assim como os sistemas especialistas, os especialistas humanos também tentam resolver problemas para os quais não existem procedimentos mecânicos para se chegar à solução. Neste caso, os seres humanos também correm o risco de chegar à soluções erradas. Para medir o desempenho de um especialista há vários critérios, tais como: porcentagem de acertos, capacidade de evitar erros de conseqüências graves, reconhecimento das próprias limitações, entre outros [Hayes-83]. Geralmente são aplicadas variantes destas métricas para medir a qualidade dos algoritmos utilizados na construção de sistemas especialistas.

Os sistemas especialistas vem sendo comercializados e usados por profissionais de suas áreas. Seus desempenhos competem com os de profissionais humanos. Uma característica muito importante é a rapidez com que as respostas devem ser obtidas.

1.1.1 ARQUITETURA DE SISTEMAS ESPECIALISTAS

A maioria dos sistemas especialistas contém uma base de conhecimentos, um módulo de interpretação da consulta, um sistema de controle, um engenho de inferência e um módulo de coleta de evidências. Seus componentes estão organizados de acordo com a estrutura mostrada na figura 1.1.

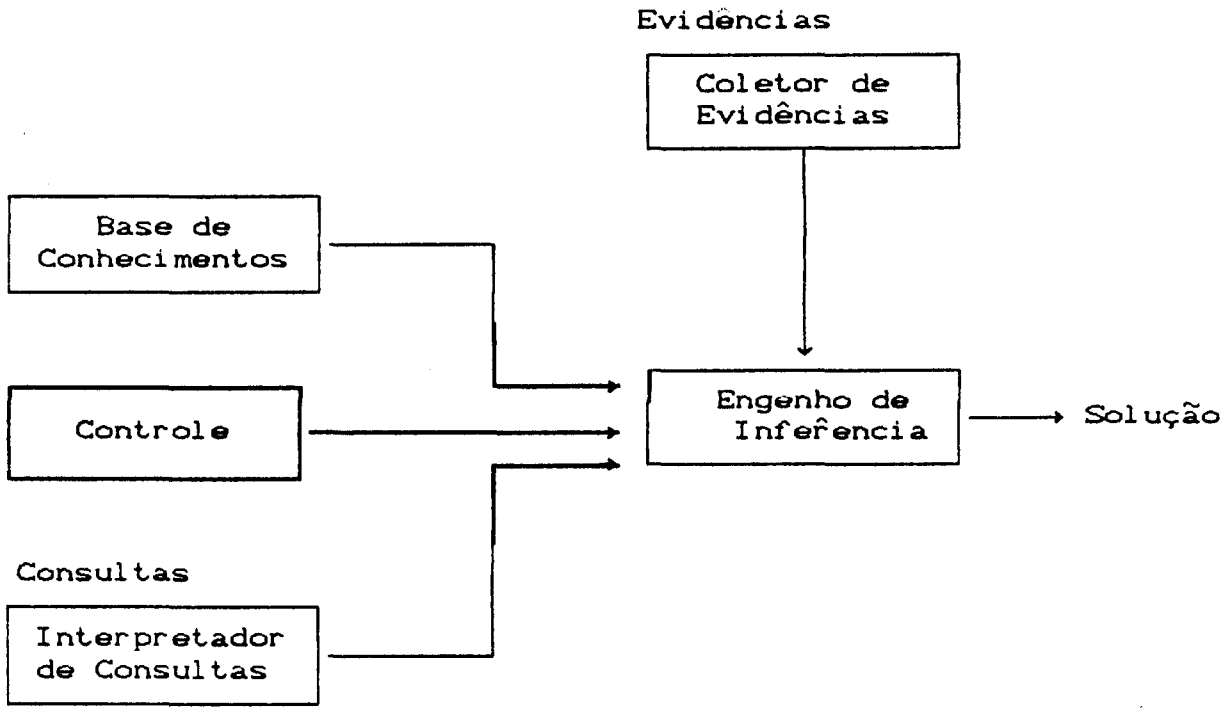


Figura 1.1 : Arquitetura de um sistema especialista

Para entendermos melhor como os sistemas especialistas trabalham, vamos a seguir definir alguns conceitos envolvidos e descrever cada um de seus componentes.

1.1.2 - DESCRIÇÃO DOS COMPONENTES

ÁREA DE ATUAÇÃO é um conjunto de objetos, características e relações que interessam a um especialista. Os objetos da área de atuação podem ficar em diferentes estados, sendo que um dado estado é definido por um conjunto de atributos. Cada um destes atributos é denominado coordenada adjetiva. O produto cartesiano das coordenadas adjetivas forma um espaço. Os diversos pontos

deste espaço representam estados.

As coordenadas adjetivas estão associadas às propriedades dos objetos. Entre estas propriedades, podemos citar as seguintes:

SITUAÇÃO EPISTEMOLÓGICA: especifica o que se conhece sobre o objeto.

INTEGRIDADE: mostra se os objetos estão inteiros ou quebrados, montados ou desmontados, etc.

RELAÇÃO : indica como dois ou mais objetos estão relacionados uns com os outros. Algumas relações típicas são igualdade, semelhança, distância, etc.

QUALIDADE : é qualquer propriedade envolvendo um único objeto. Exemplos: cor, posição, distância, etc.

Um sistema formado pelos objetos da área de atuação não pode estar em dois estados ao mesmo tempo. Além disto, o sistema pode mudar de estado. Diz-se, então, que ocorreu uma transição. Quando a transição é provocada pelo especialista, ela é denominada ação. Os sistemas são classificados de acordo com as transições realizadas.

Os sistemas especialistas trabalham da seguinte maneira: dado um estado inicial e um estado meta, os sistemas especialistas tentam encontrar uma seqüência de transições que levam do estado inicial ao estado meta. Se os estados desta seqüência estiverem todos ao longo de um eixo de coordenadas epistemológicas, diz-se que o sistema especialista está fazendo um diagnóstico. Se as transições levarem a estados ao longo de eixo de integridade

crescente, o sistema estará fazendo uma configuração ou síntese. Se as transições levarem a estados de integridade decrescente, então teremos um sistema analisador. Na prática não existem sistemas puros, ou seja, que provoquem transições ao longo de um único eixo. Por exemplo, um sistema de diagnóstico pode precisar realizar análises.

A BASE DE CONHECIMENTOS é geralmente composta de uma enorme quantidade de pequenos módulos independentes uns dos outros. Estes módulos contém descrições de objetos, de estados, relações, transições possíveis em uma área de atuação, etc. Nos sistemas especialistas, esta descrição é realizada em uma linguagem apropriada (linguagem de representação do conhecimento) [Tanimoto-87]. Neste trabalho trataremos de sistemas especialistas que utilizam regras. Frequentemente, utiliza-se uma linguagem formal.
Em princípio, nada existe na base de conhecimentos que especifique a ordem em que estas descrições devem ser utilizadas.

CONSULTA é a especificação, em geral incompleta, do estado meta. Quando a consulta é incompleta, o sistema deve completá-la por meio de pressuposições. Temos várias formas de consultas, como por exemplo:

LISTA DE HIPÓTESES : A base de conhecimentos possue embutidas algumas hipóteses sobre a forma do estado

meta. Neste caso, a consulta é um pedido para que o sistema escolha a hipótese correta.

PREDICADOS : São condições indicando as relações que os objetos devem satisfazer no estado meta.

PERGUNTAS : São pedidos de informação. Na maior parte das vezes é necessário realizar um diagnóstico para responder às perguntas.

Vários objetos descritos na base de conhecimentos são genéricos. Quando o sistema especialista tenta responder a uma consulta para, por exemplo, fazer um diagnóstico, ele deve substituir os objetos genéricos por ocorrências particulares (na secção 1.3.2 definiremos objetos genéricos/identificados e determinados). Vejamos um exemplo concreto para entender este caso. Paciente, em um sistema especialista de doenças infecciosas, é um objeto genérico da área de atuação. Quando o sistema estiver tentando responder a uma consulta, ele precisará saber qual paciente está sendo diagnosticado. Torna-se, pois, necessário substituir o objeto genérico "Paciente" por uma ocorrência bem determinada deste objeto. A substituição de objetos genéricos por ocorrências é, na maior parte das vezes, realizada por um processo dedutivo sobre a base de conhecimentos. Porém, nem sempre este processo é suficiente. Às vezes, o computador pode precisar realizar uma pesquisa que pode ter a forma de um questionário ou sensoriamento.

O ENGENHO DE INFERÊNCIA é um algoritmo de raciocínio automatizado que permite fazer deduções a respeito do conteúdo da base de conhecimentos e da consulta solicitada. Caso algum objeto da área de atuação precise ser identificado, o engenho de inferência realiza esta identificação com auxílio do módulo de coleta de evidências. O engenho de inferência deve trabalhar automaticamente, ou seja, obter respostas pela ação de procedimentos invisíveis ao usuário. Além disto, ele deve trabalhar de uma forma independente do conteúdo da base de conhecimentos.

O engenho de inferência precisa de algum tipo de CONTROLE que lhe indique a ordem em que ele deve usar os vários módulos da base de conhecimentos e os momentos em que ele precisa coletar evidências. Podemos ter dois tipos de controles externos ao sistema:

1) Controle fornecido pela pessoa que construiu a base de conhecimentos: Neste caso, o sistema especialista trabalharia de uma maneira muito semelhante a de um programa convencional. De fato, o programa convencional possui módulos descritivos independentes uns dos outros (isto é, as estruturas de dados, as fórmulas aritméticas) e um controle que é fornecido pelo programador. Exemplos de controle fornecido pelo programador em programa convencional são: IF-THEN-ELSE, FOR-DO, WHILE-DO,

REPEAT-UNTIL, CALL, RETURN, GO TO, etc.

2) Controle coletado na forma de evidencia. Neste caso, o engenho de inferência realizaria uma série de passos mudando sequencialmente de estado e tentando minimizar as diferenças entre a situação presente e o estado desejado. Este estado desejado seria uma referência. Esta maneira de trabalhar é frequentemente encontrada na natureza e em dispositivos fabricados pelo homem, tendo recebido o nome de "feedback".

Na maneira como eles são entendidos pela maioria das pessoas, os termos que definimos são vagos e até conflitantes. É portanto inviável trabalhar com estes termos em sua forma bruta. Assim sendo, procuramos nesta secção restringir a semântica destes termos, de forma que eles pudessem ser tratados mais facilmente por formalismos computacionais. Entre os termos que sofreram restrição de semântica temos: sistemas especialistas, área de atuação, estado, coordenadas adjetivas, meta, diagnóstico, consulta, pesquisa, base de conhecimentos e transição. Observemos que a terminologia assim restrita aplica-se apenas a sistemas especialistas.

Vejamos resumidamente a gênese e o funcionamento dos sistemas especialistas.

1.1.3 GÊNESE E FUNCIONAMENTO

Inicialmente, programadores constroem o engenho de inferência, o módulo de coleta de evidências, o gerador de controle e o interpretador de consultas, formando-se assim, a CONCHA do sistema especialista. A concha é usada em duas fases:

1ª Fase - Aquisição do conhecimento: Nesta fase, uma pessoa, que por convenção é chamado de engenheiro do conhecimento, coloca na base de conhecimentos as informações necessárias para se ter um bom desempenho na área de atuação. Conforme vimos, estas informações têm um caracter descritivo.

2ª Fase - Fase de consulta. Admitamos que o sistema se destine a fazer diagnóstico. Uma pessoa, o usuário final, apresenta uma pergunta sobre um problema que deseja ver resolvido. O interpretador de consultas transforma a pergunta em um conjunto de objetos cuja situação epistemológica é desconhecida. Em seguida, o engenho de inferência escolhe, na base de conhecimentos, as descrições que lhe forneçam os elementos necessários para uma série de passos dedutivos, que levem a estados nos quais a situação epistemológica dos objetos da consulta sejam conhecidos. O controle se responsabiliza para que as consultas à base de conhecimento sejam corretas, para que os resultados obtidos sejam entregues a algoritmos apropriados. Quando a consulta a base de conhecimentos não é suficiente para alimentar o processo dedutivo, o controle dispara o módulo de

coleta de evidências. Se tudo correr bem, o sistema responde a consulta de maneira satisfatória.

1.2 RESTRINGINDO A NOÇÃO DE INTELIGÊNCIA

Gostaríamos de enfatizar um ponto. Conforme bem observou Werner Jaeger [Jaeger-86], noções como a de inteligência são muito abrangentes, englobando uma ampla gama de conceitos mais simples, alguns até divergentes. No estado atual de nossa tecnologia, é quase impossível trabalhar com noções deste tipo no computador. É, portanto, necessário restringí-las. Foi o que fizemos com inteligência. Ao longo deste trabalho, consideraremos inteligência como sendo a capacidade de gerar o controle necessário para utilizar uma base de conhecimentos composta de descrições modulares. Dentro desta abordagem limitada, a inteligência será tanto menor quanto mais ajuda externa o sistema especialista precisar para gerar o referido controle. Considera-se, também, que o sistema tenha uma baixa inteligência se o controle, embora gerado sem qualquer auxílio, for de pouca sofisticação.

A maior parte dos pesquisadores que trabalham com Inteligência Artificial, quando começam a discutir implementação, acabam por delimitar a noção de inteligência. Em outras palavras, quando a discussão é realizada a nível filosófico e especulativo, a noção de inteligência é bastante ampla. Porém, quando decide-se colocar

a inteligência na máquina, ela se torna bem restrita. Vamos deixar claro que ao definirmos inteligência de forma tão restrita, não estamos agindo de forma muito diferente de outros pesquisadores.

Adotaremos a postura de que o sistema especialista exibe inteligência se, para qualquer base de conhecimentos dentro da área de atuação, ele consegue gerar o controle através de metas internas. Este processo é chamado de "pullback" pelas pessoas que trabalham com Inteligência Artificial [Negoita-85].

1.3 REPRESENTAÇÃO POR REGRAS

1.3.1 EXEMPLO DE BASE DE CONHECIMENTOS USANDO REGRAS

Existem dezenas de métodos que podem ser utilizados para realizar a descrição do conteúdo da base de conhecimentos. Destes métodos, a representação por regras é a mais popular. A sintaxe usada nas linguagens que usam este tipo de representação podem sofrer pequenas variações. Uma possibidade é a seguinte:

```
1:  deve_tomar( Paciente, Medicamento) se
        sintoma( Paciente, S) , &
        suprime( Medicamento, S) &
        não contra_indicado( Medicamento, Paciente).

2:  contra_indicado(X, Paciente) se
        agrava(X, Doença) &
        sofre( Paciente, Doença).

3:  suprime(aspirina, dor).
4:  suprime(lomotil, diarréia).

5.  agrava( aspirina, úlcera).
6.  agrava( lomotil, cirrose).
```

1.3.2 COMPONENTES DAS REGRAS

Cada bloco precedido por um inteiro é denominado regra. Cada regra é um pequeno módulo descritivo independente, em princípio, de todos os outros. Cada regra é composta de pequenas unidades denominadas estruturas. No exemplo, as estruturas são representadas por um funtor seguido de argumentos entre parênteses. Seja, por exemplo, a estrutura "sintoma(Paciente,X)". O funtor é a palavra "sintoma". Os argumentos são "Paciente" e "X". As estruturas podem ser informalmente associadas a frases simples das linguagens humanas. Por exemplo, "contra_indicado(X,Paciente)" pode ser interpretado como "X é contra indicado para Paciente".

Os objetos que aparecem como argumentos das estruturas podem pertencer a três categorias: genéricos, identificados e determinados.

- Objetos genéricos representam os membros de toda uma classe. Usaremos a convenção de representá-los por palavras que começam com letra maiúscula. Um objeto genérico da regra 1 é "Paciente".

- Os objetos identificados são aqueles que representam um único membro de uma classe. Exemplo: a palavra "aspirina" que aparece na regra 3.

- Os objetos determinados são quaisquer objetos genéricos aos quais se referiu antes. Os objetos determinados introduzem a enorme dificuldade de serem associados com contextos. Neste

trabalho não pretendemos atacar o problema de manutenção de contexto.

1.3.3 SEMÂNTICAS DAS REGRAS

As estruturas são combinadas por meio de conectivos para formar regras. Os conectivos mais comuns são "&", "se", "ou". Informalmente dá-se a estes conectivos um significado que não difere muito daquele que eles possuem no português. Por exemplo, regra 1 pode ter o seguinte significado informal: "um Paciente deve tomar um dado Medicamento se o Paciente tem um sintoma S, se o medicamento suprime S e o medicamento não é contra_indicado para o Paciente". Note que esta semântica informal é suficiente para orientar o engenheiro de conhecimentos na construção da base de conhecimentos e para servir de documentação. Temos que ela é apenas uma aproximação da semântica verdadeira.

As semânticas que os sistemas especialistas realmente usam são de definição bastante complexa e variam enormemente de sistema para sistema. Elas podem, entretanto, ser divididas em três categorias.

1) Temos sistemas que vêem as regras como sentenças lógicas utilizadas para verificar hipóteses. Neste caso, as regras são associadas a algum método de inferência para fazer demonstrações. Estas demonstrações servem, por exemplo, para escolher uma entre

diversas hipóteses. Chamaremos de PROVADORES os sistemas que funcionam desta maneira. Neste trabalho, a técnica desenvolvida aplica-se a esta classe de sistemas especialistas.

2) Na segunda categoria, as regras são vistas como sendo sentenças lógicas utilizadas para tirar conclusões e deduzir fatos. Neste caso não se tem hipóteses. Aos sistemas desta categoria, daremos o nome de SISTEMA DE DEDUÇÃO.

3) Há engenhos de inferência que vêem as regras como pares de situação-ação. A ação fica do lado esquerdo do conectivo "se". A situação é descrita pelas estruturas que estão do lado direito. Tradicionalmente, estes sistemas são conhecidos como SISTEMAS DE PRODUÇÃO.

Em sistemas especialistas reais, há diversos complicadores nas regras. Certas regras podem ter, por exemplo, alguma indicação a respeito do grau de confiança que se tem nelas. Não incluiremos tais complicadores neste trabalho, pois serviria apenas para distrair nossa atenção do problema principal que é a avaliação parcial e a introdução automática de controle.

1.4 PROVADORES

Vimos na seção 1.3 que provadores são sistemas especialistas que

partem de hipóteses e tentam demonstrar sua validade. Para fazer isto, eles precisam de algum método de inferência. Vários já foram propostos, dos quais podemos citar os mais populares:

- Métodos originários do Cálculo de Predicados.
  Temos vários métodos; porém, os mais intuitivos são aqueles baseados no modus ponens.

- Inferência Bayesiana.
  Este antigo método estatístico é bastante popular entre os que trabalham com sistemas especialistas aplicados à geologia e à engenharia.

- Teoria das Crenças de Dempster-Shaffer.
  Mais flexível do que a inferência Bayesiana, sendo o mais difundido dos métodos de inferência utilizados em provadores [Tanimoto-87].

Discutiremos apenas a inferência baseada em Cálculo de Predicados, pois desejamos apenas dar uma idéia do funcionamento de um motor de inferência, a fim de tornar mais claras as contribuições deste trabalho. Ao leitor interessado em teoria das crenças, aconselhamos a leitura de um dos trabalhos de Shaffer [Shaffer-87]. No caso de inferência bayesiana, o primeiro artigo sobre o assunto foi publicado por Duda [Duda-76].

1.4.1 INFERÊNCIA BASEADA EM CÁLCULO DE PREDICADOS

Daremos agora uma idéia intuitiva de uma das muitas maneiras que um engenho de inferência poderia usar para demonstrar uma hipótese, utilizando-se de uma base de conhecimentos expressa em termos de regras. Admitamos que as regras possam ser consideradas sentenças de Cálculo de Predicados, e que as seguintes regras estejam na base de conhecimentos.

1: amigo(X, joão) se amigo(X, paulo).
2: não amigo(pedro, joão).
hipótese: não amigo(pedro, paulo).

O conectivo "se" pode ser considerado uma implicação. As variáveis genéricas são consideradas como universalmente quantificadas. Pode-se acrescentar, portanto, as seguintes regras de inferência à base de conhecimentos.

r_1: Lei da Contraposição:
A sentença P se Q permite deduzir que não Q se não P.

r_2: Interpretação de variável universalmente quantificada.
Toda variável universalmente quantificada pode ser substituida por uma constante.

r_3: Modus Ponens. A regra p ← q permite que se escreva o consequente "p" de uma implicação em uma linha de prova se o antecedente "q" tiver aparecido na linha anterior. [Gries-81].

Para provar a hipótese, começamos por negá-la. Se, com isso, chegamos a uma contradição, a hipótese é verdadeira. Este método, originado da redução ao absurdo [Nilsson-82, Robinson-65] da antiga geometria, forma a base da maioria dos provadores de teoremas utilizados em Inteligência Artificial.

Vamos provar que a hipótese não amigo(pedro, paulo) é verdadeira.

Usando (r_2) e (1) podemos concluir que:

3: amigo(pedro,joão) se amigo(pedro, paulo).

Usando (r_1) e (3) temos que:

4: não amigo(pedro,paulo) se não amigo(pedro,jõao).

De (2) e (4) e modus ponens, concluimos que não amigo(pedro,paulo), ou seja, a hipótese é verdadeira.

Na seção 3.1 daremos uma explicação mais detalhada do mecanismo de resolução.

1.5 CONCLUSÃO

Neste capítulo delimitamos o conceito de Inteligência Artificial dentro do contexto de sistemas especialistas. Explicamos o que são sistemas especialistas inteligentes, representação por regras, e inferência baseada em Cálculo de Predicados. No próximo capítulo definiremos avaliação parcial, faremos uma breve descrição dos trabalhos publicados e apresentaremos sua aplicação em alguns algoritmos.

CAPÍTULO 2

AVALIAÇÃO PARCIAL

A idéia de avaliação parcial surgiu há muito tempo como uma técnica associada a linguagem LISP. Podemos dizer que a semente da idéia foi plantada com a publicação do trabalho de [Lombardi-67]. Não se pode dizer, entretanto, que esta idéia tenha provocado muito entusiasmo e, desta forma, pouca pesquisa se fez sobre o assunto até que, por volta de 1980, a técnica começou a ser explorada como uma possibilidade de aumentar o desempenho de linguagens de processamento simbólico, tais como LISP e PROLOG.

O primeiro estudo teórico de impacto sobre avaliação parcial pode ser creditado a Ershov [Ershov-77,Ershov-82]. Ershov imagina que a computação será realizada em uma máquina de estados na qual cada estado pode ser representado por um conjunto de variáveis com valores especificados. Dentro deste ponto de vista, um estado x causa uma computação v que leva ao estado y e que tem a forma de uma seqüência de avaliações de termos predicados e de comandos de atribuição. Houve também alguns esforços no sentido de implementar avaliadores parciais em PROLOG [Takeuchi-85, Venken-85, Sterling-86]. Tudo indica que os esforços de Takeuchi, Venken, Sterling e outros autores foram independentes dos realizados por Ershov e nenhum deles parece ter sido tão sistemático. Observamos que Ershov não é citado nem no trabalho

de Venken [Venken-85] nem no de Takeuchi e Furukawa [Takeuchi-85], nem no de Sterling [Sterling-86].

Apresentaremos um sistema diferente porém dentro do espírito daquele apresentado por Ershov. Daremos, então, nossa visão de como os trabalhos de Venken [Venken-84] e [Takeuchi-85] se encaixam e se diferenciam deste sistema. Faremos uma exposição de como nossa própria tese se encaixa nele.

Gostaríamos de observar que os autores que trabalham em avaliação parcial apresentam seus resultados em notação matemática. Esta notação é frequentemente usada para convencer os leitores da validade de certos resultados e da correção de algoritmos. Frequentemente são apresentadas provas de teoremas simples. Não se tenta, contudo, aproximar a teoria de um sistema formal, com definições, regras de inferências, axiomas e teoremas apresentados de maneira rígida e estereotipada. Adotaremos este estilo pois nos pareceu mais apropriado para transmitir idéias em uma área em que os resultados ainda não são suficientes para se construir uma teoria no sentido tradicional da palavra.

Para as nossas finalidades, vamos considerar o modelo de computação semelhante a um sistema formal, tal como definido por Delahaye [Delahaye-86], onde o programa p e´um conjunto C de fórmulas associado a uma fórmula f, a qual é denominada estrutura para entrada de dados, ou simplesmente de entrada. A computação é uma seqüência de transformações de f, usando regras de

inferência apropriadas e tendo como axiomas C.

2.1 TEORIA DA AVALIAÇÃO PARCIAL USANDO-SE SISTEMA FORMAL

DEFINIÇÃO: Dizemos que S é um sistema formal se:

1) Possui um alfabeto Ss que pode ser finito ou infinito, numerável e com uma numeração definitivamente fixada.
2) Possui um subconjunto recursivo Fs do conjunto de todas as seqüências finitas de Ss, o qual é denominado conjunto das fórmulas bem formadas de Ss.
3) Possui um subconjunto recursivo Cs de Fs, o qual é chamado de axiomas de S.
4) Possui um conjunto Rs de predicados decidíveis definidos sobre Fs, que é chamado regras de inferência.

Sejam Rs = ⟨r1,r2,r3,..,rn⟩ as regras de inferência. Ao invés de denotar ri⟨f1,f2,f3,...,fk,g⟩ para indicar o predicado ri, costuma-se escrever "f1,f2,f3,...,fk / ri ⊢ g" e ler "a partir das fórmulas f1,f2,f3,...,fk pode-se inferir g pela regra ri". Um conjunto é recursivo se existir um algoritmo que permita descobrir se um objeto pertence ou não pertence ao conjunto. Para maiores detalhes ver [Delahaye-86]. Chama-se dedução a partir das hipóteses ⟨h1,h2,...,hn⟩ a qualquer seqüência finita de fórmulas f1,f2,...,fm tal que, para qualquer i pertencente a ⟨1,2,...,m⟩:

- fi é um axioma ou

- fi é uma das hipóteses ou
- fi pode ser inferida a partir de fórmulas que a antecedem na seqüência.

Seja S um sistema formal, com um conjunto C de axiomas e uma hipótese f, onde f é uma fórmula. Seja dd = [f,f1,f2,...,fk] uma dedução a partir de f tendo C como axiomas. Diz-se que fk é uma computação de f se:

- fk foi inferida a partir de um conjunto de fórmulas que inclui f, ou se fk foi inferida a partir de um conjunto de fórmulas que inclui uma computação de f.
- Em cada passo, existe um critério rígido para se escolher a inferência.

Se fk é uma computação de f então a seqüência [f,f1,f2,...,fk] que permite deduzir fk é chamada história ou traço da computação. O conjunto de axiomas é chamado programa. A hipótese f é chamada estrutura de entrada de dados ou simplesmente de entrada.

Vejamos um exemplo para ajudar a fixar o conceito de computação e de traço. Admitamos que as fórmulas sejam funções, e que estas sejam definidas, de acordo com o cálculo lâmbda, pela seguinte sintaxe:

<expressão lâmbda> ::= <variável> |
(<expressão lâmbda> <expressão lâmbda>) |
(λ <variável> <expressão lâmbda>)

<definição> = <padrão sintático> = <expressão lâmbda>

Sobre o <padrão sintático>, vamos dizer apenas que é uma estrutura possuindo variáveis sintáticas. As seguintes definições devem ser feitas:

(d1) Dada a expressão (λv E), as ocorrências de v em E são ligadas. Se uma variável não é ligada, ela é livre.

(d2) E[X] significa uma expressão com ocorrências de X.

(d3) E[X/Y] significa uma expressão E em que todas as ocorrências de Y foram substituídas por X desde que, com este processo, nenhuma variável livre de E se torne ligada.

Depois destas definições, podemos fornecer as regras de inferência que usaremos para transformar expressões do cálculo lâmbda. Elas são:

(r1) Expansão sintática:

O lado esquerdo de um padrão sintático pode ser substituído pelo lado direito.

(r2) Conversão alfa:

Qualquer expressão na forma (λv E) pode ser convertida em (λw E[w/v]).

(r3) Conversão beta:

Expressões na forma ((λv E1) E2) podem ser convertidas em E1[E2/v].

(r4) Conversão neta:

Qualquer expressão na forma (λv (E v)) pode ser reduzida a E desde que não haja nenhuma ocorrência livre de v em E.

Dentro do contexto do cálculo lâmbda, um par programa-entrada p=(f,C) é constituído por uma expressão lâmbda f e por um conjunto de definições C. Vamos, portanto, fornecer algumas definições com a finalidade de formar um programa.

(D1) (λ(x y) E) = (λx (λy E))

(D2) (E1 E2 E3) = ((E1 E2) E3)

(D3) verdadeiro = (λ(x y) x)

(D4) falso = (λ(x y) y)

(D5) topo = (λp (p verdadeiro))

(D6) cauda = (λp (p falso))

(D7) (E1 . E2) = (λf ((f E1) E2))

(D8) [] = (verdadeiro . falso)

(D9) [E] = (falso . (E . []))

(D10) [E1 E2] = (falso . (E1 . [E2]))

(D11) [E1 E2 E3] = (falso . (E1 . [E2 E3]))

(D12) [E1 E2 ... E3] = (falso . (E1 . [E2 ... E3]))

(D13) (se E então E1 senão E2) = (E E1 E2)

(D14) vazia = (λp (p verdadeiro))

(D15) first = (λx (se (vazia x) então []
senão (topo (cauda x))))

(D16) butfirst = (λx (se (vazia x) então []
senão (cauda (cauda x))))

(D17) fput = (λ(x s) (falso . (x . s)))

Vejamos como são feitas as computações. Usaremos as definições Di,

$1 \leq i \leq 17$ e as regras de inferência rj, $1 \leq j \leq 4$.

Computação 1: A partir de (topo (E1 . E2)) podemos inferir E1.

topo0: (topo (E1 . E2)) / expansão $|\overline{\text{D5,D3,D1,D7}}$

topo1: ((λp (p (λx (λy x)))) (λf ((f E1) E2))) / beta |——

topo2: ((λf ((f E1) E2)) (λx (λy x))) / beta |——

topo3: (((λx (λy x)) E1) E2) / beta |——

topo4: ((λy E1) E2) / beta |——

topo5: E1

Conclusão 1: (topo (E1 . E2) fornece E1.

Para facilitar o entendimento das computações realizadas, vejamos uma explicação detalhada da computação 1. Nas demais computações omitiremos tais explicações.

topo0: A partir da expressão (topo (E1 . E2)) e usando a definição D5 e regra de expansão r1, inferimos ((λp (p verdadeiro)) (E1 . E2)).

A partir de ((λp (p verdadeiro)) (E1 . E2)) usando a definição D3 e regra de expansão r1, inferimos ((λp (p (λ (x y) x))) (E1 . E2)).

A partir de ((λp (p (λ (x y) x))) (E1 . E2)) usando definição D1 e regra de expansão r1, inferimos ((λp (p (λx (λy x)))) (E1 . E2)).

A partir de ((λp (p (λx (λy x)))) (E1 . E2)), definição D7 e regra de expansão r1, inferimos ((λp (p (λx (λy x)))) (λf ((f E1) E2))).

topo1: A partir de ((λp (p (λx (λy x)))) (λf ((f E1) E2))) e regra beta (r3),inferimos ((λf ((f E1) E2)) (λx (λy x))).

topo2: A partir de ((λf ((f E1) E2)) (λx (λy x))) e regra beta (r3), inferimos (((λx (λy x)) E1) E2).

topo3: A partir de (((λx (λy x)) E1) E2) e regra beta (r3), inferimos ((λy E1) E2).

topo4: A partir de ((λy E1) E2) e regra beta (r3), inferimos E1.

Computação 2: A partir de (cauda (E1 . E2)) podemos inferir E2.

cauda0: (cauda (E1 . E2)) / expansão |—— D6, D4, D1, D7

cauda1: ((λp (p (λx (λy y)))) (λf ((f E1) E2))) / beta |——

cauda2: ((λf ((f E1) E2)) (λx (λy y))) / beta |——

cauda3: (((λx (λy y)) E1) E2) / beta |——

cauda4: ((λy y) E2) / beta |——

cauda5: E2

Conclusão 2: (cauda (E1 . E2)) fornece E2.

Note que [topo1, topo2, topo3, topo4, topo5] e [cauda1, cauda2, cauda3, cauda4, cauda5] são histórias de computações. Vejamos mais algumas histórias.

Computação 3: A partir de (vazia []) podemos inferir verdadeiro.

vazia0: (vazia []) / expansão |—— D14, D8

vazia1: ((λp (p verdadeiro)) (verdadeiro .falso))/ beta |—

vazia2: ((verdadeiro . falso) verdadeiro) / expansão |—— D7

vazia3: ((λf ((f verdadeiro) falso)) verdadeiro) / beta |——

vazia4: ((verdadeiro verdadeiro) falso) / expansão |—D3,D1

vazia5: (((λx (λy x)) verdadeiro) falso) / beta |——

vazia6: ((λy verdadeiro) falso) / beta |—

vazia7: verdadeiro

Conclusão 3: (vazia []) = verdadeiro

Computação 4: A partir de (vazia [a]) podemos inferir falso.

vazia0: (vazia [a]) / expansão |—D14

vazia1: ((λp (p verdadeiro)) [a]) / beta |——

vazia2: ([a] verdadeiro) / expansão|—D9

vazia3: ((falso . (a . [])) verdadeiro) / expansão |—D7

vazia4: ((λf ((f falso) (a . []))) verdadeiro) / beta|——

vazia5: ((verdadeiro falso) (a . [])) / beta|——

vazia6: (((λx (λy x)) falso) (a . [])) / beta|——

vazia7: ((λy falso) (a . [])) / beta|——

vazia8: falso

Conclusão 4: Se a lista E não for vazia, (vazia E) = falso

Computação 5: A partir de (first [a b c]) podemos inferir a.

first0: (first [a b c]) / expansão|—D15

first1: ((λx (se (vazia x) então []

senão (topo (cauda x)))) [a b c]) / beta |—

first2: (se (vazia [a b c] então []

senão (topo (cauda [a b c]))) / expansão |—D13

first3: ((vazia [a b c]) [] (topo (cauda [a b c])))

-- vários passos análogos à computação 4,

ou seja, usando a conclusão 4 ⟶

first4: (falso [] (topo (cauda [a b c])))

/ expansão de falso |——

first5: ((λx (λy y)) [] (topo (cauda [a b c]))) / beta |——

first6: ((λy y) (topo(cauda [a b c]))) / beta |——

first7: (topo (cauda [a b c])) / expansão $|\overline{\text{D11}}$

first8: (topo (cauda (false . (a . [b c]))))

/ passos da cauda |——

first9: (topo (a . [b c])) / passos do topo |——

first10: a

Conclusão: (topo [E ...]) fornece o primeiro elemento de uma lista.

Por um processo análogo, poderíamos mostrar que (butfirst [E1 E2 ...]) fornece a cauda [E2 ...] da lista.

Seja um sistema formal destinado a realizar computações. A parte deste sistema constituído pelo alfabeto, pelas fórmulas bem formadas e pelas regras de inferência é chamada linguagem de programação, ou simplesmente linguagem. Seja Fs o conjunto das fórmulas bem formadas de uma linguagem. D e P são dois subconjuntos recursivos de Fs tal que todos os axiomas pertencem a P. (D $\cap$ P) não é obrigatoriamente vazio. Chamaremos o subconjunto D de subconjunto de dados. O subconjunto P será o subconjunto dos procedimentos. Os axiomas são procedimentos e o

conjunto deles é denominado programa. O conjunto de regras de inferência é denominado semântica da linguagem. Não examinamos se estas definições se aplicam a todas as linguagens. Provavelmente não. Elas, entretanto, se aplicam às linguagens utilizadas em Inteligência Artificial (LISP, PROLOG, LOGO, etc).

Vamos descrever D e P pela notação de Backus Naur, a qual é formada por um conjunto de regras de reconhecimento, denominadas produções. Denominaremos Dp o conjunto das produções necessárias para descrever D. Pp é o conjunto das produções que descrevem P. Dizemos que uma fórmula pertence exclusivamente a D se ela pode ser totalmente descrita pelas produções que pertencem a Dp - (Dp $\cap$ Pp). A fórmula pertence exclusivamente a P se ela é descrita pelas produções de Pp - (Dp $\cap$ Pp). Por outro lado, dizemos que a fórmula possui componentes exclusivamente em P se, para reconhecê-la, são necessárias as produções pertencentes a Pp - (Dp $\cap$ Pp).

Seja uma linguagem l(D,P,s). Se a computação de f tem componentes exclusivamente em P, dizemos que ela foi parcial. Se a computação pertence exclusivamente a D (não tem componentes em P) então ela foi total.

Vimos que uma linguagem L é constituída por um alfabeto Ss (símbolos léxicos), um conjunto recursivo fs das seqüências de Ss (fórmulas bem formadas) e um conjunto de regras de inferência (semântica). Um programa de L é o conjunto de axiomas: sistema

formal = L + programa. Uma computação é uma dedução a partir de uma hipótese de entrada chamada "dados de entrada". O provador de teoremas que aplica as regras de inferência no programa e nos dados de entrada a fim de realizar a computação é denominado avaliador. A aplicação é representada assim:

eval(dado(X,Y),p)

onde dado(X,Y) é a estrutura de entrada de dados e p é o programa. Consideremos uma linguagem Lp cujas regras de inferência produzam avaliações com as seguintes propriedades:

(1) eval(dado(x,Y),p) = (dado[x](Y), p[x])
onde x é um objeto identificado e Y é uma variável. Tanto x quanto Y são dados.

(2) eval(dado(x,y),p) = eval(dado[x](y),p[x])
onde p[x] é um programa e dado[x](Y) é uma avaliação parcial, segundo L da estrutura de entrada dado(x,Y).

O provador de teoremas Lp é denominado avaliador parcial de L.

Intérprete de uma linguagem L escrito em uma linguagem Lj é um par (inth(Pi,Ksi), intp) tal que

(3) evalj(inth(ksi,pi), intp) = eval(dado(ksi),pi).

Projetor é um intérprete da linguagem de avaliação parcial. Temos, então, que:

(4) eval(projh(dado(x,Y),p,projp) = (dado[x](Y), p[x]).

Admitamos agora uma linguagem l(Ksi,Pi,s), onde Ksi é o conjunto dos dados, Pi o conjunto dos programas e s a semântica (regras de

inferência). O resultado da avaliação de (ksi,pi), com ksi pertencente a ksi e pi pertencente a Pi, será denominado, daqui por diante, pi(ksi). Seja um intérprete (inth(Ksi,Pi),intp). Temos que:

(5) eval(inth(ksi,pi), intp) = pi(ksi).

Programa objeto obp com entrada ob(ksi) é um programa com a seguinte propriedade:

(6) eval(obh(ksi),obp) = pi(ksi).

Compilador é um programa compp aplicado à entrada comph(Pi,Ksi) e que apresenta a seguinte propriedade:

(7) eval(comph(Ksi,pi), compp) = (obh(Ksi),obp).

Vamos agora demopstrar uma série de resultados simples. A partir da expressão (4) temos:

(8) aval(projh(int(Ksi,pi),intp),projp) =
(inth[pi](Ksi),intp[pi]).

De acordo com (2), temos para qualquer ksi pertencente a ksi:
eval(inth[pi](ksi),intp[pi]) = eval(int(pi,ksi),intp) = pi(ksi).

Comparando esta fórmula com (6) concluimos que intp[pi] é um programa objeto com entrada inth[pi]. Então:

(9) (int[pi](ksi),intp[pi]) = (obh(ksi),obp)

RESULTADO 1: A avaliação parcial do intérprete com o programa conhecido, e dos dados representados por variáveis, produz o programa objeto. Isto é exatamente o que faz o compilador.

O resultado que acabamos de chegar leva a um outro: a linguagem que produz avaliações parciais possui, ela mesma, um intérprete. Este intérprete é o projetor. Se fizermos a avaliação parcial do projetor, obteremos:

(10) peval(projh(inth(Ksi,Pi),intp),projp) =
(projh[inth,intp](Ksi,Pi), projp[inth, intp]).

Pela expressão (2) temos:

(11) eval(projh[inth,intp](Ksi,pi),projp[inth,intp]) =
eval(projh(int(ksi,pi),intp),projp).

A expressão (7), juntamente com a expressão (11), leva ao seguinte:

(12) eval(projh[inth,intp](Ksi,pi),projp[inth,intp]) =
(inth[pi](Ksi),intp[pi]).

A expressão acima combinada com a expressão (8) leva a:

(13) eval(projh[inth,intp](ksi,pi),projp[inth,intp]) =
(obh(Kdi),obp).

Finalmente, combinando esta expressão com a expressão (6) obtemos:

(14) (projh[inth,intp](Ksi,pi),projp[inth,intp]) =
(comph(Ksi,pi),compp).

RESULTADO 2: Esta última fórmula diz que a avaliação parcial do projetor, para um dado intérprete, produz o compilador da linguagem correspondente ao intérprete.

Resumindo os dois resultados. temos que:o primeiro diz que, se tivermos um intérprete e um programa, podemos compilar o programa aplicando a avaliação parcial sobre (inthCKsi,pi), intp). Para isto, usamos o projetor que é, ele mesmo, um intérprete. Neste caso, a compilação está sendo efetuada por uma linguagem interpretada. Podemos também compilar o próprio projetor. Obteremos então um compilador.

Temos que Venken [Venken-84], Takeuchi-Furukawa [Takeuchi-85], Sterling [Sterling-86], utilizaram a avaliação para realizar um trabalho de compilação, muito embora não tenham deixado isto muito claro em seus artigos. Eles usaram, em todos os exemplos, linguagens de representação de conhecimento (linguagens utilizadas para preparar a base de conhecimentos) que exigiam controle fornecido pelo programador (tal como PROLOG). Este controle era implícito , ou seja, estava disfarçado na ordem com que as sentenças que formavam o programa eram introduzidas. Podemos dizer, por conseguinte, que as linguagens eram quase convencionais. Assim sendo, a avaliação parcial se tornava um modo elegante e sofisticado de construir um compilador para uma linguagem quase convencional.

2.2 FINALIDADES DA AVALIAÇÃO PARCIAL

Nesta secção vamos examinar as finalidades e aplicações da avaliação parcial, assim como mostrar que certas operações comuns em Inteligência Artificial podem ser vistas como avaliação parcial.

O princípio básico da avaliação parcial consiste em avaliar as partes de um programa que possuem dados de entrada suficientes, mantendo-se inalteradas aquelas que não possuem. O ponto importante é o seguinte: se antes da execução de um programa for possível conhecer partes dos dados que serão usados na execução, podemos utilizar tais dados para especializar o programa.

A avaliação parcial pode ser usada para duas finalidades distintas. Primeiro, ela pode ser considerada como uma ferramenta destinada a auxiliar na construção de programas eficientes. Neste caso, a avaliação parcial serviria para auxiliar o programador a raciocinar sobre seus algorítmos e a modificá-los de maneira sistemática, de modo que eles se tornem mais rápidos. Como toda ferramenta de auxílio a programação, esta também se aplica a uma certa classe de problemas, a qual iremos determinar. Uma outra finalidade seria considerá-la como um método automático de se conseguir certos resultados, tais como eficiência. Estes dois aspectos serão considerados neste trabalho.

2.3 APRESENTAÇÃO DE ALGUNS ALGORITMOS DE AVALIAÇÃO PARCIAL

Após termos analisado na seção 2.1 o conceito de avaliação parcial de maneira introdutória e genérica, veremos nesta seção alguns algoritmos concretos para nossas análises. Decidimos escolher um subconjunto do PROLOG como linguagem de implementação, pois esta escolha não afeta os resultados, e todos os algoritmos que apresentaremos ficariam igualmente claros e simples em qualquer outra linguagem com recursos de processamento simbólico. Somente deixaremos o uso do PROLOG nos casos em que este esconda algum ponto importante da discusão em questão. Quando isto ocorrer, usaremos LISP. O subconjunto do PROLOG ao qual nos restringiremos representará, neste trabalho, o mesmo papel que as versoẽs simplificadas do PASCAL representam em textos sobre estrutura de dados ou construção de compiladores. A seguir apresentaremos um resumo do PROLOG que usaremos, o qual não inclue corte e cujas relações primitivas terão seu funcionamento explicado na medida em que isto se tornar necessário.

2.3.1 PROLOG RESTRITO

As duas linguagens mais utilizadas na comunidade que trabalha com Inteligência Artificial são LISP e PROLOG. Considerando-se que pretendemos discutir sistemas especialistas que trabalham com representação por regras, e pelo fato que PROLOG

também é baseada em regras, decidimos expressar em PROLOG os algoritmos que analisaremos.

Para mostrar a sintaxe do PROLOG, vamos traduzir para esta linguagem as regras discutidas na seção 1.3.1.

```
1:   deve_tomar(Paciente, Medicamento) :-
           sintoma(Paciente, S),
           suprime(Medicamento, S),
           \+ contra_indicado(Medicamento, Paciente).

2:   contra_indicado(X, Paciente) :-
           agrava(X, Doença),
           sofre(Paciente, Doença).

3:   suprime(aspirina, dor).
4:   suprime(lomotil, diarréia).

5:   agrava(aspirina, úlcera).
6:   agrava(lomotil, cirrose).
```

As regras sofreram as seguintes alterações: o conectivo "se" foi transformado em dois pontos seguido do sinal de menos. O conectivo "&" foi substituído por vírgulas.

Na ótica do Cálculo dos Predicados, o conectivo ":-" pode ser considerado como significando "ou não" (o sinal de dois pontos representa o "ou" e o sinal de menos o "não"). Assim, a segunda sentença teria a seguinte interpretação informal:

X é contra indicado para Paciente ou não é verdade que

X agrava Doença e Paciente sofre de Doença.

Temos também que "b ← c" é equivalente a "~b ou c" [Gries-81]. Assim sendo, o conectivo ":-" também pode ser considerado como

significando implicação.

PROLOG possui um engenho de inferência que trabalha demonstrando hipóteses. O processo de demonstração é muito semelhante ao exemplificado na secção 1.4.1. Há um fato muito importante do qual não se pode esquecer: PROLOG não foi criada para ajudar a demonstrar teoremas da lógica, nem para servir de base em um sistema especialista, mas sim para ser uma linguagem de programação, ou seja, uma alternativa para linguagens tais como LISP ou SMALLTALK. Isto significa que ela é incapaz de introduzir qualquer tipo de controle no processo dedutivo. Tal como nas linguagens de programação tradicionais, este controle deve ser introduzido pelo programador.

Em linguagens como LISP ou PASCAL, o programador fornece o controle com o auxílio de construçoẽs explícitas tais como IF-THEN-ELSE, WHILE-DO e REPEAT-UNTIL. Em PROLOG, o controle está implícito e é baseado no conhecimento que se tem sobre a maneira de trabalhar do engenho de inferência da linguagem. Sabe-se que este engenho sempre escolhe as sentenças na ordem em que elas aparecem no programa. Além disto, os predicados que aparecem a direita do conectivo ":-" são escolhidos na ordem em que eles aparecem. Deste fato, teve origem a primeira maneira de se introduzir controle em um programa escrito em PROLOG, isto é,colocar as sentenças e os predicados em uma ordem conveniente.

Na prova de um teorema de lógica, a ordem em que se escolhe os passos pode ter uma influência fundamental no desempenho do provador. Uma ordem errada pode impedir que a prova seja encontrada. E uma ordem mal escolhida pode dificultar uma dada tarefa. Além disto, não existe algoritmo que permita encontrar a ordem correta (este é um resultado bem conhecido da lógica, [Shoensield-67]).

A conclusão que podemos tirar é que nada garante que um sistema, natural ou artificial, consiga provar qualquer teorema da lógica. Assim sendo, os provadores de teorema correm o risco de entrar frequentemente em laço, ou devem usar algum conhecimento ou heurística que os guie na escolha da ordem a seguir. Este conhecimento ou heurística é a parte que caracteriza a inteligência do sistema. PROLOG deixa esta parte para o programador.

Vamos analisar o funcionamento do engenho de inferência de PROLOG usando as seguintes sentenças lógicas (na notação do PROLOG):

```
1.1:   pai(j,k).
1.2:   pai(j,m).
1.3:   pai(m,n).
2.1:   avo(X,Y) :- pai(X,Z),  pai(Z,Y).
```

Suponhamos que o engenho de inferência deva provar a "avo(M,N)". O primeiro passo é negar a hipótese. Esta negação é expressa da seguinte maneira:

```
?-   avo(M,N).
```

O engenho de inferência escolhe a primeira sentença cuja cabeça (parte da sentença que está à esquerda de ":-") seja o predicado "avo(_,_)". Encontra-se a sentença 2.1. Tenta-se então provar a cauda desta sentença (parte à direita de ":-"). Prova "pai(X,Z)" com a sentença 1.1, que é a primeira que encontra para satisfazer o predicado que está tentando provar. Conclui, portanto, que X == j e Z == k. Quando tenta provar "pai(Z,Y)" verifica que, com as escolhas feitas para X e Z, tal prova não é possível. O engenho de inferência volta atrás para tentar encontrar uma outra prova para "pai(X,Z)", de modo a atribuir valores diferentes para X e Z. Esta volta para trás édenominada retrocesso ("backtracking" em inglês).

Tendo um engenho de inferência que opera por meio de retrocessos, os projetistas de PROLOG imaginaram uma outra maneira, além da ordem das sentenças, para introduzir controle no PROLOG. Trata-se do corte ("!"). O uso do corte impede o retrocesso através dele, tanto na horizontal quanto na vertical. Vejamos o seguinte exemplo:

```
1:   p :- q, r, s.
2:   p :- g, !, m.
3:   p :- v,w.
```

Se o engenho de inferência passou pela sentença 2, ele não pode mais fazer um retrocesso vertical para substituir a sentença 2 pela 3. Além disto, se ele escolheu a sentença 2 e passou pelo predicado g, não pode fazer um retrocesso horizontal para fazer uma prova alternativa de g.

A fim de garantir a eficiência de PROLOG como linguagem de programação, os projetistas foram obrigados a impor uma restrição no tipo de regra a ser usada. Se, por meio de transformações de equivalência, a regra for transformada em disjunção, só poderá haver um literal positivo. Temos que um literal é um predicado afirmado ou um predicado negado. Qualquer regra pode ser transformada em uma disjunção de literais. Por exemplo, a regra 1 seria: p ou ~q ou ~r ou ~s . O único literal positivo é p. Com esta simplificação, o método de inferência de PROLOG sempre consegue encontrar uma prova. Não se poderia garantir isto no caso de regras sem a restrição.

A principal consequência da restrição a que aludimos é que, como a única negação possível na sentença está sendo utilizada para simular o "se", torna-se impossível de simular logicamente o não. Em outras palavras, se quisermos interpretar regras de sistemas especialistas por meio de PROLOG, temos de usar o único não disponível para representar o "se". Portanto, torna-se impossível representar ocorrências de "não" como a mostrada na sentença abaixo.

```
deve_tomar(Paciente, Medicamento)     se
          sintoma(Paciente, S)       &
          suprime(Medicamento, S)    &
          não contra_indicado(Medicamento, Paciente).
```

Queremos mostrar que não é possível representar o não logicamente. É claro que sempre se pode optar por uma

representação de controle. Uma alternativa é a seguinte:

```
não(P) :- P, !, fail.
não(P).
```

A primeira sentença diz que, caso seja encontrada uma prova para P, a prova de não(P) deve falhar sem possibilidade de retrocesso. Caso a prova de P não seja encontrada, o sistema retrocede e opta pela segunda sentença, onde não(P) é considerado provado. Este tipo de não é bastante diferente daquele da lógica. Neste caso temos que não(P) é considerado verdadeiro se não for possível provar P.

Estas explicações sobre PROLOG são praticamente tudo que precisamos saber para entender os algoritmos expostos neste trabalho (não para aprender a programar) e, também, para entender a diferença entre PROLOG e uma concha de sistema especialista e a semelhança entre PROLOG e uma linguagem do tipo PASCAL ou LISP. Relembrando, a diferença entre PROLOG e a concha é que PROLOG não consegue gerar seu próprio controle. É necessário que este controle seja fornecido pelo programador, tal como em PASCAL ou LISP. Em PROLOG, porém, o controle é fornecido implicitamente enquanto que nas linguagens convencionais ele é fornecido explicitamente. Quem precisar conhecer mais sobre PROLOG pode ler um dos inúmeros livros sobre a linguagem, tais como [Arariboia-89], [Clocksin-81].

Informações sobre lógica sob o ponto de vista do cientista de

computação podem ser encontradas em [Gries-81] e no clássico de Chang e Lee [Chang-73].

Considerando-se que estaremos usando PROLOG, vamos rever os conceitos de avaliação parcial sob a ótica desta linguagem.

2.3.2 TEORIA DA AVALIAÇÃO PARCIAL PARA PROGRAMAS PROLOG

Nesta seção iremos descrever a teoria da avaliação parcial [Futamura-83, Takeuchi-85] e o método proposto por Takeuchi no qual um meta-intérprete incrementalmente especializado com respeito ao programa objeto é também construído incrementalmente. Este resultado é muito importante pois, na construção de sistemas especialistas, geralmente é muito difícil extrair todo conhecimento do especialista de uma só vez. Usualmente os sistemas são desenvolvidos incrementalmente à medida em que os conhecimentos são obtidos dos especialistas.

Seja avalp o avaliador parcial para programas PROLOG, escrito em PROLOG. Pelo fato de que avalp é um programa escrito em PROLOG, é natural representá-lo por uma relação:

(1) avalp(Programa, Dados_do_programa, Prog_espec_dados),

onde avalp representa uma relação, Programa e Dados_do_programa são dados de entrada e Prog_espec_dados é o resultado da avaliação parcial do Programa em relação ao Dados_do_programa, ou

seja, Prog_espec_dados é o programa parcialmente avaliado em relação ao Dados_do_programa.

Vejamos a definição de avaliação parcial (1) representada no diagrama da figura 2.1.

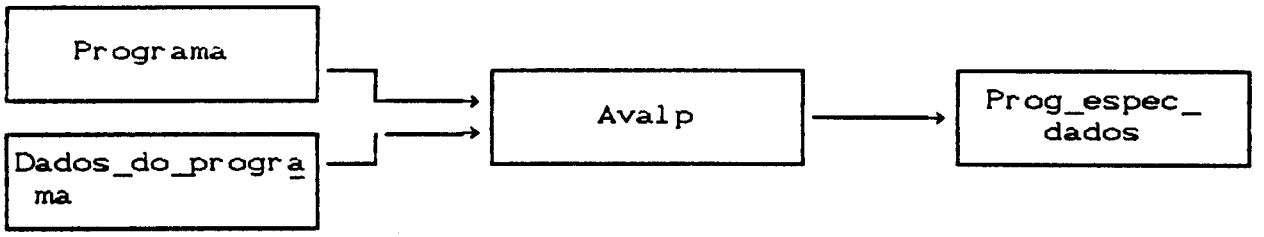


Figura 2.1 Avaliação parcial de Programa em relação aos Dados_do_programa

Seja r(U,V,W) uma relação ternária, onde U,V,W são dados de r. Podemos avaliar parcialmente r(U,V,W) em relação ao dado U. O resultado desta avaliação pode ser denotado por:

(2) r_u(V,W)

Assim, pela definição de avaliador parcial (1) e por (2), podemos representar a avaliação parcial de r(U,V,W) em relação a U por:

(3) avalp(r(U,V,W), U, r_u(V,W))

Como r é uma relação qualquer (um programa escrito em PROLOG), suponhamos que r(U,V,W) seja um engenho de inferência ei(RC,Cs,Res) escrito em PROLOG, onde RC é uma representação de conhecimento (um programa escrito em uma linguagem de

representação de conhecimento), Cs é a consulta e Res a resposta à consulta. Neste caso temos a seguinte unificação:

(4) r = ei, U = RC, V = Cs, W = Res.

Substituindo-se (4) em (3), temos que a avaliação parcial de um engenho de inferência em relação a uma representação de conhecimento RC é dada por:

(5) avalp(ei(RC,Cs,Res), RC, ei_rc(Cs,Res)),

onde ei_rc é o engenho de inferência ei(RC,Cs,Res) especializado para a representação de conhecimento RC.

A relação (5) pode ser representada pelo diagrama da figura 2.2.

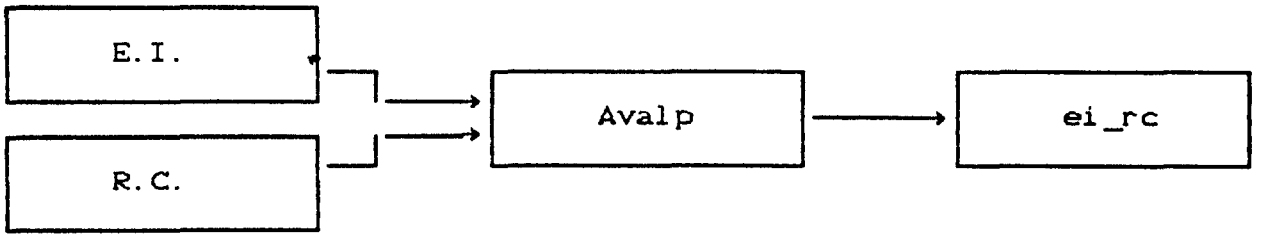


Figura 2.2 Avaliação parcial do engenho de inferência em relação a representação do conhecimento.

A entrada de ei_rc é a consulta Cs e Res é a saída. Vejamos a figura 2.3.

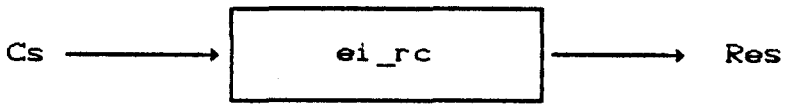


Figura 2.3 Dada a consulta Cs, ei_rc fornece Res

Para que ei(RC, Cs, Res1) e ei_rc(Cs, Res2) sejam programas equivalentes (programa ei(RC, Cs, Res1) antes e após avaliação parcial), é necessário que, dada uma consulta Cs, tenhamos:

(6) ei(RC, Cs, Res1) e ei_rc(Cs, Res2) e Res1 == Res2

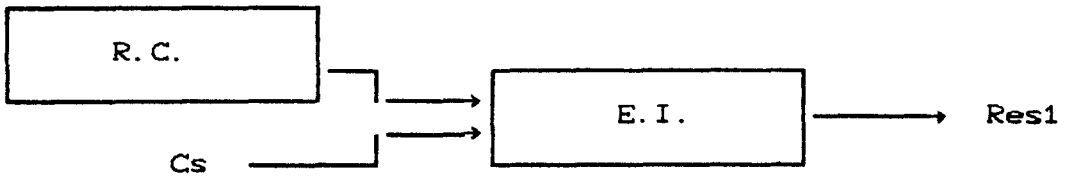


Figura 2.4 Diagrama do funcionamento do engenho de inferência

Comparando a figura 2.3 e a figura 2.4 pela relação (6) temos que, dada a consulta Cs temos que Res = Res1. Porém, o sistema da figura 2.3 é mais eficiente que o sistema da figura 2.4.

Como avalp é um programa escrito em PROLOG, podemos avaliar parcialmente avalp em relação a um engenho de inferência (programa escrito em PROLOG), específico de um sistema bem definido de regras, ou seja, especializar avalp para trabalhar apenas com este engenho de inferência. Assim obtemos:

```
(7)   avalp( avalp( ei(RC, Cs, Res), RC, ei_rc(Cs, Res)),
             ei(RC, Cs, Res),
             avalp_ei(RC, ei_rc(Cs, Res)) ).
```

onde avalp_ei é o avaliador parcial especializado para o engenho de inferência ei(RC, Cs, Res).

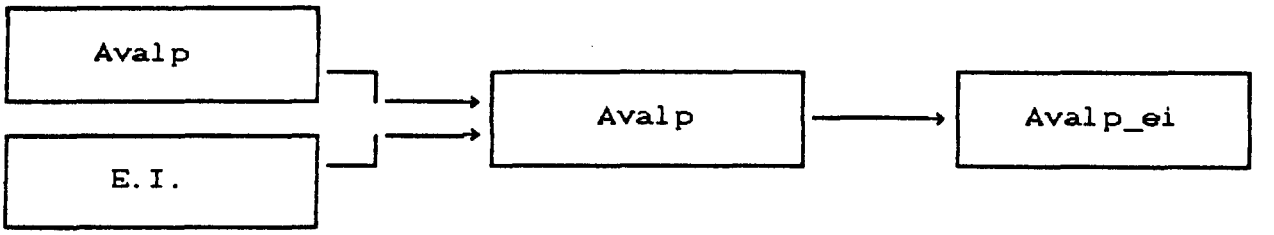


Figura 2.5 Avaliação parcial do avalp em relação ao E.I.

A entrada de avalp_ei é uma representação de conhecimento RC, a saída é o engenho de inferência especializado para a representação de conhecimento RC. Vejamos o diagrama da figura 2.6.

Figura 2.6 Diagrama de avalp_ei

Chamaremos avalp_ei de compilador de conhecimentos (comp_conh) e ei_rc de conhecimento ativo ou conhecimento executável (conh_exe). Os nomes compilador e executável foram usados para enfatizar o "paralelismo" entre os sistemas de representação de conhecimento e linguagens de programação clássicas.

Temos então que:

(8) avalp_ei == comp_conh e ei_rc == conh_exe

Substituindo-se (8) em (7), obtemos:

```
(9)  avalp( avalp(ei(RC, Cs, Res), RC, conh_exe(Cs, Res)),
            ei(RC, Cs, Res),
            comp_conh(RC, conh_exe(Cs, Res)) )
```

Substituindo-se (8) em (6), obtemos:

(10) ei(RC, Cs, Res1) e conh_exe(Cs,Res2) e Res1 == Res2.

A programação incremental é muito importante na construção de sistemas especialistas. Isto é devido ao fato de que, geralmente, deseja-se que o conhecimento seja acrescentado incrementalmente à base de conhecimentos. Utilizando-se a teoria da avaliação parcial aplicada ao engenho de inferência, o seguinte resultado mostra que é possível a acumulação incremental do conhecimento usando-se a avaliação parcial.

Seja o conhecimento RC expresso por um conjunto de regras Cj

(11) $$RC = \bigcup_{j=1}^{n} C_j$$

Seja avalp_ei_k($\bigcup_{j=K+1}^{n} C_j$, ei_rc(Cs,Res)) um compilador de conhecimentos tal que, dadas as regras $\bigcup_{j=K+1}^{n} C_j$, produza um conhecimento executável equivalente a $RC = \bigcup_{j=1}^{n} C_j$. A questão é que avalp_ei_k não precisa receber as regras de C1 a Ck para gerar o conhecimento todo. O avalp_ei_k pode ser considerado como um "compilador" que já "conhece" as regras de 1 a k. É um "compilador" que já possui uma "biblioteca" de conhecimentos prévios.

Temos que avalp_ei_k pode ser construído com a seguinte fórmula:

(12) avalp(avalp_ei_k($\bigcup_{j=k+1}^{n} C_j$, ei_rc(Cs, Res)),

Ck+1,

$$\text{avalp\_ei\_k+1}\Big(\bigcup_{j=k+2}^{n} C_j,\ \text{ei\_rc}(Cs,\ Res)\Big)\Big).$$

para k = 0,1,...,n, onde ei_rc(Cs, Res) tem o conhecimento equivalente a $\bigcup_{j=1}^{n} C_j$.

Para k = 0, $\text{avalp\_ei\_0}\Big(\bigcup_{j=1}^{n} C_j,\ \text{ei\_rc}(Cs,\ Res)\Big)$ é equivalente a

(13) $$\text{avalp\_ei}\Big(\bigcup_{j=1}^{n} C_j,\ \text{ei\_rc}(Cs,\ Res)\Big).$$

Faremos a seguir uma demonstração informal da fórmula (12)

Substituindo-se (11) em (13) temos que:

(14) avalp_ei(RC, ei_rc(Cs, Res))

Admitamos que o engenho de inferência já tenha embutido os conhecimentos equivalentes a $\bigcup_{j=1}^{k} C_j$. Chamemos este engenho de inferência especializado de ei_k.

Temos que o compilador de conhecimentos de ei_k é:

(15) $$\text{avalp\_ei\_k}\Big(\bigcup_{j=k+1}^{n} C_j,\ \text{ei\_rc}(Cs,\ Res)\Big)$$

Aplicando-se em (9) o engenho de inferência ei_k e o conhecimento $\bigcup_{j=k+1}^{n} C_j$, obtemos:

(16) $$\text{avalp}\Big(\text{avalp}\Big(\text{ei\_k}\Big(\bigcup_{j=k+1}^{n} C_j,\ Cs,\ Res\Big),$$

$\bigcup_{j=k+1}^{n} C_j$, conh_exe(Cs, Res)),

ei_k($\bigcup_{j=k+1}^{n} C_j$, Cs, Res),

avalp_ei_s($\bigcup_{j=k+1}^{n} C_j$, conh_exe(Cs, Res))).

Vamos mostrar que avalp_ei_s é equivalente a avalp_ei_k, ou seja que avalp_ei_k é equivalente ao compilador de conhecimentos tal que, dada as regras $\bigcup_{j=k+1}^{n} C_j$, produz um conhecimento executável equivalente a $\bigcup_{j=1}^{n} C_j$.

Fazendo-se a avaliação parcial de avalp_ei_s($\bigcup_{j=k+1}^{n} C_j$, conh_exe(Cs, Res)) em relação a C_{k+1}, temos:

(17) avalp(avalp_ei_s($\bigcup_{j=k+1}^{n} C_j$, conh_exe(Cs, Res)),

C_{k+1},

avalp_ei_t($\bigcup_{j=k+2}^{n} C_j$, conh_exe(Cs, Res))).

Pela definição de avaliador parcial, temos que o conh_exe(Cs,Res) gerado por avalp_ei_s($\bigcup_{j=k+1}^{n} C_j$, conh_exe(Cs, Res)) deve ser equivalente a ei_k($\bigcup_{j=k+1}^{n} C_j$, Cs, Res), ou seja, deve produzir o mesmo Res.

Pelo fato de que ei_k($\bigcup_{j=k+1}^{n} C_j$, Cs,Res) tem o conhecimento equivalente a $\bigcup_{j=1}^{n} C_j$, então conh_exe também terá conhecimentos equivalentes a $\bigcup_{j=1}^{n} C_j$.

Deste modo, por (15) temos que: conh_exe(Cs,Res) é equivalente a ei_rc(Cs, Res).

Assim, podemos concluir também que:

avalp_ei_s($\bigcup_{j=k+1}^{n} C_j$, X1) é equivalente a avalp_ei_k($\bigcup_{j=k+1}^{n} C_j$, X2), pois X1 é equivalente a X2, que por sua vez é equivalente a ei_rc(Cs,Res).

Portanto, a fórmula (17) torna-se:

(18) avalp(avalp_ei_k($\bigcup_{j=k+1}^{n} C_j$, ei_rc(Cs, Res)),
Ck+1,
avalp_ei_t($\bigcup_{j=k+2}^{n} C_j$, ei_rc(Cs, Res))).

Pelo fato de que avalp_ei_t($\bigcup_{j=k+2}^{n} C_j$, ei_rc(Cs,Res)) e avalp_ei_k+1($\bigcup_{j=k+2}^{n} C_j$, ei_rc(Cs, Res)) produzem os mesmos resultados (ei_rc(Cs, Res)), temos que (18) torna-se:

(18) avalp(avalp_ei_k($\bigcup_{j=k+1}^{n} C_j$, ei_rc(Cs, Res)),
Ck+1,
avalp_ei_k+1($\bigcup_{j=k+2}^{n} C_j$, ei_rc(Cs, Res))).

Para completar a prova, basta mostrarmos que ei_k existe. Mostraremos isto através de indução sobre k.

Se k = 0, então ei_k é o ei_0, ou seja, é o ei (o engenho de inferência). Portanto, temos que ei_0 existe.

Suponhamos que ei_k-1 exista.

Vamos provar que ei_k existe. Temos pela definição de avaliação parcial que:

avalp(ei_k-1($\bigcup_{j=k}^{n} C_j$, Cs, Res),
Ck, ei_k($\bigcup_{j=k+1}^{n} C_j$, Cs, Res))

Portanto, temos que ei_k existe.

2.3.3 ALGUMAS APLICAÇÕES DA AVALIAÇÃO PARCIAL

Analisando os trabalhos anteriores, bem como os resultados de nossa própria pesquisa, verificamos que se tentou usar a avaliação parcial nos seguintes problemas.

(P1) Na construção de compiladores

(P2) Para aumentar a eficiência de programas

(P3) Transposição do fosso semântico

(P4) Em aprendizado baseado em explicações

Vejamos cada uma destas aplicações, possivelmente acompanhadas de exemplos.

(P1) NA CONSTRUÇÃO DE COMPILADORES

Vimos nas discussões teóricas da avaliação parcial que, se aplicarmos um avaliador parcial conveniente a um intérprete de uma linguagem L, tendo um programa $\prod$ como dado, obteremos o objeto de $\prod$.

Pagan [Pagan-88] descreve um método, baseado no conceito teórico de computação parcial, de converter manualmente e sistematicamente intérpretes de linguagens de programação (iterativos e recursivos, escritos em uma linguagem de implementação) em compiladores, sem precisar lidar diretamente com linguagem de máquina. Neste método o intérprete pode ser

manualmente transformado em um tradutor que recebe como entrada uma forma intermediária de um programa fonte e produz como saída um programa funcionalmente equivalente, escrito na linguagem de implementação do intérprete. Este programa pode, então, ser traduzido para linguagem de máquina, usando o compilador existente na linguagem de implementação. Assim, temos que o programa em linguagem de máquina executará muito mais rápido do que o intérprete obtendo, portanto, um ganho em velocidade de execução do programa compilado.

Esta discussão foi forçosamente superficial pois nossa especialidade não é compiladores. Para maiores detalhes podemos citar os artigos de [kahn-84] e [Pagan-88].

(P2) AUMENTAR A EFICIÊNCIA DE PROGRAMAS

Vimos nas discussões teóricas que podemos executar algumas das operações contidas em um programa mesmo antes de se conhecer os dados de entrada que serão usados. Seja P um programa escrito em PROLOG (decidimos que sempre usaríamos PROLOG). Segundo Komorowsky (em [Sterling-86]), esta antecipação pode ser efetuada por três métodos: poda, propagação progressiva e propagação regressiva das estruturas. A poda mais simples consiste em eliminar as sentenças que não são realmente utilizadas no programa. Veremos mais adiante um tipo de poda mais sofisticada. A propagação progressiva consiste

em propagar qualquer atribuição parcial dos argumentos de uma meta para uma submeta. A propagação regressiva, consiste em substituir uma meta determinística pela cauda da sentença que a unifica.

Venken [Venken-85] apresenta um avaliador parcial para aumentar a eficiência de programas PROLOG, e consequentemente para ser utilizado como ferramenta de otimização. Os programas escritos em PROLOG são transformados em programas equivalentes (escritos em PROLOG, porém mais eficientes e específicos, pois estão avaliados parcialmente). Temos que este método corresponde a figura 2.1.

Vamos dar um exemplo de um avaliador simples para este tipo de aplicação.

Consideremos o seguinte programa escrito em PROLOG

```
mostra_tela(Tela) :-
1           tela(Tela, Comeco, Fim),
2           clear(Comeco, Fim),
            enqto(
3 6 9             texto(Tela, L, C, Texto),
4 7 10            put(L, C, Texto)
5 8 11          ),
            enqto(
12 15             campo(Tela, Nome, L1, C1, Comp),
13 16             putd(L1, C1, Comp, '.')
14 17           ).


enqto(X,Y) :- X, Y, fail.
enqto(X,Y).
```

onde clear, put, putd são primitivas.

Consideremos os dados do programa:

```
tela(teste,  5, 15).
texto(teste, 5, 15, ' Tela A - Programa ').
texto(teste, 9, 15, ' Tela B - Comandos ').
texto(teste, 11, 5, ' Tela C - Dados ').

campo(teste, comando, 9, 13, 1).
campo(teste, dados, 11, 13, 60).
```

Para executar o programa mostra_tela(teste) é necessário realizar 17 inferências lógicas. Note que as inferências 5, 8, 11, 14 e 17 referem-se ao predicado fail do predicado enqto.

Vejamos os passos executados por avalp para obter o programa especializado.

1º) Aplicamos avalp no predicado mostra_tela

```
?- avalp( mostra_tela( teste), L).
```

2º) O avaliador encontra a definição de mostra_tela e irá fazer a avaliação parcial

3º) Tela unifica-se com teste.

4º) tela(teste, Comeco, Fim) unifica-se com tela(teste, 5, 15). Ocorre uma propagação regressiva, onde substituimos a meta tela pela sua cauda (true). Os valores de Comeco = 5 e Fim = 15 são propagados por todo o corpo da cláusula mostra_tela.

5º) clear(5,15) unifica-se com a primitiva clear(_,_), portanto, não há nada a fazer.

6º) enqto é um predicado com cláusula definida no programa.

Portanto, deve ser executado. A 1ª cláusula do predicado enqto(X,Y) diz para executarmos primeiro X, depois Y, e falhar para que outras soluções para X e Y sejam encontradas. A 2ª cláusula sempre sucede. No nosso exemplo, X unifica-se com texto(teste, L, C, Texto) e Y unifica-se com put(L,C,Texto). Vejamos a execução de enqto:

- texto(teste, L, C, Texto) unifica-se com texto(teste, 5, 15,' Tela A - Programa'); ocorre uma propagação regressiva, onde sustituimos a meta texto pela sua cauda true. Ocorre uma propagação progressiva, onde os valores de L = 5, C = 15 e Texto = ' Tela A - Programa) são propagados . Por sua vez, put(5,15,' Tela A - Programa ') é primitiva; portanto, não há nada a fazer. O predicado fail falha para que outras soluções de X e Y sejam obtidas.
- texto(teste, L, C, Texto) unifica-se com texto(teste,9,15,' Tela B - Comandos ') e ocorre uma propagação regressiva, onde sustituimos a meta texto pela sua cauda true. Os valores de L = 9, C = 15 e Texto = ' Tela B - Comandos) são propagados . Por sua vez, put(9,15,' Tela A - Comandos ') é primitiva; portanto, não há nada a fazer. O predicado fail falha para que outras soluções de X e Y sejam obtidas.
- texto(teste, L, C, Texto) unifica-se com texto(teste,11,5,' Tela C - Dados ') e ocorre uma propagação regressiva, onde sustituimos a meta texto pela

sua cauda true. Os valores de L = 11, C = 5 e Texto = ' Tela C - Dados ') são propagados . Por sua vez, put(11,5,' Tela C - Dados ') é primitiva; portanto, não há nada a fazer. O predicado fail falha para que outras soluções de X e Y sejam obtidas.

7º) Temos uma outra chamada para o predicado enqto, onde X = campo(Tela, Nome, L1, C1, Comp) e Y = putd(L1, C1, Comp, '.'). (A execução é idêntica ao passo 6).

Após a aplicação do avaliador parcial no programa em relação aos dados, ou seja, se executarmos o programa

```
avalp(mostra_tela(teste),L).
```

obteremos o seguinte programa mais eficiente.

```
mostra_tela(teste) :-
1          clear(5, 15),
2          (put(5, 15,' Tela A - Programa '),
3           put(9,15, 'Tela B - Comandos '),
4           put(11, 5, ' Tela C - Dados '),
5          (putd(9, 13, 1, '.'),
6           putd(11, 13, 60, '.'),
7           true)).
```

Neste caso, para executar o programa mostra_tela(teste), será necessário realizar apenas 07 inferências lógicas.

Vejamos a listagem do avaliador parcial para programas PROLOG .

```
avalp( Meta, Clausula) :-
        execute(Meta, [], [], And_Or),
        imprima(And_Or, Clausula).
```

```
execute((X,Y), Resto, P, LR) :-
        execute(X, resto(Y,Resto), P, LR).


execute((X;Y), Resto, P,  and( or(Resp1, Resp2), P)) :-
        execute(X, Resto, [], Resp1),
        execute(Y, Resto, [], Resp2).


execute(X, Resto, P, LR) :-
        primitiva(X, P, PP),
        execute_resto(Resto, PP, LR).


execute(X, Resto, P, and(Or, P) ) :-
        setof(C, todas_clausulas(X, Resto, C), M),
        lista_or(M, Or).


execute(nil, Resto, P, LR) :-
        execute_resto(Resto, P, LR).



execute_resto(nil, Resp, Resp).

execute_resto(resto(P,LP), Part, Resp) :-
        execute(P, LP, Part, Resp).


todas_clausulas(X, Resto, C) :-
        clause(X, Y),
        execute(Y, Resto, nil, C).


primitiva(X, Resto, and(X, Resto)) :-
        prim(X),
        efeito_lateral(X),!.

primitiva(X, LR, LR) :-
        prim(X),
        call(X),!.

primitiva(X, LR, and(X, LR)) :-
        prim(X).


prim(write(X)).

prim(nl).
```

```
prim(X is Y).
prim(X = Y).

efeito_lateral(write(X)).
efeito_lateral(nl).
```

O predicado avalp(Arg1, Arg2) recebe Arg1, chamada para ser avaliada como entrada e fornece Arg2, que corresponde a avaliação parcial de Arg1.

No predicado execute(Arg1, Arg2, Arg3, Arg4), temos que: Arg1 é a chamada para ser avaliada; Arg2 corresponde a lista de chamadas para ser avaliada após a avaliação de Arg1; Arg3 é o resultado parcial das avaliações e Arg4 é a lista resultante, correspondente a saída da avaliação.

(P3) TRANSPOSIÇÃO DO FOSSO SEMÂNTICO

Os vários sistemas baseados em regras são muito semelhantes entre si com relação a sintaxe; porém, diferenciam-se muito com relação à semântica. Consideremos dois sistemas B e C, ambos baseados em regras, com a mesma sintaxe, porém com semânticas diferentes. Esta diferença semântica entre B e C pode ter duas origens. A primeira seria a diferença de estratégia, na qual o

engenho de inferência B pode ter estratégia diversa da utilizada pelo engenho C para escolher a regra apropriada. A segunda seria a diferença no método de dedução, onde dois engenhos podem dar tratamentos distintos para a mesma regra. Por fosso semântico queremos expressar todas as diferenças de estratégias e métodos de dedução.

O fosso semântico pode ter componentes apenas na estratégia, apenas no método dedutivo ou em ambos. Admita-se, agora, que o sistema C seja suficientemente completo para fazer tudo que B faz, ou seja, podemos escrever regras em C que descrevam o comportamento de B em qualquer circunstância. Tais regras são chamadas meta-regras e conseguem transpor o fosso semântico, ou seja, para uma dada base de conhecimentos, as meta-regras permitem que se obtenha, com o engenho de inferência de C, o mesmo resultado que se obteria com o engenho de B.

Temos, porém, que a presença de meta-regras diminui consideravelmente o desempenho do sistema. Propõe-se, portanto, que se use avaliação parcial nas meta-regras e na base de conhecimento para obter uma nova base de conhecimentos que possa ser manipulada diretamente pelo engenho C, produzindo o mesmo resultado que a base original manipulada por B. Para esclarecer esta transposição de fosso semântico, vejamos os diagramas das figuras 2.7, 2.8, 2.9, 2.10.

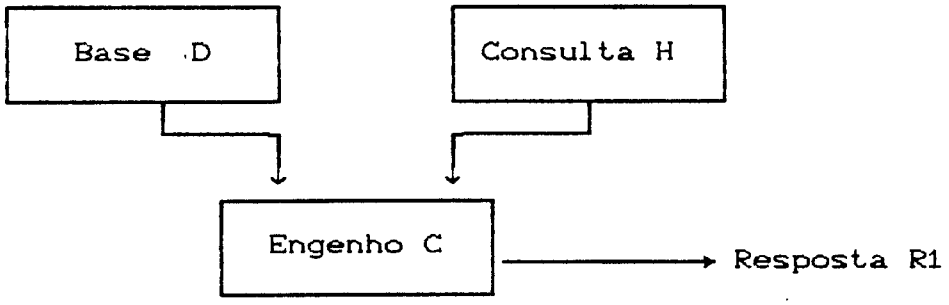


Figura 2.7 O engenho C produz a resposta R1 para a base D e a consulta H

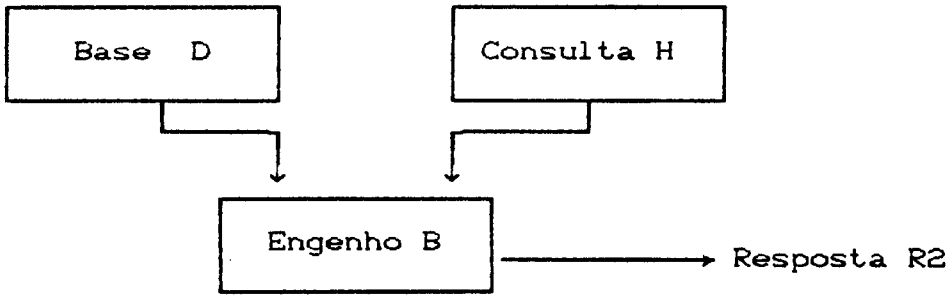


Figura 2.8 O engenho B produz a resposta R2 para a base D e a consulta H

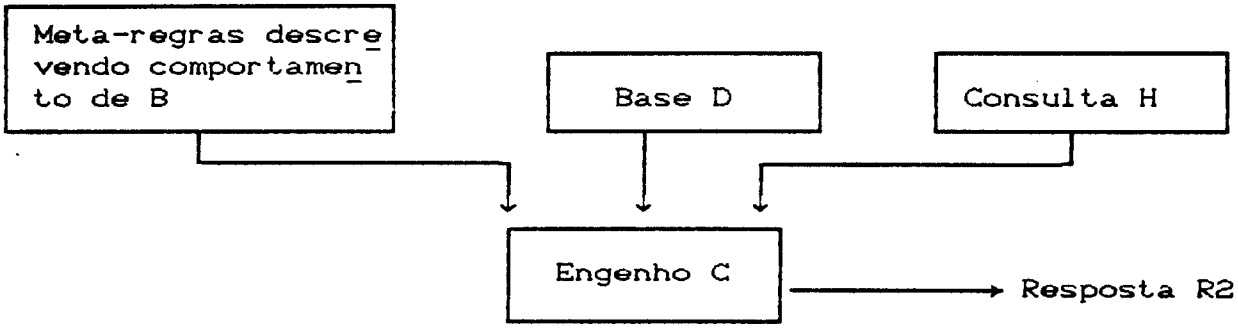


Figura 2.9 Vencendo o fosso semântico entre B e C através de meta-regras

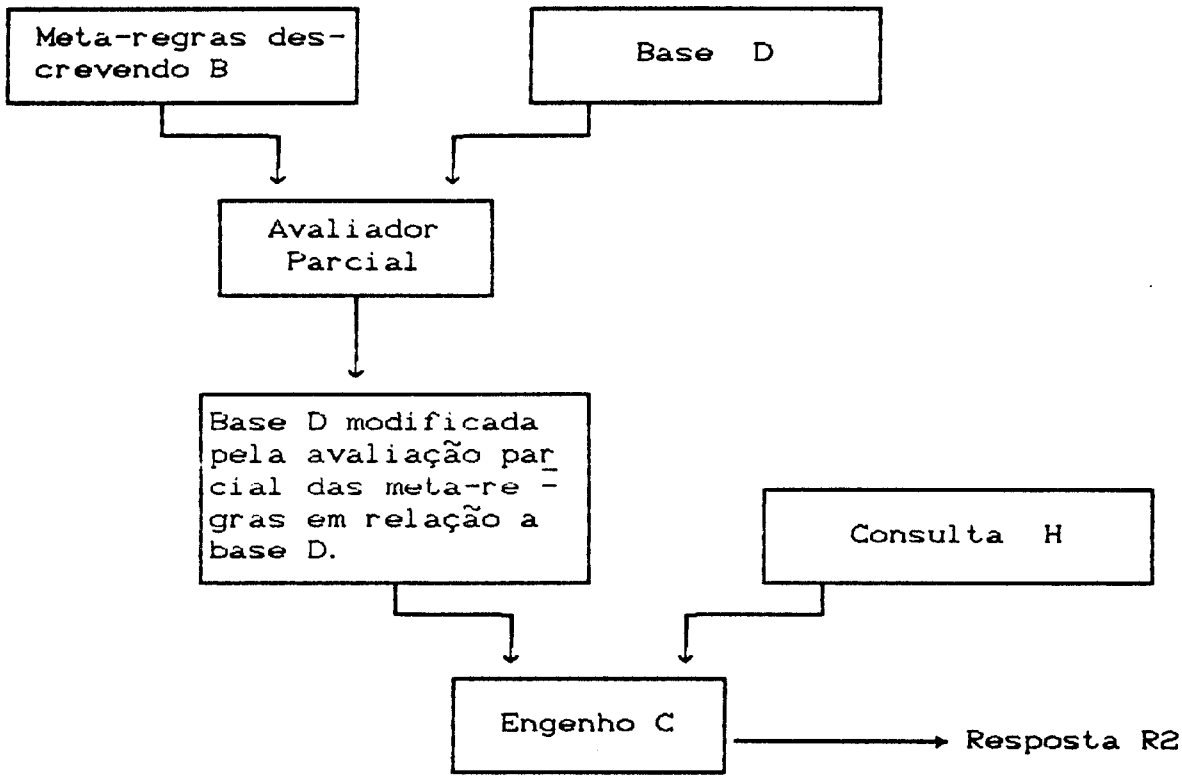


Figura 2.10 A avaliação parcial modifica D de modo que se possa obter, com o engenho C, as mesmas respostas que se obteria com o engenho B e com a base D original.

Vamos verificar como se poderia fazer a avaliação parcial para o caso do fosso semântico ter componentes apenas no método dedutivo. As dificuldades para o caso de fosso semântico na estratégia são outras, e não serão tratadas aqui. Tomemos um caso concreto, pois é mais fácil raciocinar em cima de casos concretos. Esperamos, entretanto, que as conclusões sejam bastante gerais.

Admita-se que o engenho C seja a própria linguagem PROLOG. As regras do engenho B terão a mesma forma (sintaxe) que as regras

PROLOG, mas serão vistas como regras de um cálculo de predicados normal. No sistema B não serão feitas as hipóteses simplificadoras presentes em PROLOG. Em outras palavras, B será um provador de teoremas de Cálculo de Predicados sem qualquer hipótese simplificadora. Por sua vez, C será PROLOG e, portanto, terá as seguintes hipóteses simplificadoras:

(1) Cada sentença, ao ser colocada na forma disjuntiva, terá apenas um literal positivo, o qual é chamado cabeça. Vejamos um exemplo: Seja a sentença PROLOG:

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

Esta sentença pode ser lida da seguinte maneira: X é avô de Y se X é pai de Z e Z é pai de Y. X é avô de Y ou X não é pai de Z ou Z não é pai de Y. Na leitura disjuntiva, temos apenas um literal positivo, ou seja, avo(X,Y).

(2) not(P) significa não(P) e deve ser interpretado da seguinte maneira: O engenho tenta provar P. Se não conseguir, not(P) é verdadeiro. Se conseguir, not(P) é falso.

(3) P ; Q significa o seguinte:

P ; Q :- P.

P ; Q :- Q.

Qualquer que seja uma fórmula do Cálculo de Predicados, ela pode ser expressa como um conjunto de regras na forma:

P ; Q ; S :- M , N , not(G), D.

O símbolo ":-" representa a implicação. O ";" representa "ou". A "," representa a conjunção "e". Esta forma é similar em sintaxe à usada em PROLOG. Há, contudo, duas diferenças. A primeira é a possibilidade de P ; Q a esquerda de ":-". Esta possibilidade não existe em PROLOG. A segunda é que se pode ter literais negativos como o not(G) a direita de ":-". Também isto não é possível em PROLOG e seria interpretado como significando not(G) é verdadeiro se G é falso. Há um fosso semântico entre as sentenças do PROLOG e as do Cálculo de Predicados.

Deduções com sentenças de Cálculo de Predicados exigem algoritmos muito ineficientes. Além disto, estes algoritmos não podem decidir se uma prova existe. Isto ocorre em teoria: porém, na prática, os algoritmos podem ser auxiliados por heurísticas e por controle, o que os torna mais viáveis para uma ampla classe de problemas. Devido a este fato, foram desenvolvidos vários sistemas baseados nestas sentenças. Tais sistemas, para atingirem um certo grau de eficiência, exigem programas extremamente complexos, que tentam tomar decisões cuidadosas para não seguirem caminhos longos e tortuosos no processo dedutivo. Verificaremos que, com a avaliação parcial, podemos fazer um sistema extremamente simples e de eficiência comparável a outros mais complexos.

Vamos descrever o método dedutivo usado pelo engenho B do nosso exemplo, o qual faz dedução usando o princípio de resolução e a redução ao absurdo.

O seguinte programa é constituído por um conjunto de meta-regras capazes de executar uma base de conhecimentos constituída por regras de cálculo de predicados.

Como vimos, a base de conhecimentos D é constituída por uma conjunção de regras na forma:

P ; Q ; R ; S :- M , N.

Neste caso, ela pode ser transformada em uma disjunção de literais (predicados afirmados ou negados). Assim, temos:

$$P \vee Q \vee R \vee S \vee \sim M \vee \sim N$$

Podemos também transformar a sentença em uma negação de conjunções:

falso :- $\sim P \wedge \sim Q \wedge \sim R \wedge \sim S \wedge M \wedge N$

a qual é equivalente a

$$\sim (\sim P \wedge \sim Q \wedge \sim R \wedge \sim S \wedge M \wedge N)$$

Temos que uma conjunção de fórmulas deste tipo pode ser provada usando o princípio de resolução e a redução ao absurdo. Um engenho de inferência usando o príncipio de resolução deve introduzir a negação da consulta junto a base de conhecimentos e

aplicar sucessivamente a seguinte regra de inferência (RI): Dadas as regras:

r1: :- (P1 ∧ Q1 ∧ R1 ∧ S1 ...)

r2: :- (P2 ∧ Q2 ∧ ~ R1 ∧ S2 ..)

podemos concluir:

r3: :- (P1 ∧ Q1 ∧ S1 ∧ ... ∧ P2 ∧ Q2 ∧ S2...), pois r2 fornece R1 :- P2 ∧ Q2 ∧ ... ∧ S2.... Podemos, portanto, substituir r2 por P2 ∧ Q2 ∧ ... ∧ S2.... Se chegarmos a um absurdo, ou seja, a conjunção é impossível de satisfazer, concluimos que a negação da consulta é falsa, e portanto, a consulta é verdadeira.

Exemplo: Consideremos a seguinte base de conhecimentos (baseConh).

```
r1:  homem(wang) ; mulher(wang).
r2:  homem(ling) ; mulher(ling).
r3:  humano(X)   :- homem(X).
r4:  humano(X)   :- mulher(X).
```

O primeiro passo é transformar estas regras em uma negação de conjunções. Para facilitar a manipulação destas negações de conjunções no engenho implementado, decidimos representá-las por meio de lista e guardá-las em uma base de dados para que sejam posteriormente usadas. Teremos, então, as seguintes regras transformadas.

d_1: se([not homem(wang), not mulher(wang)]).

d_2: se([not homem(ling), not mulher(ling)]).

d_3: se([not humano(X), homem(X)]).

d_4: se([not humano(X), mulher(X)]).

onde cada argumento L de se(L) é suposto negado. A lista d_1 expressa da seguinte forma:

:- (not homem(wang) ∧ not mulher(wang)).

Suponha-se que desejamos saber quais são os humanos da base de conhecimento. Neste caso, teremos a seguinte consulta: humano(X1).

1º passo: Suponhamos que a consulta seja falsa, então podemos introduzir a negação da consulta (falso :- humano(X1)) junto a base de conhecimentos, ou seja, introduzir

(a) se([humano(X1)]).

2º passo: Aplicar sucessivamente a regra de inferência RI à base de conhecimentos.

Aplicando-se a regra RI em d_3 e (a)

d_3: se([not humano(X), homem(X)]).

(a) se([humano(X1)]).

obtemos:

(b) se([homem(X1)])

Aplicando-se RI em (b) e d_1, obtemos:

(c) se([not mulher(wang)]), X1 == wang,

Aplicando-se RI em (c) e d_4, obtemos:

(d) se([not humano(wang)])

Como X1 == wang, concluimos que (b) e (a) se tornam:

(b_1) se([homem(wang)])

(a_1) se([humano(wang)])

Resolvendo-se (a_1) e (d) chegamos ao absurdo de que wang é humano e wang não é humano. Portanto, a consulta é verdadeira para X1 = wang, ou seja, wang é um dos humanos da base de conhecimentos.

O seguinte programa é a listagem do engenho B, o qual é constituído por um conjunto de meta-regras capazes de executar uma base de conhecimentos constituída por regras de cálculo de predicados.

Listagem do engenho B

```
neg(not(not(X)), Y) :- neg(X,Y).
neg(not(X), X) :- !.
neg(X, not(X)).


or((X ; Y), Z, [NX|R]) :- !,
      neg(X, NX),
      or(Y, Z, R).
or(X, Z, [NX|Z]) :- neg(X, NX).


and((X , Y), [X|R]) :- !,and(Y, R).
and(X, [X]) .


norm((X :- Y), N) :- !,
      and(Y, NY),
      or(X, NY, N).
```

```
norm((X ; Y), Z) :- !,  or((X ; Y), [], Z).
norm((X , Y), Z) :- !,  and((X , Y), Z).
norm(X, [NX]) :- neg(X, NX).


processe(end_of_file) :- !.
processe(X) :-
        norm(X,Y),
        assertz( se(Y)),
        fail.


rec(A) :- abolish(se/1),
        see(A),
        repeat,
                read(F),
                processe(F),!,
        seen.


on(X, [X|Y]) :- !.
on(X, [_|Z]) :- on(X,Z).


contra_posit([X|R], NX, R) :- neg(X, NX).
contra_posit([X|R], N, [X|NR]) :-
        contra_posit(R, N, NR).


sent(H, T) :-
        se(Y),
        contra_posit(Y, H, T).


prove(G,A) :-
        neg(G, NG),
        on(NG, A).
prove(G,A) :-
        sent(G, B),
        not( on(G, A)),
        pro(B, [G|A]).


pro([],_).
pro([G|GS], A) :-
        prove(G,A),
        pro(GS, A).


p(X) :- prove(X, []).
```

Vejamos uma explicação detalhada de cada predicado do programa engenho B.

Predicado neg(Arg1, Arg2), onde Arg1 é o predicado a ser negado e Arg2 é a negação de Arg1 (ou seja, é o predicado Arg1 negado).
Na 1ª cláusula temos que a negação de not(not(X)) é igual a negação de X.
Na 2ª cláusula temos que a negação de not(X) é X.
Na 3ª cláusula temos que a negação de X é not(X).

O predicado or(Arg1, Arg2, Arg3) transforma a disjunção Arg1 em negações de conjunções, colocando-as na lista Arg2, obtendo-se assim Arg3. Vejamos um exemplo: Dado or((X ;Y; Z), [W], Arg3) obtemos Arg3 = [not(X), not(Y), not(Z), W].
A 1ª cláusula nega o 1º elemento da disjunção, coloca-o na lista Arg3 e chama recursivamente or para negar os demais elementos da disjunção.
A 2ª cláusula de or é para o caso de Arg1 conter somente um predicado a ser negado. Portanto, basta negá-lo e colocá-lo no Arg3.

O predicado "and" recebe como entrada uma conjunção Arg1 e coloca-a em uma lista Arg2. Vejamos um exemplo: Dado and((X,Y,Z,W), Arg2), obtemos Arg2 = [X, Y, Z, W].
1ª cláusula de "and" coloca o 1º elemento da conjunção na lista

Arg2 e chama recursivamente "and" para colocar os demais elementos da conjunção.
A 2ª cláusula de "and" é para o caso de Arg1 conter somente um predicado. Portanto, basta colocá-lo na lista Arg2.

O predicado norm(Arg1, Arg2) recebe uma regra, Arg1 como entrada e coloca-a na forma normal Arg2, ou seja, em uma conjunção de predicados afirmados ou negados, representados por uma lista de predicados.
A 1ª cláusula é para o caso de Arg1 conter uma regra da forma X :- Y. A formal normal obtida será uma lista contendo a negação dos predicados de X, seguido dos predicados de Y. Vejamos um exemplo: Dado norm((X ; Y :- Z , W), LN), obteremos LN = [not(X), not(Y), Z, W].
A 2ª cláusula é para o caso de Arg1 conter uma regra da forma X ; Y. A forma normal obtida será uma lista contendo a negação dos predicados de X e de Y. Vejamos um exemplo: Dado norm((X ; Y ; Z), LN), obteremos LN = [not(X), not(Y), not(Z)].
A 3ª cláusula é para o caso de Arg1 conter uma regra da forma X. ,Y. A forma normal obtida será uma lista contendo os predicados de X e de Y. Vejamos um exemplo: Dado norm((X , Y , Z), LN), obteremos LN = [X, Y, Z].
A 4ª cláusula é para o caso de Arg1 conter uma regra unitária, ou seja, apenas um predicado. Vejamos um exemplo: Dado norm(X, LN), obteremos LN = [not(X)].

O predicado processe(Arg1) recebe uma regra, Arg1 como entrada, transforma-a, usando o predicado "norm", em uma lista de predicados afirmados ou negados (LN), ou seja, transforma-a em uma forma normal e finalmente a inserindo na base de dados na forma se(LN). Este predicado sempre falha.

O predicado rec(Arg1) recebe um arquivo Arg1 como entrada, retira da base de dados toda regra da forma se(_), abre o arquivo e para cada sentença S lida aplica opredicado processe(S). Pelo fato de que o predicado processe sempre falha, toda sentença de Arg1 será transformada na forma normal.

O predicado on(Arg1, Arg2) recebe como entrada o predicado Arg1 e a lista de predicados Arg2 e sucede se Arg1 pertencer a Arg2. Vejamos um exemplo. Dado on(X, [Z, not(W), X, P]), o predicado sucederá. Porém, dado on(X, [P, Q, R]), o predicado falhará.

O predicado contra_posit(Arg1, Arg2, Arg3) recebe como entrada a lista de predicados Arg1 e o predicado Arg2, e fornece como saída Arg3, que equivale ao Arg1 sem a negação do predicado Arg2. Vejamos um exemplo: Dado contra_posit([X, not(Y), Z, P], Y, L), obtemos L = [X, Z, P].

O predicado sent(Arg1, Arg2) recebe como entrada um predicado Arg1, procura na base de dados alguma regra da forma se(L), na

qual L contenha a negação de Arg1. A saída Arg2 corresponde a lista L sem a negação de Arg1.

O predicado prove(Arg1, Arg2) recebe como entrada o predicado Arg1, a ser provado, e uma lista Arg2 contendo os predicados já provados.
Na 1ª cláusula de prove(Arg1, Arg2) temos que se a negação do predicado Arg1 a ser provado pertencer a lista Arg2 dos predicados já provados verdadeiros, então o predicado a ser provado será falso. Com isto, chegamos a uma contradição e a prova termina.
Na 2ª cláusula de prove(Arg1, Arg2), o predicado sent(Arg1, Arg3) percorre a base de dados da forma se(Arg4), chama o predicado contra_posit(Arg4, Arg1,Arg3) para verificar se not Arg1 se unifica com algum predicado da lista Arg4 (para aplicação da regra de inferência RI). Temos que Arg3 é a lista Arg4 sem o predicado not Arg1. Deste modo temos que Arg1 foi provado. Porém antes de colocá-lo na lista Arg2 dos predicados já provados, prove certifica-se de que o predicado Arg1 ainda não pertence a lista Arg2. Após esta checagem, prove insere Arg1 na lista Arg2, chama o predicado pro(Arg3, [Arg1|Arg2]) para avaliar os próximos predicados (elementos da lista Arg3) a serem provados, para continuar o processo dedutivo, até encontrar um absurdo.

O predicado "pro" chama recursivamente "prove" para provar cada um dos predicados da lista.

Observe que o predicado prove(Arg1,Arg2) é o que realiza o processo dedutivo do engenho de inferência B, descrito anteriormente. Analisando as figuras 2.9 e 2.10 temos que o predicado "prove" corresponde a "meta-regras descrevendo o comportamento de B".

O predicado p(Arg1) recebe como entrada a consulta Arg1 que deverá ser provada. A prova será realizada pelo predicado prove(Arg1, []).

Para executarmos o engenho B devemos seguir os seguintes passos:

O 1º passo é carregar o programa do engenho de inferência.

```
?- [-'engenhoB'].
```

O 2º passo é transformar as regras da base de conhecimentos 'baseconh', que estão na forma de cálculo de predicados, para a forma normal (FN) e guardá-las na base de dados na forma se(FN). Para isto digite:

```
?- rec('baseconh').
```

A partir deste momento pode-se iniciar as sequências de consultas. As consultas devem ser realizadas da seguinte maneira:

```
?- p(Consulta).
```

Vejamos alguns exemplos:

Dada a consulta: `?- p(humano(X)).`

Respostas:

```
X = wang  -> ;
X = ling  -> ;
X = wang  -> ;
```

```
                        X = ling

Dada a consulta:        ?-  p(humano(wang)).
Resposta:               yes.

Dada a consulta:        ?-  p(humano(ling)).
Resposta:               yes.

Dada a consulta:        ?-  p(mulher(wang)).
Resposta:               yes.

Dada a consulta:        ?-  p(mulher(ling)).
Resposta:               yes.
```

Para facilitar o entendimento do algoritmo do engenho B, vamos mostrar a execução da consulta "Quais são os humanos?".

```
            ?-  *p(humano(X)).

p(humano(X)).
 prove(humano(X), []).

  neg(humano(X), NG),              % 1a cláusula de prove
  NG = not humano(X)
  on( not humano(X), []).   -->    falha.

  sent(humano(X), B).              % 2a cláusula de prove
   se(Y),                          % 1a cláusula de sent
   Y = [not homem(wang), not mulher(wang)],
   contra_posit([not homem(wang), not mulher(wang)],
                humano(X), B),   --> falha


   Y = [not homem(ling), not mulher(ling)],
   contra_posit([not homem(ling), not  mulher(ling)],
                humano(X), B),   --> falha


   Y =  [not humano(X), homem(X)],
   contra_posit([not humano(X), homem(X)],
                humano(X), B)    --> sucede
```

```
B = [homem(X)],

on( humano(X), [])  -->  falha,

not (on(humano(X), [])) -->  sucede

*************************************************************
*           Resultado 1:  humano(X) é verdadeiro            *
*************************************************************


pro([homem(X)], [humano(X)]),   % 2a cláusula de pro

 prove(homem(X), [humano(X)]), % 1a cláusula de prove
  neg(homem(X), NG),
  NG =  not homem(X)
  on( not homem(X), [humano(X)]),  -->falha


  sent(homem(X), B),
   se(Y),
   Y = [not homem(wang), not mulher(wang)],
   contra_posit([not homem(wang),not mulher(wang)],
                homem(X), B)   -->  sucede

   B = [not mulher(wang)],
   X = wang                                % unifica

   on(homem(wang), [humano(wang)])  --> falha

   not(on(homem(wang), [humano(wang)])  --> sucede


*************************************************************
*           Resultado 2: homem(wang) é verdadeiro           *
*************************************************************


   pro([not mulher(wang)], [homem(wang), humano(wang)]),

    prove(not mulher(wang), [homem(wang), humano(wang)])
     neg(not mulher(wang), NG),
     NG = mulher(wang)
     on( mulher(wang), [homem(wang), humano(wang)])  --> falha


     sent(not mulher(wang), B),
      se(Y),
      Y = [not homem(wang), not mulher(wang)]
```

```
contra_posit([not homem(wang), not mulher(wang)],
                not mulher(wang),  B)  --> falha

Y = [not homem(ling), not mulher(ling)]
contra_posit([not homem(ling), not mulher(ling)],
                not mulher(wang), B)  -->  falha

Y = [not humano(X), homem(X)]
contra_posit([not humano(X), homem(X)],
                not mulher(wang), B)  -->  falha

Y = [not humano(X), mulher(X)]
contra_posit([not humano(X), mulher(X)],
                not mulher(wang), B)  -->  sucede

B = [not humano(wang)]

on(not mulher(wang),[homem(wang),humano(wang)])-->falha

not(on(not mulher(wang),[homem(wang),humano(wang)]))
                                                  --> sucede


**********************************************************************
*          Resultado 3: not mulher(wang) é verdadeiro                *
**********************************************************************


pro([not humano(wang)],
    [not mulher(wang), homem(wang), humano(wang)]),

  prove(not humano(wang),
        [not mulher(wang), homem(wang), humano(wang)]),

   neg(not humano(wang), NG),
   NG = humano(wang)
   on(humano(wang),
      [not mulher(wang),homem(wang),humano(wang)])
   -->   SUCEDE


**********************************************************************
*          Resultado 4: not humano(wang) é verdadeiro                *
**********************************************************************
```

Temos que humano(wang) é um elemento da lista dos predicados já provados verdadeiros [not mulher(wang),homem(wang),humano(wang)],

então, not humano(wang) é falso. Absurdo, pois o resultado 4 diz que not humano(wang) é verdadeiro. Portanto, wang é um dos humanos.

Comparando com a teoria da avaliação parcial para programas PROLOG, temos três etapas para vencer o fosso semântico.

Na 1ª etapa temos um intérprete prove(G,A) do engenho B.

Na 2ª etapa, "admita-se que tenhamos" um avaliador parcial (avalp) que pode ser aplicado a prove (ver figura 2.2)

avalp(H :- Cd, prove(G,A), Hprove(G,A))

Neste caso, a figura 2.2 é representada pela figura 2.11

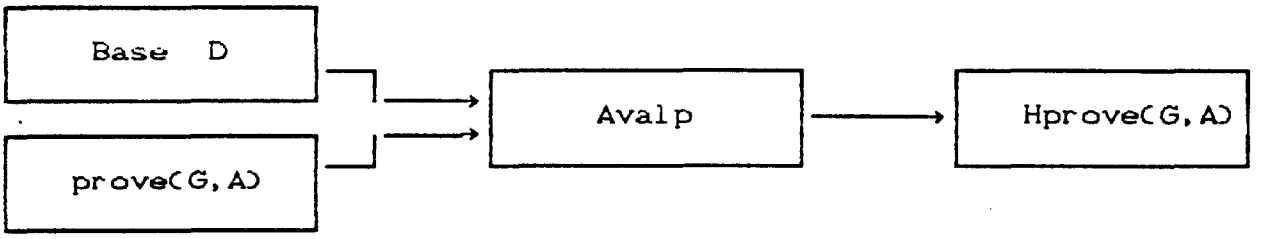


Figura 2.11 Avaliação parcial de prove(G,A) em relação a Base D

Hprove é a base B (regras do cálculo dos predicados) transformada em sentenças do PROLOG.

Na 3ª etapa, apliquemos uma avaliação parcial (aval) ao avaliador parcial da figura 2.11 (ver figura 2.5).

```
aval(  avalp(H :- Cd, prove(G,A), Hprove(G,A)),
       prove(G,A),
       avalprove( H :- Cd, Hprove(G,A))  )
```

Neste caso a figura 2.5 é representada pela figura 2.12

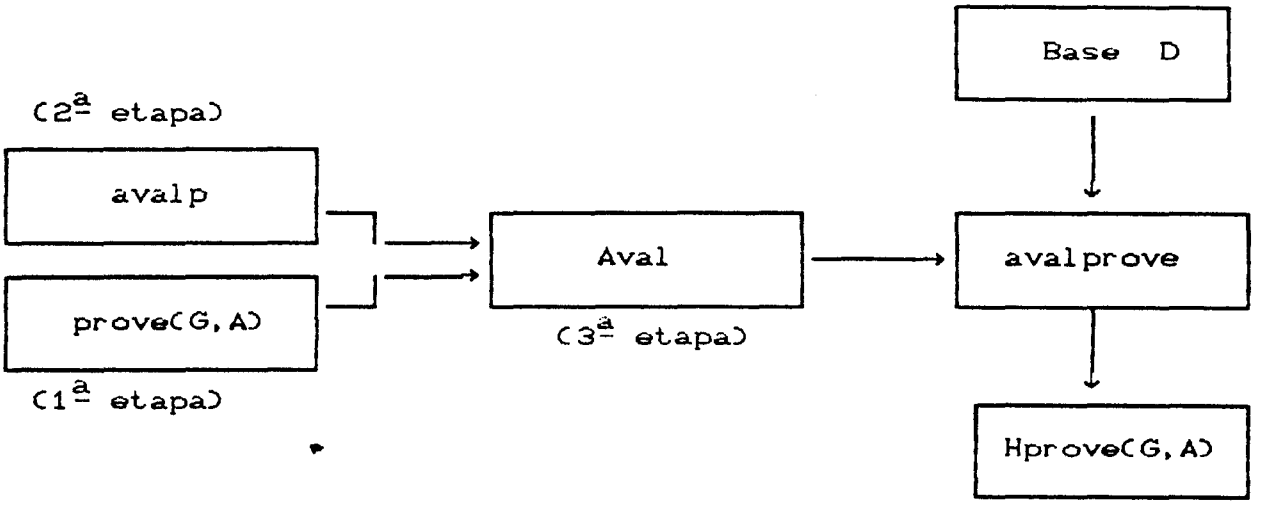


Figura 2.12 Avaliação parcial de avalp em relação a prove(G,A). Obtemos avalprove, que é o avaliador parcial especializado para o engenho de inferência B.

Note que, no momento da aplicação de aval, o único dado conhecido é o programa prove. Devido a isto, obteve-se um avalprove(H :- Cd, Hprove(G,A)), que recebe uma base de dados D e produz um Hprove(G,A) (programa equivalente em PROLOG). O comportamento de Hprove(G,A) é idêntico ao de prove(G,A) na presença das sentenças H :- Cd, ou seja, dada uma consulta G, obtemos a mesma resposta R.

Resumindo, temos inicialmente que:

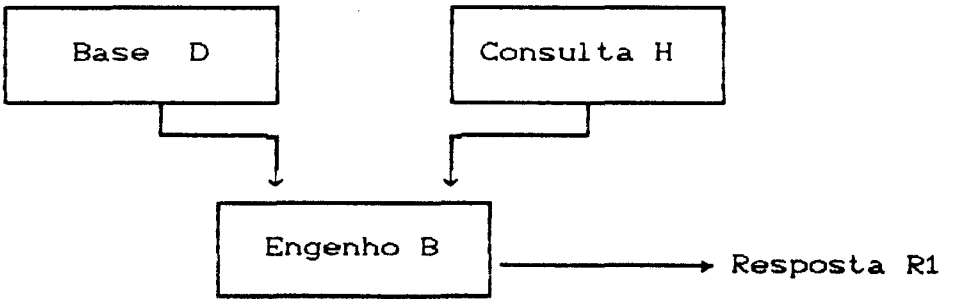


Figura 2.13 Dada a consulta H, o engenho B produz resposta R1

Após as três etapas realizadas para vencer o fosso semântico temos a figura 2.14.

Construímos manualmente um avalprove(H :- Cd, Hprove(G,A)), tal que:

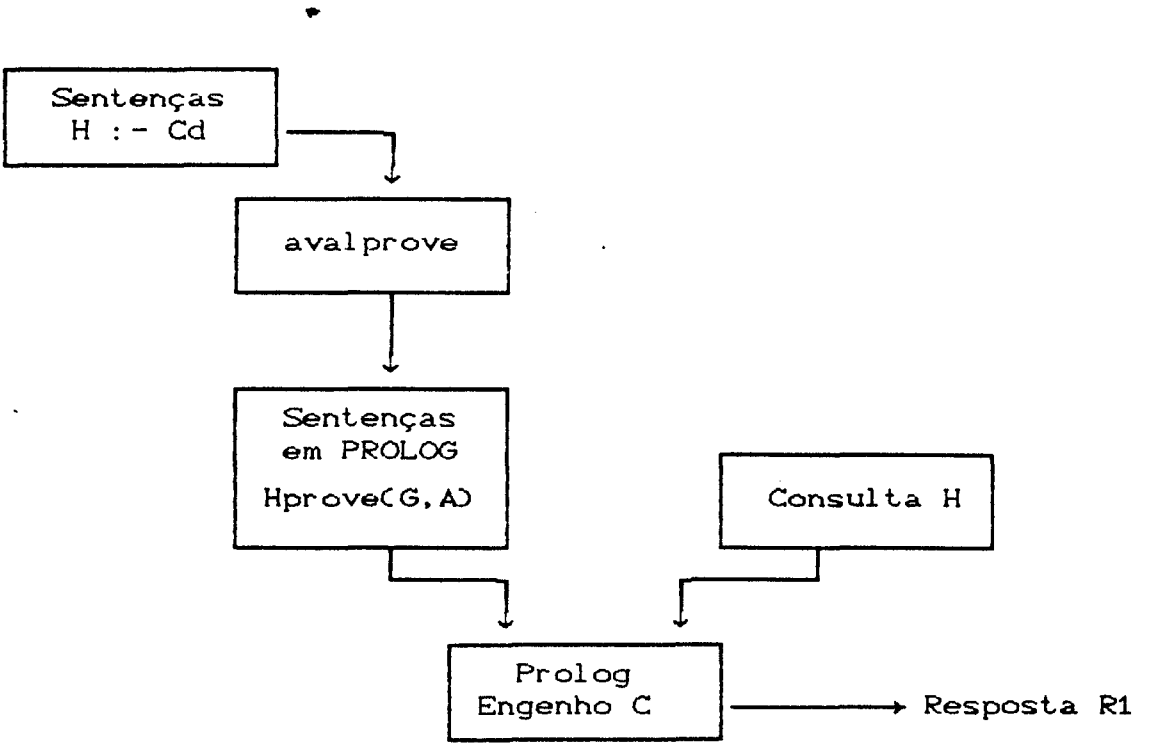


Figura 2.14 Dada a consulta H, o engenho C produz resposta R1

Temos que dada a consulta H, o engenho C produz a mesma resposta R1 que o engenho B.

Avalprove foi construído fazendo avaliação manual de prove. Vejamos a listagem de avalprove.

Listagem de avalprove

```
neg(not(not(X)), Y) :- neg(X,Y).
neg(not(X), X) :- !.
neg(X, not(X)).


or((X ; Y), Z, [NX|R]) :- !,
      neg(X, NX),
      or(Y, Z, R).
or(X, Z, [NX|Z]) :- neg(X,NX).


 and((X , Y), [X|R]) :- !,
      and(Y,R).
 and(X, [X]).


 norm((X :- Y), N) :- !,
      and(Y, NY),
      or(X, NY, N).
 norm((X ; Y), Z) :- !,
      or((X ; Y), [], Z).
 norm((X , Y), Z)  :-
      and((X , Y), Z).
 norm(X, [Y]) :- neg(X, Y).


 propagar_frente(not(F), A, NF) :- !,
      F =.. [FN|ARGS],
      name(FN, L),
      name(NFN, [110|L]),
      NF =.. [NFN, A|ARGS].
 propagar_frente(F, A, FA) :-
      F  =.. [FN|ARGS],
      FA =.. [FN,A|ARGS].
```

```
seg(F, (H :- on(NF,A)) ) :-
     propagar_frente(F, A, H),
     neg(F, NF).


armz((H :- T)) :- clause(H,T),!.
armz(SX) :- assertz(SX).


propagar_tras([X],A,Y) :- !,
     seg(X,SX),
     armz(SX),
     propagar_frente(X, A, Y).
propagar_tras([], A, true) :- !.
propagar_tras([X|Y], A, (NX,NY)) :-
     propagar_frente(X, A, NX),
     seg(X, SX),
     armz(SX),
     propagar_tras(Y, A, NY).


 avalp(F, [], H) :- !,
      propagar_frente(F, A, H).
 avalp(F, B, (H :- not( on(F,A)), R) ) :-
      propagar_frente(F, A, H),
     *propagar_tras(B, [F|A], R).

 contra_posit([X|R], NX, R) :- neg(X,NX).
 contra_posit([X|R], N, [N|NR]) :-
      contra_posit(R, N, NR).


 processe(end_of_file) :- !.
 processe(X) :-
      norm(X,Y),
      contra_posit(Y, F, C),
      avalp(F, C, S),
      assertz(S),
      fail.

 on(X, [X|Y]) :- !.
 on(X, [_|Z]) :- on(X,Z).


 rec(A) :- abolish(se/1),
       see(A),
       repeat,
              read(F),
              processe(F),!,
       seen.
```

Vejamos uma explicação detalhada de cada predicado de avalprove.

Os predicados neg(_,_), or(_,_,_), and(_,_), norm(_,_), on(_,_), rec(_), são os mesmos da listagem do engenho B. Vejamos os demais predicados.

O predicado propagar_frente(Arg1, Arg2, Arg3) recebe como entrada o predicado que será avaliado, Arg1, e um meta-argumento Arg2 (argumento da meta-regra prove, ou seja, a lista que guarda os predicados já avaliados). Fornece como saída Arg3, um predicado equivalente a Arg1, acrescentado o argumento extra (Arg2).

A 1ª cláusula diz que se o predicado Arg1 for not(Funtor(A1,A2,...,An)), então Arg3 será um novo predicado da forma nFuntor(A1, A2,...., An, Arg2), ou seja, concatena-se a letra 'n' ao Funtor, obtendo-se nFuntor.

Vejamos um exemplo. Dado propagar_frente(not humano(X), A, NP), obteremos NP = nhumano(A, X).

A 2ª cláusula é para Arg1 com predicados afirmativos. Neste caso, obteremos o predicado com o mesmo funtor, porém com um novo argumento. Vejamos um exemplo. Dado propagar_frente(humano(X),A,P), obteremos P = humano(A, X).

O predicado seg(Arg1, Arg2) recebe como entrada o predicado a ser avaliado, Arg1, executa o predicado

propagar_frente(Arg1, A, PArg1) para avaliar Arg1, e fornece a cláusula (PArg1 :- on(NArg1, A)), onde NArg1 é a negação do predicado Arg1. Vejamos um exemplo. Dado seg(homem(wang), PArg1) obteremos PArg1 = homem(A, wang) e a cláusula Arg2 = homem(A,wang) :- on(not homem(wang), A), onde a cabeça da cláusula obtida equivale ao Arg1 acrescentado de um meta-argumento (avaliado parcialmente), e o corpo da forma on(Negação_de_Arg1, Meta_argumento).

O predicado armz(Arg1) recebe como entrada a cláusula Arg1 e a armazena na base de dados.
A 1ª cláusula verifica se a cláusula Arg1 já está armazenada na base de dados. Se encontrá-la, então não devemos armazená-la novamente.
A 2ª cláusula armazena a cláusula Arg1 na base de dados.

O predicado propagar_tras(Arg1, Arg2, Arg3) recebe como entrada uma lista Arg1 de predicados a serem avaliados, um meta-argumento Arg2, e fornece como saída Arg3, que é uma lista correspondente a avaliação parcial da lista Arg1. Temos que a avaliação parcial de cada predicado consiste em colocar o meta argumento no predicado (usando-se propagar_frente). O predicado propagar_tras aplica o predicado propagar_frente para cada elemento da lista a ser avaliada e, ao mesmo tempo armazena, caso ainda não esteja armazenada, a cláusula formada pelo predicado seg.

A 1ª cláusula de propagar_tras(Arg1, Arg2, Arg3) é para o caso da lista a ser avaliada, Arg1, conter somente um predicado. Temos que Arg2 é um meta_argumento, e Arg3 é uma lista contendo apenas o predicado avaliado de Arg1. Neste caso, forma-se a claúsula (Arg1_avaliado_com_meta_arg :- on(Negação_de_arg1, Meta_arg), que é armazenada (caso ainda não tenha sido).

A 2ª cláusula de propagar_tras é para o caso em que a lista a ser avaliada, Arg1 é vazia, ou seja, não há predicado a ser avaliado.

A 3ª cláusula de propagar_tras é para o caso de haver mais de um elemento na lista a ser avaliada, Arg1. Neste caso, executa o predicado propagar_frente para o primeiro elemento da lista, e chama recursivamente propagar_tras para que os demais predicados sejam avaliados.

O predicado avalp(Arg1, Arg2, Arg3) tem como entrada Arg1, a cabeça de uma cláusula para ser avaliada e Arg2, o corpo da cláusula a ser avaliado. Fornece como saída Arg3, a cláusula avaliada (cuja cabeça é o predicado Arg1 avaliado parcialmente, aplicando-se o predicado propagar_frente e cujo corpo é o predicado Arg2 avaliado parcialmente, aplicando-se o predicado propagar_tras).

A 1ª cláusula avalp(Arg1, Arg2, Arg3) é para o caso de cláusulas unitárias (sem corpo, ou seja, Arg2 = []). A cláusula obtida Arg3 (também unitária) equivale ao Arg1 avaliado parcialmente (aplicando-se propagar_frente).

A 2ª cláusula avalp(Arg1, Arg2, Arg3) é para o caso onde a

cláusula a ser avaliada possui um corpo. Aplica-se propagar_frente(Arg1, A, PArg1), onde PArg1 é Arg1 avaliado parcialmente; em seguida, aplica-se propagar_tras(Arg2,[Arg1|A],R), onde R é Arg2 avaliado parcialmente. O resultado da avaliação é a cláusula PArg1 :- not(on(Arg1,A)), R.

O predicado contra_posit(Arg1, Arg2, Arg3) recebe como entrada Arg1, uma lista de predicados a serem avaliados. Fornece como saída Arg2, a negação do primeiro elemento da lista Arg1, e Arg3 equivale ao Arg1 sem o primeiro elemento. Note que, sendo Arg1 uma cláusula representada por uma lista, temos que Arg2 é a cabeça e Arg3 o corpo.
A 1ª cláusula de contra_posit(Arg1, Arg2, Arg3), fornece Arg2, que equivale a negação do primeiro elemento de Arg1, e Arg3, que equivale a Arg1 sem o primeiro elemento.
A 2ª cláusula é para que sejam devolvidas as negações (cabeças)de outros elementos de Arg1 e seus respectivos corpos.

O predicado processe(Arg1) recebe uma regra Arg1 como entrada, transforma-a em uma lista de predicados (LN) afirmados ou negados (aplicando-se o predicado norm), executa o predicado contra_posit para obter de cada elemento da lista (LN) a sua negação (cabeça) e a lista sem o elemento (corpo), avalia parcialmente a cláusula, chamando o predicado avalp e insere na base de dados a cláusula avaliada parcialmente. Este predicado falha para que sejam

fornecidas por contra_posit outras cabeças e corpos.

Temos que avalprove é um avaliador parcial especializado em prove. Desta forma, avalprove simula prove usando informaçoẽs da base de conhecimentos.

Para executarmos avalprove devemos seguir os seguintes passos:

1º Passo: Carregar o programa avalprove.

```
?-    [-'avalprove'].
```

2º Passo: Executar o predicado rec('baseconh') para transformar as regras da base de conhecimentos, que estão na forma de sentenças do cálculo de predicados, para sentenças de PROLOG, obtendo-se assim Hprove(G,A), que será manipulado diretamente pelo engenho de inferência do PROLOG. Hprove(G,A) é o resultado da avaliação parcial de prove(G,A) em relação a base de conhecimentos, ou seja, Hprove é o programa prove especializado para a base de conhecimentos.

```
?-   rec('baseconh').
```

Como vimos, após a execução do passo 2, obteremos o seguinte programa Hprove(G,A).

Hprove

```
humano(A, B) :-   not   on(humano(B), A),
                  homem( [humano(B) | A], B).

humano(A, B) :-   not   on(humano(B), A),
                  mulher( [humano(B) | A], B).
```

```
nhumano(A, B) :-  on(humano(B), A).


nhomem(A, wang) :- on( homem(wang), A).

nhomem(A, ling) :- on( homem(ling), A).

nhomem(A, B) :-     not on(not homem(B), A),
                    nhumano( [not homem(B) | A], B).


homem(A, wang) :-   not  on(homem(wang), A),
                    nmulher([homem(wang) | A], wang).

homem(A, ling) :-   not  on(homem(ling), A),
                    nmulher([homem(ling) | A], ling).

homem(A, B) :-      on( not homem(B), A).


nmulher(A, wang) :-  on( mulher(wang), A).

nmulher(A, ling) :-  on( mulher(ling), A).
nmulher(A, B) :-     not    on(not mulher(B),  A),
                     nhumano([not mulher(B) | A], B).


mulher(A, wang) :-  not on(mulher(wang), A),
                    nhomem([mulher(wang) | A], wang).
mulher(A, ling) :-  not on(mulher(ling), A),
                    nhomem([mulher(ling) | A], ling).

mulher(A, B) :-     on( not  mulher(B), A).


p(X) :- X =.. [Fn|Args],
        NX =.. [Fn, []|Args],
        call(NX).
```

Para examinarmos como processe transforma as regras do cálculo de predicados em sentenças PROLOG, tomemos apenas um exemplo:

```
?- processe( (homem(wang) ; mulher(wang)) ).

processe( (homem(wang) ; mulher(wang)) ).

 norm( (homem(wang) ; mulher(wang)), FN)
 FN = [not homem(wang), not mulher(wang)]

 contra_posit([not homem(wang), not mulher(wang)], F, C)
   F = homem(wang)
   C = [not mulher(wang)]]

 avalp( homem(wang), [not mulher(wang)], S)

   propagar_frente( homem(wang), A, H)
     H = homem( A, wang)

   propagar_tras([not mulher(wang)], [homem(wang)|A], R)

   seg(not mulher(wang), SX),

       propagar_frente(not mulher(wang), A1, H1)
       H1 = nmulher(wang, A1)
       neg(not mulher(wang), NG)
       NG = mulher(wang)
     SX = (nmulher(wang, A1) :- on(not mulher(wang),A1))

     Armz(nmulher(wang, A1) :- on(not mulher(wang),A1))

     propagar_frente( not mulher(wang), [homem(wang)|A], R)
     R = nmulher([homem(wang)|A],wang)

   S = ( homem(A, wang) :- not( on( homem(wang),A)),
                           nmulher([homem(wang)|A], wang) )
```

A partir deste momento pode-se iniciar as sequências de consultas. As consultas devem ser realizadas da seguinte maneira:

```
?- p(Consulta).
```

A consulta p(Consulta) será executada diretamente pelo engenho de inferência do PROLOG, portanto, será mais eficiente que a consulta p(Consulta) executada no engenho B. Para compararmos tal

eficiência vejamos como o engenho PROLOG executa a consulta p(humano(X)), já executada pelo engenho B.

Dada a consulta: ?- p(humano(X)).

Respostas:
```
X = wang  ->  ;
X = ling  ->  ;
X = wang  ->  ;
X = ling
```

PROLOG executará as seguintes inferências:

```
p(humano(X))
    NX = humano([],X)
    call(humano([],X))
```

Há duas cláusulas para humano(_,_),PROLOG escolhe a primeira

```
humano(A,B) :- not on(humano(B),A),
               homem( [humano(B)|A], B).
```

```
executa on(humano(B), []) -->. falha, então
        not on(humano(B),[])   sucede
```

```
executa homem( [humano(B)], B)
```

Há tres cláusulas para homem(_,_), PROLOG escolhe a primeira

```
homem(A, wang) :- not on(homem(wang),A),
                  nmulher([homem(wang)|A],wang).
```

```
executa on(homem(wang), [humano(B)])  --> falha, então
        not on(homem(wang),[humano(B)])   sucede
```

```
executa nmulher([homem(wang), humano(wang)], wang)
```

Há três cláusulas para nmulher(_,_), PROLOG unifica-a com primeira

```
nmulher(A, wang) :- on(mulher(wang),A).
```

```
executa  on(mulher(wang), [homem(wang),humano(wang)])  falha
```

```
então    not on(mulher(wang),[homem(wang)])            sucede
```

PROLOG tenta a terceira cláusula

```
nmulher(A, B) :- not (on mulher(B),A),
```

```
                    nhumano([not mulher(B)|A],B).

executa on(mulher(wang),[homem(wang),humano(wang)])     falha

então not(on(mulher(wang),[homem(wang),humano(wang)])) sucede

executa nhumano([not mulher(wang),homem(wang),humano(wang)],
                wang)

Há uma cláusula para nhumano(_,_), PROLOG tenta a primeira

nhumano(A,B) :- on(humano(B),A)

executa on(humano(wang,[not mulher(wang),homem(wang),
                        humano(wang)])) sucede

Portanto, wang é humano.
```

Finalmente delineamos a construção de aval, que faz a avaliação parcial de avalp(H :- Cd, prove(G,A), Hprove(G,A)) em relação a prove, ou seja, produz avalprove (figura 2.12). Em outras palavras, dado prove como entrada, nosso terceiro programa (aval) nos fornece avalprove. É interessante notar que foi feita uma avaliação parcial de programas que realmente não existiam (avalp da figura 2.12). Quando realizamos a avaliação parcial de avalp(H :- Cd, prove(G,A), Hprove(G,A)), avalp somente "existe na nossa imaginação". Além disso, aval funciona sem exigir a "existência real" de avalp. De fato, a estrutura de avalp está implícita no código de aval.

A listagem abaixo é de um aval simplificado, correspondente a figura 2.12.

Listagem do aval simplificado

```
pr( ( prove(G, A) :- neg(G, NG), on(NG,A)), fato(G)).
pr( (prove(G, A) :- not(on(G,A)), pro(B,[G|A])), sent(G,B)).

fato(G) :- abolish(jafornecido/1),
           se(L),
           percorre(L,G).

percorre([X|Y], FX) :-  functor(X, Fn, Aridade),
                        functor(FX, Fn, Aridade),
                        not(jafornecido(FX)).
percorre([X|Y], Z) :- percorre(Y, Z).


adnot(G, NG) :- G =.. [Fn|Args], name(Fn, LFn),
                name(NFn,[110|LFn]), NG =.. [NFn|Args].

geracab(not(G), [NFnG|Args]) :-
            G =.. [FnG|Args], adnot(FnG, NFnG).
geracab(G, [FnG|Args]) :- G \= not(_), G =.. [FnG|Args].

geraval(XX) :- pr((P :- Cd), F),
               P =.. [_, G|Args],
               XX = (aval((Cab :- Cd)) :-
                      F, geracab(G, [FnG|ArgG]),
               Cab =.. [FnG|ArgCab]),
               append(Args, ArgG, ArgCab).

append([], X, X).
append([X|Y], Z, [X|W]) :- append(Y, Z, W).
```

Para executarmos aval (simplificado) e obtermos avalprove (simplificado), devemos seguir os seguintes passos:

1º Passo: Carregar o programa aval.

```
?-   ['aval'].
```

2º Passo: O predicado geraval deve ser aplicado a cada predicado de prove para gerar avalprove (simplificado), ou seja, um

avaliador especializado em prove.

```
?- geraval(L), assertz(L),fail.
```

Temos que geraval gera a cláusula L, que o predicado assertz(L) armazena na base de conhecimento e falha para que outras cláusulas sejam geradas.

3º Passo: Se digitarmos listing(aval), veremos os predicados (formando avalprove) gerados pelo 2º passo.

```
?- listing(aval).
```

Obteremos:

```
aval( (A :- neg(B, C), on(C,D)) ) :-
        fato(B),
        geracab(B, [E|F]),
        A =.. [E, D|F].

aval( (A :- not on(B,C), pro(D, [B|C])) ) :-
        sent(B, D),
        geracab(B, [E|F],
        A =.. [E,C|F].
```

Para executarmos o avalprove simplificado, gerado pelo 3º passo para obter Hprove (sentenças em PROLOG), devemos seguir os seguintes passos:

4º Passo: Carregar o engenho B.

```
?- [-'engenhob'].
```

5º Passo: Transformar as regras da base de conhecimentos 'baseconh', que estão na forma de cálculo de predicados,

para a forma normal.

```
?- rec('baseconh').
```

6º Passo: Obter Hprove (baseconh convertida para sentenças em PROLOG):

```
?- aval(L).
```

Obteremos:

```
L = homem(X, Y)     :- neg(homem(Y), Z),
                       on(Z, X).

L = homem(X, wang)  :- not  on(homem(wang), X),
                       pro([not mulher(wang)],
                           [homem(wang)|X]).

L = homem(X, ling)  :- not  on(homem(ling), X),
                       pro([not mulher(ling)],
                           [homem(ling)|X]).

L = mulher(X, Y)    :- neg(mulher(Y), Z),
                       on(Z, X).

L = mulher(X, wang) :- not  on(mulher(wang),X),
                       pro([not homem(wang)],
                           [mulher(wang)|X]).

L = mulher(X, ling) :- not  on(mulher(ling),X),
                       pro([not homem(ling)],
                           [mulher(ling)|X]).

L = humano(X, Y)    :- not  on(humano(Y),X),
                       pro([homem(Y)],
                           [humano(Y)|X]).

L = humano(X, Y)    :- not  on(humano(Y),X),
                       pro([mulher(Y)],
                           [humano(Y)|X]).

L = nhomem(X, Y)    :- not  on( not homem(Y), X),
                       pro([not humano(Y)],
                           [not homem(Y)|X]).
```

```
L = nmulher(X, Y) :-   not  on(not  mulher(Y),X),
                       pro([not humano(Y)],
                           [not mulher(Y)|X]).
```

Sterling e Takeuchi [Sterling-86, Takeuchi-85], descrevem uma ferramenta para transformar um sistema especialista composto de uma base de conhecimentos e uma coleção de meta-intérpretes PROLOG em um eficiente programa escrito em PROLOG. Este processo é executado em duas etapas: primeiro avalia parcialmente o meta-intérprete em relação a base de conhecimentos, ou seja, especializa o meta-intérprete para a base de conhecimentos; na segunda etapa coloca os meta-argumentos dos meta-intérpretes nas regras da base de conhecimentos, transformando-as desta forma em um programa PROLOG. Este método corresponde as figuras 2.2 e 2.11.

O método proposto por Takeuchi [Takeuchi-85] a respeito da avaliação parcial de base de conhecimentos incremental, corresponde as figuras 2.5 e 2.12.

(P4) APRENDIZADO BASEADO EM EXPLICAÇÕES

A generalização baseada em explicações (EBG) é uma técnica usada em aprendizagem de máquina [Harmelen-88] para formular conceitos gerais com base em exemplos testes específicos.

Como vimos, a avaliação parcial vem sendo usada na comunidade de programação lógica como um método de otimização de programas. Dado um programa (definição de uma função), junto com uma especificação parcial da entrada obteremos, após a avaliação, uma nova versão do programa menos geral (programa especializado para os valores de entrada fornecidos) porém, mais eficiente que a versão original.

[Harmelen-88] mostra que, no contexto da programação lógica, a avaliação parcial e a generalização baseada em explicação, embora desenvolvidas para propósitos diferentes, consistem do mesmo algoritmo que pode ser implementado com apenas algumas diferenças. Para mostrar a equivalência entre as duas técnicas, primeiro analisa suas definiçoes, apresenta suas implementações simplificadas, compara seus programas e através de um exemplo, mostra como a computação em EBG corresponde para avaliação parcial.

O algoritmo da generalização é o seguinte:
Analisa o exemplo de teste em termos dos conhecimentos do domínio

e do conceito meta (a ser aprendido). Produz uma explicação de porquê o exemplo teste é uma instanciação do conceito meta. Esta explicação é então generalizada, ou seja, é feita uma abstração do exemplo teste particular para formular a definição geral.

Temos as seguintes entradas de dados para o algoritmo de generalização baseada em explicação:

(E1) Conceito meta: Uma definição do conceito a ser aprendido.

(E2) Exemplo teste: Uma instanciação do conceito meta.

(E3) Teoria do domínio: Um conjunto de regras para ser usada na explicação de porquê o exemplo teste é uma instanciação do conceito meta.

(E4) Critério de Operacionalidade: Um critério sobre definição do conceito, que especifica a forma com que a definição do conceito deve ser expressa. Este critério define um conjunto de predicados facilmente avaliados da teoria do domínio.

Podemos comparar as entradas de dados dos algoritmos da generalização baseada em explicação e da avaliação parcial.

- O conceito meta (E1) corresponde ao nome do programa a ser avaliado parcialmente.
- O exemplo teste (E2) corresponde aos valores de entrada das variáveis do programa a ser avaliado parcialmente.
- A teoria do domínio (E3) corresponde as cláusulas e fatos que

constituem o programa a ser avaliado parcialmente.

- O critério de operacionalidade (E4) corresponde ao critério que o avaliador parcial utiliza para parar a construção da árvore de prova.

Vamos apresentar uma implementação do algoritmo de generalização baseado em explicação e mostrar como um conceito meta pode ser aprendido, baseando-se em um exemplo teste específico.

LISTAGEM DO ALGORITMO DE GENERALIZAÇÃO

```
apren(_) :- retract(resp(L)), fail.
apren(R) :- functor(R, Fn, Aridade),
        functor(RG, Fn, Aridade),
        ebl(R, RG, Resp),
        generaliza(f(RG,Resp), f(Cab,RespG)),
        RG = R,
        simplifica(RespG, Resp, RespS),
        nomeia_vars(RespS, 1, _),
        prettywrite((Cab :- RespS)), fail.


ebl(L, GL, GL) :- opera(L), !.
ebl( (G1, G2), (GG1, GG2), (L1, L2) ) :-
        ebl(G1, GG1, L1),
        ebl(G2, GG2, L2).
ebl( G, GG, L) :-
        clause(GG, GC),
        generaliza( (GG :- GC), (G1 :- C) ),
        G1 = G,
        ebl(C, GC, L).


generaliza(P, Q) :- generaliza(P, Q, [], _).

generaliza(P, Q, V1, V1) :- atomic(P), !, P=Q.
generaliza(P, Q, V1, V1) :- atomic(Q), !, P=Q.
generaliza(P, Q, V1, V1) :- var(P), var(Q),
        memb(v(P,Q), V1 ),  !.
generaliza(P, Q, V1, [v(P, Q)|V1]) :- var(P), var(Q), !.
generaliza(P, Q, V1, V2) :- functor(P, Fn, Aridade),
```

```
        functor(Q, Fn, Aridade),
        generalizargs(P, Q, 1, Aridade, V1, V2).


generalizargs(P, Q, N, Aridade, V1, V1) :- N > Aridade, !.
generalizargs(P, Q, N, Aridade, V1, V3) :-
        arg(N, P, AP),
        arg(N, Q, AQ),
        generaliza(AP, AQ, V1, V2),
        M is N+1,
        generalizargs(P, Q, M, Aridade, V2, V3).


memb(v(P, Q), [v(P1, Q)|R]) :- P == P1, !.
memb(V, [X|R]) :- memb(V, R).


simplifica(C, CG, C1) :-
        transforma_em_lista(C, LC),
        transforma_em_lista(CG, LCG),
        elimina_superfluos(LC,LCG, LS),
        transforma_em_conj(LS, C1).


transforma_em_lista(C, [C]) :- C \= (_,_), !.
transforma_em_lista((C,S), LC) :-
        transforma_em_lista(C, C1),
        transforma_em_lista(S, S1),
        appen(C1, S1, LC).


appen([], X, X) :- !.
appen([X|Y], Z, [X|W]) :- appen(Y, Z, W).


transforma_em_conj([X], X) :- !.
transforma_em_conj([X,Y], (X,Y)) :- !.
transforma_em_conj([X|RX], (X,RC)) :-
        transforma_em_conj(RX, RC).


elimina_superfluos([R], [R1], [R]) :- !.
elimina_superfluos([R1|RR], [S1|RS], [R1|RT]) :-
        resp(S1), !, deletar(S1, RR, RS, PR, PS),
        elimina_superfluos(PR, PS, RT).
elimina_superfluos([R1|RR], [S1|RS], [R1|RT]) :-
        elimina_superfluos(RR, RS, RT).


deletar(S1, [], [], [], []) :- !.
deletar(S1, [R|RR], [S1|RS], PR, PS) :- !,
        deletar(S1, RR, RS, PR, PS).
deletar(S1, [R|RR], [S|RS], [R|PR], [S|PS]) :-
```

```
        deletar(S1, RR, RS, PR, PS).


prettywrite((R :- C)) :-
        write(R), write(' :- '), nl,
        prettycauda(C), nl.
prettycauda((C,R)) :-
        write('              '),write(C),
        write(', '), nl, prettycauda(R).
prettycauda(C) :-
        write('              '),   C \= (_,_),
        write(C), write('.').


opera(true).
opera(L) :- functor(L, Fn, Aridade),
        system(Fn/Aridade), !,
        call(L).
opera(L) :- L \= (A,B),
        L \= (A :- B),
        L \= (A ; B),
        functor(L, FnL, Aridade),
        functor(Teoria, FnL, Aridade),
        \+ clause(Teoria, _),
        nl,
        pergunta(L).


pergunta(L) :- resp(L), !.
pergunta(L) :-
        functor(L, Fn, Aridade),
        grounded(L, 1, Aridade), !,
        write('Eh verdade que '), write(L), write('?'), nl,
        analise_sim_nao('h'), assertz(resp(L)).
pergunta(L) :- functor(L, Fn, Aridade),
        functor(Q, Fn, Aridade),
        L =.. [Fn|Args],
        Q =.. [Fn|ArgsQ],
        nomeie_variaveis(1,Args, ArgsQ, Nms, LNms ),
        leia_lista(Q, Nms, [], LNms), assertz(resp(L)).


um_nome(1,'X'). um_nome(2,'Y'). um_nome(3,'Z').
um_nome(4,'W'). um_nome(5,'P'). um_nome(6,'Q').
um_nome(7,'R'). um_nome(8, 'S').um_nome(9,'T').
um_nome(10,'U'). um_nome(11, 'V'). um_nome(12, 'W').


leia_lista(Q, Nms, nada,_) :- !, fail.
leia_lista(Q, Nms, nao, _) :- !, fail.
leia_lista(Q, Nms, LNms, LNms) :- !.
leia_lista(Q, [Nm], _, LNms) :-
        write('Tenho que '), write(Q), nl,
```

```
        write('Preciso saber os valores de '),
        write(Nm), nl,
        write('Digite-o: '),
        read(Lista),
        leia_lista(Q, Nms, [Lista], LNms).
leia_lista(Q, Nms, _, LNms) :-  !,
        write('Tenho que '), write(Q), nl,
        write('Preciso saber os valores de '),
        write(Nms), nl,
        write('Digite-os em forma de lista'), nl,
        write('Assim: [2,5,3,66].'), nl,
        read(Lista),
        leia_lista(Q, Nms, Lista, LNms).


nomeie_variaveis(N,[], [], [], []) :- !.
nomeie_variaveis(N,[X|RL], [Y|RQ], [Y|RArgsQ], [X|RArgs]) :-
        var(X), um_nome(N,Y), M is N+1,
        nomeie_variaveis(M,RL, RQ, RArgsQ, RArgs).
nomeie_variaveis(N,[X|RL], [X|RQ], RArgsQ, RArgs) :-
        nonvar(X), nomeie_variaveis(N,RL, RQ, RArgsQ, RArgs).


grounded(L,N, Aridade) :- N > Aridade, !.
grounded(L, N, Aridade) :-
        arg(N, L, AP),
        nonvar(AP),
        M is N+1,
        grounded(L, M, Aridade).


analise_sim_nao('n') :- !, fail.
analise_sim_nao('N') :- !, fail.
analise_sim_nao('S') :- !.
analise_sim_nao('s') :- !.
analise_sim_nao(_) :-
        write('Responda apertando as teclas s(im) ou n(ao)'), nl,
        get0(X), analise_sim_nao(X).


nomeia_vars(P,N,M) :- var(P), um_nome(N,P), !, M is N+1.
nomeia_vars(P, N,N) :- var(P), !.
nomeia_vars(P, N, N) :- atomic(P), !.
nomeia_vars(P, N, M) :- functor(P, Fn, Aridade),
        nomeia_args(P, 1, Aridade, N, M).


nomeia_args(P, N, Aridade, V1, V1) :- N >  Aridade, !.
nomeia_args(P, N, Aridade, V1, V3) :- arg(N, P, AP),
        nomeia_vars(AP, V1, V2),
        M is N+1,
        nomeia_args(P, M, Aridade, V2, V3).
```

Exemplo1: Suponhamos que queremos aprender sobre o conceito de ligar uma máquina M em uma tomada T, fornecendo as seguintes entradas:

Conceito Meta

```
ligar_maquina(M, T) :- compativel(M, T).
```

Esta regra representa a sentença: É possível ligar a máquina M na tomada T se M e T forem compatíveis.

Exemplo

```
*ligar_maquina(geladeira,t1).
```

Esta regra representa a sentença: Sabe-se que é possível ligar geladeira na tomada t1.

Teoria do domínio

```
compativel(M,T) :-
        mesma_classe_tensao(M,T),
        corrente_inferior(M,T).
```

Esta regra representa a sentença: Máquina M e tomada T são compatíveis se M e T pertencem a mesma classe de tensão e a corrente de M é inferior a corrente de T.

```
mesma_classe_tensao(M,T) :-
        tensao(M,V1),
        tensao(T,V2),
```

```
        DV is abs((V1 -V2)/V2),
        DV =< 0.05.
```

Esta regra representa a sentença: A máquina M e tomada T pertencem a mesma classe de tensão se V1 é a tensão de M e V2 é a tensão de T e DV é igual ao valor absoluto de V1 menos V2 dividido por V2 e DV é menor ou igual a 0.05.

```
corrente_inferior(M,T) :-
        corrente(M,I1),
        corrente(T,I2),
        I1 =< I2.
```

Esta regra representa a sentença: Máquina M tem corrente inferior a tomada T se a corrente de M é I1 e a corrente de T é I2 e I1 é menor ou igual a I2.

```
corrente(X,I) :-
        tensao(X,V),
        potencia(X,P),
        I is P/V/sqrt(3).
```

Esta regra representa a sentença: A corrente de X é I se tensão de X é V e a potência de X é P e I é igual a P dividido por V e o resultado dividido pela raiz quadrada de 3.

Note que as regras compativel(M,T), mesma_classe_tensao(M,T), corrente_inferior(M,T) e corrente(X,I) serão usadas para explicar o porquê é possível ligar a geladeira na tomada t1.

Critério de Operacionalidade

```
opera(true).
opera(L) :-
        funtor(L, Fn, Aridade),
        system(Fn/Aridade),!,
        call(L).
```

```
opera(L) :-
        L \= (A,B),
        L \= (A :- B),
        L \= (A ;  B),
        funtor(L,  FnL, Aridade),
        funtor(Teoria, FnL, Aridade),
        \+ clause(Teoria,_),
        nl,
        pergunta(L).
```

Durante a definição do conceito é necessário um critério de operacionalidade. No algoritmo de generalização implementado,este critério é definido pelo predicado opera. Na 1ª cláusula temos que, se o predicado a ser analisado é true, não há nada a fazer. Na 2ª cláusula temos que se o predicado pode ser executado, opera irá executá-lo. Na 3ª cláusula temos que, se não tivermos informaçoẽs sobre o predicado analisado, o sistema faz pergunta ao usuário a fim de obter tais informações.

Para obtermos o conceito geral de ligar_tomada devemos executar o seguinte predicado.

```
?-  aprender( ligar_tomada(geladeira,t1)).
```

O sistema fará as seguintes perguntas ao usuário:

```
"Tenho que tensao(geladeira,X)
 Preciso  saber os valores de X
 Digite-o:  "
 Podemos digitar  110.
```

"Tenho que tensao(t1,X)

Preciso saber os valores de X

Digite-o: "

Podemos digitar 110.

"Tenho que potencia(geladeira,X)

Preciso saber os valores de X

Digite-o: "

Podemos digitar 1000.

"Tenho que potencia(t1,X)

Preciso saber os valores de X

Digite-o: "

Podemos digitar 1100.

Em seguida fornecerá o seguinte conceito de ligar_maquina:

```
ligar_maquina(X,Z) :-
        tensao(X,Y),
        tensao(Z,W),
        P is abs((Y -W) / W,
        P =< 0.05,
        potencia(X,Q),
        R is  Q / S /  sqrt(3),
        potencia(Z,T),
        U is T/ V / sqrt(3),
        R =< U.
```

Vejamos um outro exemplo:

Exemplo2: Suponhamos que queremos aprender sobre o conceito de empilhar, fornecendo as seguintes entradas:

Conceito Meta

```
empilhar(X,Y) :-
        suporta(Y, Peso),
        pesa(X, W),
        W < Peso.
```

Esta regra representa a sentença: É possível empilhar X em Y se Y suporta um certo Peso e X pesa W e W é menor que Peso.

Exemplo

```
empilhar(caixa, mesa).
```

Esta regra representa a sentença: Sabe-se que é possível empilhar caixa na mesa.

Teoria do domínio

```
pesa(X, W) :-
        contem(X, Objs),
        existem(N, Objs),
        peso(Objs, P),
        W is N * P.
```

Esta regra representa a sentença: X pesa W se X contém um certo objeto Objs e existe um número N de Objs e cada Objs pesa P e W é N vezes P.

```
pesa(X, W) :-
        peso_vale(X,W).
```

Esta regra representa a sentença: X pesa W se W é o peso de X.

O critério de operacionalidade implementado é geral para qualquer conceito que se deseja aprender.

Para obtermos o conceito geral de empilhar devemos executar o seguinte predicado.

```
?-  aprender( empilhar(caixa, mesa)).
```

O sistema fará as seguintes perguntas ao usuário:

```
"Tenho que suporta(mesa,X)
 Preciso saber  os valores de X
 Digite-o:  "
 Podemos digitar   100.

"Tenho que contem(caixa, X)
 Preciso saber  os valores de X
 Digite-o:  "
 Podemos digitar   copos.

"Tenho que existem(X, copos)
 Preciso saber  os valores de X
 Digite-o:  "
 Podemos digitar   100.

"Tenho que peso(copos, X)
 Preciso saber  os valores de X
 Digite-o:  "
 Podemos digitar   0.1  .
```

Obteremos o seguinte conceito de empilhar:

```
empilhar(Z, X) :-
        suporta(X, Y),
        contem(Z, W),
        existem(P, W),
        peso(W, Q),
        R is P * Q,
        R < Y.
```

A seguir o sistema fará a seguinte pergunta:

```
"Tenho que peso_vale(caixa, X)
 Preciso saber  os valores de X
 Digite-o:  "

 Podemos digitar   10.
```

Um outro conceito de empilhar é obtido:

```
empilhar(Z, X) :-
        suporta(X, Y),
        peso_vale(Z, W),
        W < Y.
```

Temos que, para o algoritmo de generalização obter a definição do conceito meta, primeiro faz uma avaliação parcial do conceito em relação ao exemplo teste e, em seguida, uma generalização do conceito avaliado parcialmente.

2.4 CONCLUSÃO

Neste capítulo apresentamos a teoria da avaliação parcial utilizando um sistema formal, com definição, regras de inferências e axiomas. Para ajudar a fixar o conceito de computação, foi dado um exemplo dentro do contexto de cálculo lâmbda. Mostramos as finalidades e aplicaçoẽs da avaliação parcial, mostramos os trabalhos de Sterling, Takeuchi, Venken, Ershov, entre outros, bem como a descrição de alguns algoritmos que aumentam a eficiência dos sistemas especialistas. Para auxiliar na compreensão dos algoritmos expostos, foi apresentada uma breve explicação sobre a linguagem PROLOG.

CAPÍTULO 3

INTRODUÇÃO AUTOMÁTICA DE CONTROLE EM SISTEMAS ESPECIALISTAS

Neste capítulo, vamos tratar das dificuldades de se introduzir controle automaticamente em sistemas especialistas, antes mesmo de se conhecer a consulta. Descreveremos, de maneira informal e formal, o ciclo dedutivo de um sistema especialista. Apresentaremos um algoritmo para introdução automática de controle usando avaliação parcial.

3.1 APRESENTAÇÃO INFORMAL DO CICLO DEDUTIVO DE UM S.E.

Vamos considerar uma minúscula base de conhecimentos composta por um conjunto de regras, onde cada regra obedeça a seguinte sintaxe:

<regra> ::= <conclusão> se <condições> | <fato>
<fato> ::= <predicado>
<conclusão> ::= <predicado>
<condições> ::= <literal> | <literal> & <condições>
<predicado> ::= <funtor> (<lista_de_argumentos>) | <funtor>
<literal> ::= <predicado> | nao(<predicado>)
<lista_de_argumentos> ::= <argumento> |
<argumento>,<lista_de_argumentos>
<funtor> ::= <minúscula> <minúsculas> | <minúscula>
<minúscula> ::= a | b | c | d | e | f | g | ....
<minúsculas> ::= <minúscula> | <minúscula> <minúsculas> |
_<minúsculas>

<argumento> ::= <variável> | <dado>

Consideremos a seguinte base de conhecimentos, a qual obedece a sintaxe dada acima.

(r1) pai(j, k).

(r2) pai(k, m).

(r3) pai(k, n).

(r4) avo(X, Y) se pai(X, Z) & pai(Z, Y).

Podemos fazer a seguinte consulta:

avo(j, Q) ?

Como vimos na seção 1.4 (capítulo 1), é possível tentar responder a esta pergunta através de um provador mecânico de teoremas. Este provador seria um programa que consideraria a base de conhecimentos como sendo um conjunto de sentenças lógicas e a consulta como sendo uma hipótese a ser provada. Ele tentaria provar a hipótese a partir das sentenças. O engenho de inferência da figura 1.1 seria, no caso, o próprio provador de teoremas.

A maioria dos sistemas especialistas possuem o controle embutido na base de conhecimentos (controle dado pela pessoa que construiu a base de conhecimentos), onde é importante a ordem das sentenças e dos literais dentro das condições, para que o engenho de inferência encontre as soluções das consultas. Neste caso, a base de conhecimentos assemelha-se a um "programa disfarçado". Seria interessante que ela não possuísse controle embutido, e sim, um controle coletado na forma de evidência, pois, desta

forma, não afetaria o trabalho do engenho de inferência.

Temos um problema a considerar: mesmo que a base de conhecimentos seja altamente descritiva, o engenho de inferência pode mostrar-se incapaz de responder à pergunta. Seria interessante que o engenho de inferência informasse quando não fosse possível responder à consulta.

Em resumo, temos três considerações desejáveis para um sistema especialista:

(1) A base de conhecimentos não deve incluir controle, nem mesmo "disfarçado", na ordem de sentenças ou de conjunções (como acontece em um programa escrito em PROLOG).

(2) O engenho de inferência deve responder à consulta, provando a veracidade ou falsidade do predicado correspondente.

(3) O engenho de inferência deve dizer que não foi possível responder à consulta quando isto ocorrer.

Se as regras da base de conhecimentos forem suficientemente gerais para se tornarem equivalentes às sentenças do cálculo de predicados de primeira ordem, temos que NÃO É POSSÍVEL realizar as três considerações citadas utilizando-se apenas de métodos formais. Este é um resultado da lógica já conhecido [Shoensield-67]. Vamos, portanto, analisar o quanto regras como

as usadas em sistemas especialistas se aproximam de sentenças do cálculo de predicados de primeira ordem [Clocksin-84].

Nos livros de cálculo de predicados de primeira ordem, os predicados são semelhantes aos usados para montar as regras anteriormente citadas. Estes predicados, ou sentenças atômicas, podem ser combinados de várias maneiras para formar as mais diversas sentenças compostas. Primeiro, pode-se usar conectivos lógicos os quais são maneiras formalmente definidas de se expressar noções familiares, tais como negação, conjunção, alternativas, implicação e equivalência. O significado preciso destes conectivos é dado em [Gries-87]. Usaremos a seguinte sintaxe:

~ P	;	não(P)
P & Q	;	P e Q
P v Q	;	P ou Q
P → Q	;	P implica Q
P ↔ Q	;	P é equivalente a Q

Os quantificadores também servem para formar sentenças complexas, fornecer meios para expressar conjuntos de objetos e dizer o que é verdadeiro sobre eles.

O cálculo de predicados possui dois tipos de quantificadores: o quantificador universal e o existencial. Dadas uma variável v e uma sentença S, podemos formar:

todo(v, S)	;	S é verdadeiro para todo v
existe(v, S)	;	existe um v para o qual S é verdadeiro

Através de uma série de manipulações simbólicas é possível transformar qualquer sentença do cálculo de predicados em uma conjunção de disjunções de literais [Clocksin-84], a qual possui o seguinte formato:

<conjunção_de_disjunção>::= <disjunção> |
<disjunção> & <conjunção_de_disjunção>
<disjunção> ::= <literal> | <literal> v <disjunção>

Quando observamos um conjunto de regras na base de conhecimentos, supomos que todas são verdadeiras ao mesmo tempo (considerando-se uma base de conhecimentos simples, em que são descartadas partições em submundos distintos e influências contextuais diferentes em cada submundo). Se considerarmos a mesma interpretação para "verdadeiro ao mesmo tempo" na lógica e na base de conhecimentos, concluimos que a base de conhecimentos é uma conjunção de regras.

Para termos a base de conhecimentos como uma conjunção de disjunções, vamos comparar cada regra com uma disjunção. Podemos considerar que o "se" da regra seja uma implicação apontada para a esquerda. Desta forma, a regra de avo (r_4) pode ser expressa por: $avo(X,Y) \leftarrow (pai(X,Z) \& pai(Z,Y))$

onde todas as variáveis são universalmente quantificadas.

No cálculo de predicados temos que:

$$(P \leftarrow Q) \text{ é equivalente a } (P \vee \sim Q)$$

Assim, a regra (r_4) "avo(X,Y)" é equivalente a:

$$avo(X,Y) \vee \sim (pai(X,Z) \& pai(Z,Y))$$

Aplicando o teorema de De Morgan, encontramos:

$$avo(X,Y) \vee \sim pai(X,Z) \vee \sim pai(Z,Y)$$

Concluimos então que a regra é uma disjunção de literais. A única restrição para se ter uma regra é que exista ao menos um literal positivo. Esta restrição é, contudo, menor. Podemos concluir, portanto, que regras são equivalentes a sentenças do cálculo de predicados. Logo, as três considerações desejáveis são irrealizáveis.

Se as três considerações são irrealizáveis, somente nos resta abandonar uma delas. Geralmente, abandona-se o fato de não se incluir controle. Decide-se, pois, por incluir controle no processo dedutivo desempenhado pelo engenho de inferência. O controle dirá, a cada passo, que hipótese e regra de inferência utilizar.

Podemos considerar o controle como sendo constituído por um conjunto extra de regras de inferência, as quais tem duas conseqüências desfavoráveis:

(c_1) Substituição de certos conectivos e operações por aproximações. Estas aproximações fazem com que a resposta à consulta, em alguns poucos casos, não seja satisfatória. Porém, na maioria das vezes, a consulta é satisfatória.

(c_2) O sistema torna-se incapaz de utilizar corretamente bases de conhecimentos constituídas por qualquer conjunto de sentenças.

Vejamos a gravidade das duas conseqüências. A conseqüência (c_1) é menos grave, visto que as aproximações são bastante boas na maior parte dos casos práticos. No caso das regras que temos estudado, um exemplo é substituir o não da lógica tradicional pela seguinte aproximação:

> Para provar nao(P) tente primeiro provar P.
> Caso consiga, então nao(P) é falso.
> Caso não consiga, então nao(P) é verdadeiro.

Esta aproximação cria a chamada hipótese do mundo fechado: qualquer "coisa" é verdadeira apenas se está dito explicitamente no modelo do mundo em questão que ela é verdadeira.

A conseqüência (c_2) é mais grave. Na construção da base de conhecimentos deve-se evitar certas sentenças, pois o sistema seria incapaz de tratá-las eficientemente. Suponhamos que, antes da introdução do controle, tenhamos o seguinte esquema de prova:

(1) Seja uma consulta na forma de uma conjunção de predicados positivos.
(2) Faça a hipótese ser a consulta negada e colocada na forma de uma disjunção de literais. Todos os literais serão negativos.
(3) Coloque a base de conhecimentos na forma de uma conjunção de disjunções. Como vimos, isto é sempre possível [Clocksin-84]. Todos os literais de cada disjunção, exceto um, serão negativos.
(4) Escolha um dos literais negativos da hipótese. Seja ~ C este literal.

(5) Escolha uma das disjunções da base de conhecimentos em que apareça o literal positivo C. Esta disjunção é verdadeira pois, caso contrário, toda base de conhecimentos seria falsa. Seja R a disjunção escolhida.

(6) Faça a hipótese ser R v hipótese negada.

(7) Elimine C v ~ C pois esta disjunção é sempre verdadeira.

(8) Se encontrar uma contradição com a eliminação de C v ~ C, a negação da consulta é falsa. Logo, a consulta é verdadeira.

(9) Se a hipótese não ficou vazia, volte ao passo (4).

Para que o algoritmo fique mais claro, vejamos um exemplo concreto. Admita-se a seguinte base de conhecimentos:

```
suc(V, s(V))                    ; sucessor de um número V é s(V).
p(X) se suc(Y,X) & p(Y)         ; X é positivo se X é sucessor de Y e
                                ; Y é positivo
p(s(0))                         ; o sucessor de zero é positivo
```

Suponhamos que a consulta seja verificar se o sucessor do sucessor de zero é positivo, ou seja:

p(s(s(0)) ?

Os passos da prova segundo o algoritmo dado são os seguintes:

(1) p(s(s(0)) ? ; neste caso, a consulta possui apenas um ; predicado positivo

(2) Hipótese é a consulta negada. Prova por absurdo.

~ p(s(s(0))

(3) Colocar a base de conhecimentos na forma de uma conjunção

de disjunções.

[d_1] suc(V, s(V)) &

[d_2] p(X) v ~ suc(Y,X) v ~ p(Y) &

[d_3] p(s(0))

(4.1) Escolher um dos literias negativos da hipótese. Neste caso, temos apenas um.

~ p(s(s(0)))

(5.1) Escolher uma das disjunções da base de conhecimentos em que apareça o literal p(s(s(0))), ou seja, a forma positiva do literal positivo da hipótese. Escolhemos [d_2]. Para que p(X) seja igual a p(s(s(0))), fazemos X = s(s(0)). O sinal "=" aqui expressa a unificação da lógica.

(6.1) Devemos fazer a nova hipótese ser [d_2] v ~ p(s(s(0))), ou seja:

p(s(s(0))) v ~ suc(Y,s(s(0))) v ~ p(Y) v ~ p(s(s(0)))

(7.1) Eliminar p(s(s(0))) v ~ p(s(s(0))) pois é sempre verdadeiro e, portanto, não irá influenciar na veracidade da hipótese. Temos agora, a seguinte hipótese.

~ suc(Y,s(s(0))) v ~ p(Y)

(8.1) Não encontramos uma contradição.

(9.1) Como a hipótese não ficou vazia, devemos voltar para o passo (4).

(4.2) Escolher um dos literais negativos da hipótese. Vamos escolher ~ p(Y) .

(5.2) Escolher uma das disjunções da base de conhecimentos. Vamos escolher [d_3] com Y = s(0).

(6.2) Nova hipótese será: [d3] v ~ suc(s(0),s(s(0))) v ~ p(s(0)), ou seja, p(s(0)) v ~ suc(s(0),s(s(0))) v ~ p(s(0)).

(7.2) Eliminar p(s(0)) v ~ p(s(0)). Temos agora, a seguinte hipótese: ~ suc(s(0), s(s(0))).

(8.2) Encontramos uma contradição ~ suc(s(0),s(s(0))) e [d1] = suc(V,s(V)), onde V = s(0). Logo, a hipótese ~ p(s(s(0))) é falsa.

CONCLUSÃO: A consulta p(s(s(0)) é verdadeira.

Do procedimento de prova, concluimos que os passos (4) e (5) são cruciais para que a prova chegue a um bom termo. Admitamos que se no passo (5.2) do exemplo dado tivéssemos escolhido [d2] ao invés de [d3], e que a partir deste ponto, escolhêssemos [d2] sistematicamente. Se isto ocorresse, a prova ficaria em um laço eterno e nunca chegaria a uma conclusão. Em resumo, o sistema seria incapaz de informar que a prova não existe. Desta forma, a terceira consideração desejável não seria possível de ser satisfeita. Precisamos, neste caso, de um bom algoritmo que sempre faça as melhores escolhas. Acontece que tal algoritmo não existe. Este é um bem conhecido resultado de indecibilidade dos sistemas de lógica de primeira ordem [Shoensield-67].

Se não existe algoritmo para fazer as escolhas dos passos (4) e (5), várias atitudes podem ser tomadas. Uma possibilidade é deixar que o engenheiro de conhecimentos (pessoa que projeta a base de conhecimentos) indique, de alguma forma, a escolha que

deve ser feita. Ele poderia, por exemplo, colocar em primeiro lugar as sentenças que devessem ser escolhidas primeiro, e, nas condições, os predicados mais promissores deveriam anteceder os menos promissores. Uma outra possibilidade seria colocar indicações de prioridade nas regras e nas conjunções. Mais sofisticada ainda seria a criação de predicados especiais que indicariam ao engenho de inferência que regra deveria ser escolhida antes. Quem forneceria estes predicados seria o engenheiro do conhecimentos. Qualquer uma destas soluções tem um problema em comum: "o projetista da base de conhecimentos estaria introduzindo o controle no sistema".

A introdução do controle por qualquer um dos processos discutidos é a única tarefa que exige inteligência. De fato, com exceção dos passos (4) e (5), a dedução pode ser executada mecanicamente. Passar a tarefa de fazer a escolha para o engenheiro de conhecimentos é eliminar a parte "inteligente" do sistema. Porém, não é isto que desejamos em se tratando de um trabalho sobre Inteligência Artificial. Precisamos, portanto, analisar como o engenheiro de conhecimentos introduz o controle no sistema e tentar automatizar o processo.

3.2 APRESENTAÇÃO FORMAL DO CICLO DEDUTIVO DO S.E.

Nesta secção vamos definir de maneira precisa o ciclo dedutivo de

um engenho de inferência como o mostrado informalmente na seção anterior. Para que fiquem claras as dificuldades em se implementar o referido ciclo, apresentaremos o algoritmo em uma notação funcional. Esta é a única secção em que usaremos a notação funcional para especificar algoritmos. Nas demais seções iremos aderir às especificações lógicas.

DEFINIÇÃO 1:

Para cada dois objetos A e B existe um PAR A.B. O PAR é uma estrutura de dados que possui um construtor, dois seletores, um reconhecedor e uma representação, com a seguinte forma:

construtor: É o ponto. Comparando com LISP, X.Y é equivalente a (cons X Y). Em LOGO, X.Y é equivalente a FPUT X Y

seletores: car(X.Y) = X

cdr(X.Y) = Y

reconhecedor: consp(X.Y) = verdadeiro

consp(2) = falso

representação: X.Y, onde o ponto tem associatividade direita

DEFINIÇÃO 2:

LISTA é uma seqüência ordenada de objetos colocados entre parênteses. Consideraremos uma lista (B C D F) como sendo equivalente a B.C.D.F.(), onde () é o identificador da lista vazia.

NOTAÇÃO:

Variáveis lógicas universalmente quantificadas são representadas por dois pontos seguidos de uma seqüência de letras. Exemplos de tais variáveis são :x, :y, :estado. O predicado (varp :x) reconhece tais variáveis. Se o argumento de varp for uma variável, o predicado sucede. Caso contrário, falha. Admita-se que o código do caracter dois pontos seja 58. Neste caso, a definição em LISP de varp é a seguinte:

```
(defun varp(V)
    (eq (char-code V) 58))
```

DEFINIÇÃO 3:

Uma LIGAÇÃO é um par cujo car é uma variável.

DEFINIÇÃO 4:

Um AMBIENTE é um conjunto de ligações, nenhuma das quais possui o mesmo car. Representaremos ambientes por listas cujos elementos são ligações. Dada uma variável :x e um ambiente E, a função (assoc ':x E) fornece a ligação de E cujo car é :x. Se tal ligação não existir, (assoc ':x E) fornecerá a lista vazia ().

DEFINIÇÃO 5:

Admita-se que (assoc ':x E) não é vazio. Neste caso dizemos que :x está LIGADO em E e denotamos (lig :x E). A definição de lig é:

```
(defun lig(X E)
    (assoc X E))
```

DEFINIÇÃO 6:

Se (lig :x E) então VALOR IMEDIATO de :x em E pode ser definido como (cdr (assoc X E)), ou seja, é o valor que está ligado a :x. Em LISP temos a seguinte definição:

```
(defun vi(X E)
    (cdr (assoc X E)))
```

DEFINIÇÃO 7:

VALOR FINAL de um objeto X em um ambiente E é denotado por (vf X E) e tem a seguinte especificação:

```
(vf X E) :  se (lig X E) então (vf  (vi X E) E)
            senão X
```

Dada uma variável X e um ambiente E, valor final deve conseguir seu valor imediato. Se este valor for uma variável, procura-se o valor dela. Continua-se esta procura até conseguir uma variável não ligada ou um objeto que não seja variável. EM LISP, temos a seguinte definição para (vf X E):

```
(defun vf(X E)
    (let ((( S  (assoc X E)))
         (if S (vf (cdr S) E) X) ))
```

Vamos mostrar que esta definição satisfaz a especificação.

Admita-se que E seja realmente um ambiente. Temos que S é (assoc X E). Então (if S (vf (cdr S) E) X) é equivalente a

```
(if (assoc X E)  (vf (cdr (assoc X E)) E) X)
```

O programa em LISP é, portanto, equivalente a

```
(defun vf(X E)
    (if (assoc X E) (vf (cdr(assoc X E)) E) X) )
```

Pela definiçao 5 de (lig X E) temos que:

```
(defun vf(X E)
    (if (lig X  E) (vf (cdr (assoc X E)) E) X) )
```

Finalmente, temos pela definição 6 de (vi X E):

```
(defun vf(X E)
    (if (lig X E) (vf (vi X E) E) X) )
```

Concluimos que a especificação é equivalente ao programa em LISP antes das transformações. O programa, entretanto, é mais eficiente. Temos que (vi X E) somente será executado quando (lig X E), como era de se esperar.

DEFINIÇÃO 8:

DEREFERENCIAMENTO de X no ambiente E é basicamente a substituição de todas as variáveis ligadas de X por seus valores finais em E. o dereferenciamento é denotado por (deref X E) e especificado como:

```
(deref X E) == se (consp X)
               então (deref (car X) E).(deref (cdr X) E)
               senão se (lig X E)
                     então (deref (vf X E) E)
                     senão X
```

Na definição 7, partimos de um programa eficiente (em termos) e mostramos que ele era equivalente a especificação. Agora vamos proceder na direção contrária. Vamos partir de um espelho da especificação e chegar a um programa mais eficiente. O espelho da especificação é:

```
(defun deref(X E)
    (if (consp X)
        (cons (deref (car X) E) (deref (cdr X) E))
        (if (lig X E)
            (deref (vf X E) E)
            X) ) )
```

Se (lig X E), temos pela definição 7 de valor final (vf X E) que (vf X E) é equivalente a (vf (vi X E) E). Temos então que:

```
(defun deref(X E)
    (if (consp X)
        (cons (deref (car X) E) (deref (cdr X) E))
        (if (lig X E)
            (deref (vf (vi X E) E) E)
            X) ) )
```

Substituindo-se (lig X E) e vi(X E) por suas definições:

```
(defun deref(X E)
    (if (consp X)
        (cons (deref (car X) E) (deref (cdr X) E))
        (if (assoc X E)
            (deref (vf (cdr (assoc X E)) E) E) )))
```

Se fatorarmos (assoc X E), temos a seguinte definição:

```
(defun deref(X E)
    (if (consp X)
        (cons (deref (car X) E) (deref (cdr X) E) )
        (let (( S  (assoc X E)))
             (if S  (deref (vf (cdr S) E) E) X) )))
```

DEFINIÇÃO 9:

UNIFICAÇÃO é qualquer maneira de se comparar dois objetos onde as variáveis podem receber valores convenientes para garantir a igualdade. Em qualquer sistema especialista que trabalhe com formas de representação de conhecimento baseada em predicados, qualquer forma de unificação é indispensável. A maneira mais elementar de unificar X com Y em E é conseguir uma extensão E' mais geral de E, tal que:

(deref X E) = (deref Y E').

A especificação funcional da unificação é a seguinte:

```
(unif X Y E) == se E é  'nunifica
                então   'nunifica
                senão seja A = (vf X E)  B = (vf Y E)
                se A é B então E senão
                se varp A  então (A.B).E senão
                se varp B  então (B.A).E senão
                se átomo A ou átomo B  então 'nunifica senão
                (unif (car  A) (car B)
                      (unif (cdr A) (cdr  B) E) )
```

O espelho desta especificação funcional em LISP é a seguinte:

```
(defun unif(X Y E)
    (if (eq E 'nunifica) 'nunifica)
        (let ((A (vf X E))
              (B (vf Y E)) )
             (cond ((eq A B) E)
                   ((varp A) (cons (cons A B) E))
                   ((varp B) (cons (cons B A) E))
                   ((or (atom A) (atom B)) 'nunifica)
                   ((unif (car A) (car B)
                          (unif (cdr A) (cdr B) E)))))))
```

DEFINIÇÃO 10:

HIPÓTESE é uma conjunção de literais. Em geral, usaremos a letra

Q para denotar hipóteses (Q de "question"). A hipótese é diretamente derivada da consulta. Na maior parte dos casos a hipótese é a expressão lógica da consulta.

DEFINIÇÃO 11:

Dado um ambiente E e uma hipótese Q, RENOMEAÇÃO é uma regra "C se H" que não contém nenhuma variável em comum com Q ou ligada em E. Dado um conjunto de regras, não é difícil preparar um programa em LISP que forneça renomeações. Seja, por exemplo, os seguintes fatos:

```
pai(1,2).
pai(2,3).
pai(3,4).
```

Como não há nenhuma variável nestes fatos, eles são renomeações de quaisquer Q e E. O programa LISP que fornece as renomeações poderia ser o seguinte:

```
(defun pai(x y)
   '( ((pai 1 2))
      ((pai 2 3))
      ((pai 3 4)) ))
```

Vejamos um caso mais complicado. Sejam as duas regras abaixo relacionadas.

```
avo(:x, :y)  se  pai(:x, :z)  &  pai(:z, :y).
avo(:x, :y)  se  pai(:x, :z)  &  mae(:z, :y).
```

Por algum tipo de compilador, estas regras poderiam ser transformadas no seguinte programa em LISP:

```
(defun avo(x y)
    (list  (let ((:x (nvar)) (:y (nvar)) (:z (nvar)))
                (list (list 'avo :x :y)
                      (list 'pai :x :z)
                      (list 'pai :z :y) ))
           (let ((:x (nvar)) (:y (nvar)) (:z (nvar)) )
                (list (list 'avo :x :y)
                      (list 'pai :x :z)
                      (list 'mae :z :y))) ))
```

onde (defun nvar() (gensym ':)). Observe que cada regra de avô sairá, após renomeada, no seguinte formato:

((avô :x :y) (pai :x :z) (pai :z :y)).

Este formato é mais fácil de manipular.

A definição para mãe poderia ser:

```
(defun mae(x y)
   '( ((mae 4 5)) ))
```

Em geral, representaremos uma regra "C se H" por C.H. Além disto, transformaremos predicados na forma p(x,y,z) para a forma (p x y z), a qual é conhecida como notação funcional de Church.

DEFINIÇÃO 12:

Suponhamos que todos os predicados estejam na notação funcional de Church. Dado o predicado P, o ambiente E e uma base de conhecimentos BC, RELEVANTES de P em E é denotado por (rels P E) e é definido como uma lista de renomeações C.H tal que (unif P C E) não é nunifica. A definição não exige que (rels P E) contenha renomeaçoẽs de todas as regras de BC que unificam com P em E. Além disto, as renomeações de (rels P E) não precisam estar na mesma ordem que as regras estavam na BC. Note que a função (avo x

y) que definimos em LISP fornece as relevantes de predicados na forma (avo :x :y). O mesmo acontece com (pai :x :y) e (mae :x :y).

DEFINIÇÃO 13:

Note que para facilitar a manipulação, passamos a representar regras por (P Q R S ...) em vez de P se Q & R & S & .... Assim, a regra de avo(:x, :y) se pai(:x, :z) & pai(:z, :y) se transformou em ((avo :x :y) (pai :x :z) (pai :z :y)). De modo análogo, conjunções são representadas por listas de literais. Seja uma conjunção X na forma de lista e um ambiente E. Define-se ESCOLHA (escolha X E) como sendo (L A.R) tal que X = (append L A.R), onde append é a função que concatena listas.

DEFINIÇÃO 14:

Seja uma conjunção Q e um ambiente E. Denotamos RESOLUÇÃO por (res Q E L A R C H Unif) e definimos da seguinte maneira:

```
(res Q E L A R C H Unif) ==
                se  (L A.R) = (escolha Q E) e
                    C.H pertencente a (rels Q E) e
                    Unif é (unif A C E)
                então Unif.(deref (append L H R) Unif)
```

As definições que se seguem calculam as resoluções de todas as relevantes de P em E.

```
(defun resrel(P E)
    (resolvedores (crels P E) (escolha P E) E () ))
```

```
(defun resolvedores(Rls LAR E Rs)
    (let ( (L (first LAR))
           (A (first (second LAR)))
           (R (cdr (second LAR))) )
         (dolist (Rn Rls)
           (when (listp (unif A (first Rn) E))
                        (push (res Q E L A R C H
                                   (unif A (first Rn) E)) Rs)))
         Rs))


(defun res(Q E L A R C H Unif)
    (if (and (equal (list L (cons A R)) (escolha Q E))
             (member (cons C H) (rels Q E))
             (equal Unif (unif A C E)))
        (cons Unif (deref (append L H R) Unif)) ))

(defun rels (P)    (eval (car P)) )

(defun escolha(P)   (list () P) )
```

Vamos fazer as simplificações. A função res que acabamos de fornecer é um espelho da especificação. Ela é usada apenas dentro de resolvedores. (rels P E) são renomeações candidatas a relevantes. Temos que res somente é chamada depois de se verificar que (listp (unif A (first Rn) E)). Rn é uma sentença e, portanto, tem a forma C.H. Dizer que (unif A C E) é uma lista significa que a unificação teve êxito pois, quando ela fracassa, o resultado é o símbolo 'nunifica. Se esta unificação teve êxito, a sentença C.H passada para res é relevante. Não é pois necessário verificar se (member (cons C H) (rels Q E)). Além disso, Unif é igual a (unif A C E) e, desta forma, não é necessário verificar se (equal Unif (unif A C E)). Finalmente, LAR é (escolha P E). Logo (list L (cons A R)) é igual a (escolha Q E). Assim sendo, podemos remover o

condicional da definição de res. Teremos a seguinte definição:

```
(defun res(Q E L A R C H Unif)
    (cons Unif (deref (append L H R) Unif)))
```

Como os argumentos Q, E, A e C não são usados podem, portanto, ser removidos. Teremos:

```
(defun res(L R H Unif)
    (cons Unif  (deref (append L H R) Unif)))
```

Na definição de resolvedores, podemos fatorar (unif A (first Rn) E). Além disto, devemos remover os argumentos não utilizados de res. Teremos:

```
(defun resolvedores (Rls LAR E Rs)
    (let ( (L (first LAR))
           (A (first (second LAR)))
           (R (cdr   (second LAR))) )
         (dolist (Rn Rls)
            (let ((Un (unif A (first Rn) E)))
                 (when (listp Un)
                       (push (res L R (cdr Rn) Un) Rs)) ))
         Rs))
```

Finalmente podemos preparar o engenho de inferência propriamente dito, da seguinte forma:

```
(defun prova(SOLVED WAITING)
    (loop
         ((null  WAITING) SOLVED)
         (let ((Y (pop WAITING)))
              (dolist (R (resrel (cdr Y) (first Y)))
                      (if (cdr R)
                          (push R WAITING)
                          (push (car R) SOLVED))))))
```

```
(defmacro consulta(Padrao)
    '(mapcar (quote (lambda (E) (deref (quote, Padrao) E)))
             (prova ()  (list (cons () (quote (, Padrao )))))))
```

Observe que as diversas respostas são colocadas na lista SOLVED. Poderíamos opcionalmente ter tentado comparar estas respostas, escolhendo a melhor. Neste caso, precisaríamos de uma métrica, que poderia ser uma medida de crença ou coeficiente de certeza. Este ponto, entretanto, não é importante para o problema que desejamos estudar. O ponto IMPORTANTE é que, em uma certo momento do programa, precisamos realizar uma (escolha P E). Esta escolha corresponde ao passo (4) da prova informal (seçao 3.1).

Vejamos um exemplo. Se desejarmos saber sobre os avôs, devemos fazer a seguinte consulta:

```
(consulta (avo :x :y) )
```

teremos a seguinte resposta:

```
( (avo 3 5)  (avo 2 4)  (avo 1 3) ).
```

Vejamos a sua execução:

Inicialmente temos, SOLVED = () e

```
WAITING = ( () (avo :x :y) )
```

Há duas sentenças relevantes de avô:

```
( ((avo :x :y) (pai :x :z) (pai :z :x))
  ((avo :x :y) (pai :x :z) (mae :z :y)) )
```

Para a primeira sentença:

```
( (avo :x :y) (pai :x  :z)  (pai :z :y) )
```

encontramos: 1) (pai 1 2) e

(pai 2 3) então

(avo 1 3) será guardado em SOLVED

2) (pai 2 3) e

(pai 1 3) então

(avo 2 4) será guardado em SOLVED

Para a segunda sentença:

((avo :x :y) (pai :x :z) (mae :z :y))

encontramos: 1) (pai 3 4) e

(mae 4 5) então

(avo 3 5) será guardado em SOLVED

portanto, SOLVED = ((avo 3 5) (avo 2 4) (avo 3 5)).

3.3 TÉCNICA USADA NA IMPLEMENTAÇÃO DO ALGORITMO

Antes de apresentarmos com detalhes os processos da figura 3.2, iremos primeiro descrever a técnica que usamos para programar. Temos, na moderna Engenharia de Software, uma tendência de construir programas que se possam provar corretos. Embora discuta-se a confiabilidade destas provas, não há dúvidas de que elas diminuem bastante a quantidade de erros de lógica em algoritmos complexos.

Para provar formalmente que programas estão corretos costumam-se usar a notação ⟨P⟩ S ⟨Q⟩ para indicar que a execução de um programa S começa em um estado satisfazendo ao predicado P e termina em outro estado satisfazendo ao predicado Q. Nesta notação, P é chamado pré-condição e Q é a pós-condição (capítulo 7 de [Gries-87]), onde o predicado wp(S,Q) é definido como verdadeiro em qualquer estado em que, a partir da execução de S, leva fatalmente a um estado no qual Q é verdadeiro. Neste caso, ⟨P⟩ S ⟨Q⟩ é equivalente a P ⇒ wp(S, Q).

Assim sendo, ⟨P⟩ S ⟨Q⟩ é uma sentença do cálculo de predicados. Mais especificamente, é a sentença:

wp(S,Q) é verdadeiro se P é verdadeiro.

Suponhamos que P seja um predicado que reconheça se os dados de entrada do programa S estão corretos, ou seja, satisfaz o usuário e Q seja um predicado que é verdadeiro quando o estado final satisfaz ao conjunto das respostas corretas, ou seja, respostas que satisfazem ao usuário. O par P e Q é a especificação do programa, o qual indica o que é um dado correto e o que é uma resposta correta, respectivamente. Admitiremos que a especificação é mais simples de se construir do que o próprio programa pois, em geral, é mais fácil dizer o que se tem e o que se deseja obter do que determinar como conseguir o que se quer.

Admitamos que o par P e Q seja a especificação. Neste caso, se ⟨P⟩ S ⟨Q⟩ é verdadeiro então S é um programa correto, ou seja,

satisfaz a especificação. De fato, se fornecido a S um dado correto (que satisfaz P) obtém-se uma resposta correta (que satisfaz Q).

Existe dois tipos de correção: total e parcial. Provamos que um programa está totalmente correto se provarmos que ele, além de satisfazer a especificação, pára. Provamos que um programa está parcialmente correto se provarmos que satisfaz a especificação sem provar que pára. Resumindo:

correção total = correção parcial + garantia de parada.

No caso de correção total, o predicado que é verdadeiro é denotado por ⟨P⟩ S ⟨Q⟩. Na correção parcial, o predicado verdadeiro é P {S} Q.

Ao longo desta tese, usaremos um modo menos usual de se conseguir programas corretos. Este modo funciona para linguagens baseadas na lógica. A idéia é bastante intuitiva e é possível certamente que já tenha ocorrido a várias pessoas, muito embora não tenhamos nenhuma referência bibliográfica para citar.

O método que usaremos é o seguinte: Tentar encontrar uma classe de programas bastante ampla para resolver qualquer um dos problemas nos quais estamos interessados. Em seguida, provaremos que qualquer programa desta classe está correto. Desta forma, basta termos cuidado de construir apenas programas pertencentes a tal classe. Fazendo assim, podemos cometer apenas dois tipos de erros que não podem ser detectados pelo compilador:

1) Erros de especificação.

2) Erros em se determinar se um programa está na classe que contenha apenas programas corretos.

Iremos nos preocupar apenas com correção parcial. Os programas serão escritos em PROLOG, e nossas especificações deverão satisfazer às seguintes restrições:

1) P é uma disjunção de conjunções. P = P1 v P2 v P3 v ....
2) Seja P = P1 v P2 v P3 v .... É necessário que Pi ∧ Pj seja falso para todo par i,j .
3) Q é uma disjunção de conjunções. Q = Q1 v Q2 v Q3 v ....

Temos que toda sentença do cálculo de predicados pode ser colocada na forma de uma disjunção de conjunções, ou seja, na forma normal disjuntiva. Logo, não há maiores comentários para (1) e (3). Vamos mostrar que a restrição (2) pode também ser contornada sem maiores problemas.

Admita-se que a pré-condição tenha a forma p1 v p2 v p3 v ... e que p1 e p2 não satisfaçam a restrição (2), ou seja, suponhamos que p1 ∧ p2 é verdadeiro.

Temos que ~ p2 v p2 é sempre verdadeiro; então, a pré-condição pode ser transformada em:

(a) (p1 ∧ (~ p2 v p2)) v p2 v p3 v ...

Temos que (p1 ∧ (~ p2 v p2)) é equivalente a p1 ∧ ~ p2 v p1 ∧ p2;

então, (a) pode-ser transformada em:

(b) $p1 \wedge \sim p2 \vee p1 \wedge p2 \vee p2 \vee p3 \vee \ldots$

$p2 \wedge$ verdadeiro é equivalente a p2. Esta expressão pode ser transformada em:

(c) $p1 \wedge \sim p2 \vee p1 \wedge p2 \vee p2 \wedge$ verdadeiro $\vee p3 \vee \ldots$

Temos que $p1 \wedge p2 \vee p2 \wedge$ verdadeiro é equivalente a $p2 \wedge (p1 \vee$ verdadeiro); então (c) pode ser transformado em:

(d) $p1 \wedge \sim p2 \vee p2 \wedge (p1 \vee$ verdadeiro$) \vee p3 \vee \ldots$

$(p1 \vee$ verdadeiro) é sempre verdadeiro. Então:

(e) $(p1 \wedge \sim p2) \vee (p2 \wedge$ verdadeiro$) \vee p3 \vee \ldots$

$(p2 \wedge$ verdadeiro) é p2

Finalmente, $(p1 \wedge \sim p2) \vee p2 \vee p3 \vee \ldots$

Esta pré-condição transformada satisfaz o pré-requisito (2). De fato, $(p1 \wedge \sim p2) \wedge p2 \equiv p\bot \wedge (\sim p2 \wedge p2) \equiv p1 \wedge$ falso $\equiv$ falso

Suponhamos, portanto, que tenhamos transformado as pré-condições e pós-condições de modo a satisfazer os requisitos. Seja X o dado de entrada do programa e Y, a saída. Seja um programa em PROLOG com a seguinte estrutura:

```
r(X,Y) :- P1(X), R1(X,Y), Q1(Y).
r(X,Y) :- P2(X), R2(X,Y), Q2(Y).
```

Seja r(X1,Y1) a solução que satisfaz à primeira sentença. Negando r(X1,Y1) temos, pelo princípio da resolução:

$$\sim r(X1,Y1) \vee r(X1,Y1) \vee \sim (P1(X1) \wedge R1(X1,Y1) \wedge Q1(Y1))$$

Isto é equivalente a:

$$\sim (P1(X1) \wedge R1(X1,Y1) \wedge Q1(Y1)) = \text{falso}$$

pois partimos do princípio de que r(X1,Y1) é solução. Conclui-se, então, que:

$$P1(X1) \wedge R1(X1,Y1) \wedge Q1(Y1) = \text{verdadeiro}$$

Pelo requisito (2), a que os predicados Pi devem satisfazer, concluimos que P2(X1) é falso. Podemos, então, escrever:

$$(P1(X1) \vee P2(X1)) \wedge R1(X1,Y1) \wedge (Q1(Y1) \vee Q2(Y1))$$

O mesmo raciocínio que usamos para duas regras pode ser extendido. Temos, então, que as soluções obedecerão, em geral, as equações do tipo:

$$(P1(X) \vee P2(X) \vee P3(X) \vee \ldots) \wedge R(X,Y) \wedge (Q1(Y) \vee Q2(Y) \vee \ldots)$$

A solução obedece desta forma às especificações e está correta.

Agrupamos todos os argumentos de entrada na variável X e todos os argumentos de saída em Y. Isto, entretanto, não afeta a prova. O resultado, portanto, pode ser usado no caso de haver mais de dois argumentos. Além disso, R(X,Y) pode ser uma conjunção de literais. Isto significa que o resultado é válido para regras do tipo:

```
r(X,Y) :- P1(X), Ra(X,Y), Rb(X,Y), Rc(X,Y), ..., Q1(Y).
```

Resumindo, iremos adotar o seguinte estilo de programação: prepararemos especificações P = P1 v P2 v ... e Q = Q1 v Q2 v ... de tal maneira que Pi $\wedge$ Pj seja falso para qualquer par i,j. Em seguida escreveremos o programa usando regras na forma ri :- Pi, S, Qi. Se tal programa produzir resposta, ela estará correta. O programa, entretanto, pode não ser eficiente. O último passo,

que não será mostrado neste trabalho, é torná-lo eficiente através de transformações cuidadosas.

3.4 ALGORITMO PROPOSTO

Não existe um algoritmo geral que permita introduzir o controle no sistema especialista. Vimos, na seção 3.1, o problema correspondente à (escolha Q E), ou seja, qual predicado escolher para se fazer a resolução. Vamos mostrar nesta seção o algoritmo de fazer a escolha.

Temos que, conhecendo-se os objetos da área de atuação, é possível saber em que estados eles devem estar quando certos predicados, que denominaremos pontos de inspeção, forem processados. Fazendo uma analogia, podemos considerar as condições de uma regra, semelhantes a um quebra-cabeças desmontado e embaralhado. Os predicados seriam as peças e os argumentos seriam os encaixes. Há um complicador; os encaixes não são totalmente conhecidos, podem estar substituídos por variáveis universais. Entretanto, sabemos de algumas informações sobre os encaixes, como por exemplo, quais são "machos" ou "fêmeas" e, talvez, isto seja suficiente para montar todo o quebra-cabeças, ou seja, colocar os predicados na ordem em que eles devem ser escolhidos, de modo que (escolha Q E) possa reduzir-se a (list () Q). Veremos portanto, nesta seção, o algoritmo utilizado para ordenar as condições dos predicados das

regras.

3.4.1 EXEMPLO DA APLICAÇÃO DO ALGORITMO

O algoritmo implementado manipula bases de conhecimentos contendo regras nas quais os argumentos dos predicados sejam símbolos (seqüências de letras minúsculas), inteiros, reais aproximados (pontos flutuantes), expressões aritméticas e variáveis, obedecendo a seguinte sintaxe:

<lista_de_argumentos> ::= <argumento> |
<argumento>,<lista_de_argumentos>

<argumento> ::= <funtor> | <variaveis> | <expressão> | <inteiros>

<funtor> ::= <minúscula> | <minúscula> <minúsculas>

<minúscula> ::= a | b | c | d | e | f | g | ...

<minúsculas> ::= <minúscula> | <minúscula> <minúsculas> |
_<minúsculas>

<variáveis> ::= <maiúscula> | <maiúscula> <maiusculas>

<maiúsculas> ::= <maiúscula> | <minúscula> <maiúsculas> |
<maiúscula> <maiúsculas> | <inteiros>

<inteiros> ::= 0 |1 | 2 | 3 | 4 | 5|...

<reais> ::= <inteiros> | <inteiros>.<inteiros>

<expressões> ::= <expressão> + <expressão> |
<expressão> - <expressão> |
<expressão> * <expressão> |
<expressão> / <expressão> |

<expressão> ::= <inteiros> | <reais> | <variáveis> | <expressão>

O exemplo que será usado nesta secção para mostrar o algoritmo é o seguinte:

```
regra(fat,1,fat(0,1)).
regra(fat,2,fat(N,N*F1) se fat(N-1,F1))
```

A base de conhecimentos está expressa em uma forma conveniente para PROLOG manipular.

3.4.2 DESCRIÇÃO DO ALGORITMO USANDO D.F.D.

O algoritmo para introduzir o controle correspondente à (escolha Q E), baseia-se em rearranjar as condições de cada regra de modo que os literais estejam em ordem de prioridade para resolução. Após esta ordenação, o sistema deve sempre pegar o literal mais a esquerda para fazer a resolução . O critério que usaremos para obtermos a prioridade dos literais é o seguinte: em princípio, os literais devem estar em ordem tal que sejam conhecidos os valores das variáveis a medida em que eles se tornarem indispensáveis. Estes valores tornam-se indispensáveis quando, por exemplo, precisamos realizar um cálculo com eles. A figura 3.2 apresenta o DFD (DIAGRAMA DE FLUXO DE DADOS) do sistema que fará a ordenação.

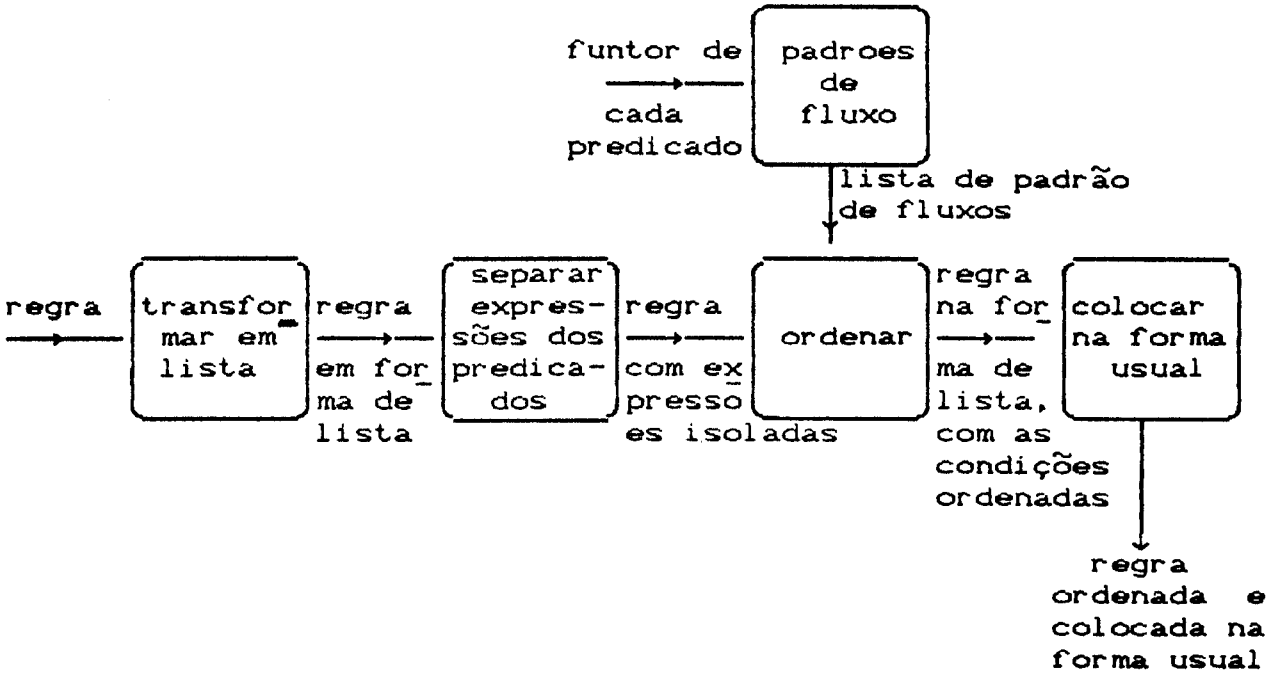


Figura 3.1 DFD do sistema de ordenação

Temos que dada as regras da base de conhecimentos da seção 3.4.1, após a aplicação do algoritmo obtemos as seguintes regras:

```
regra(fat,1,fat(0,1)).
regra(fat,2,fat(N,X) se Y vale N-1 & fat(Y,F1) & X vale N*F1).
```

Observe que a segunda regra de fat ficou com as condições rearranjadas em ordem (esquerda para direita) de prioridade para resolução.

Vamos analisar cada um dos processos da figura 3.1, considerando como exemplo a segunda regra de fat, a qual indica como calcular o fatorial de um inteiro maior do que zero.

regra(fat,2,fat(N,N*F1) se fat(N-1,F1))

1º passo: "transformar em lista"

Este processo colocará a regra na forma de uma lista de literais para facilitar a sua manipulação. Após sua execução a regra estará na forma de lista:

[fat(N,N*F1), fat(N-1,F1)]

2º passo: "separar expressoes dos predicados"

Este processo separa as expressões aritméticas que estão nos argumentos dos predicados. Para isto substitui-se, no predicado, a expressão por uma variável qualquer V e iguala-se esta variável com o resultado da expressão. Obteremos, então, uma regra com expressões isoladas. Após sua execução a regra estará no seguinte formato:

[fat(N,X), X vale N*F1, fat(N1,F1), N1 vale N-1]

No processo de prova, "X vale N*F1" unificará o valor de N*F1 com X. Para fazer esta unificação, o sistema deve ter condições de calcular este valor, ou seja, quando N e F1 forem conhecidos. Os literais devem ser resolvidos em uma ordem tal que, quando se chegar a X vale N*F1, tanto N como F1 sejam conhecidos. Este pode ser um critério de ordenação, ou melhor, um critério para

(escolha Q E). Entretanto, não é apenas este predicado "X vale Expressão" que exige o valor conhecido de certas variáveis. Temos também que argumentos de entrada dos predicados devem ser conhecidos. Devemos, portanto, determinar todos os predicados que exigem valores de variáveis conhecidos.

3º passo: "padroes de fluxo"

O próximo passo é tentar recuperar a maior parte possível dos encaixes do quebra-cabeças. Para isto faremos o seguinte: cada argumento dos predicados da cabeça de uma regra será classificado como entrada obrigatória ou entrada_e_saída. Quando o predicado for chamado para unificação, os seus argumentos de entrada obrigatória devem ser constantes. Já os de entrada_e_saída podem tanto ser uma constante quanto uma variável. Denotaremos os argumentos de entrada obrigatória por '+' enquanto os outros serão denotados por '-'. Comparando com os encaixes, os argumentos do tipo '+' serão análogos aos encaixes machos. Os do tipo '-', por sua vez, correspondem aos encaixes fêmeas. A classificação dos argumentos em machos e fêmeas recebe o nome de padrão de fluxo.

Este processo obtém a lista Padrão de Fluxo de cada definição (conjunto de regras que definem um predicado). Após sua execução, obteremos o seguinte padrão de fluxo:

['+','-']

4º passo: "ordenar"

Conhecidos os padrões de fluxo de todos os predicados torna-se possível ordenar a regra. Este processo de ordenação utiliza a avaliação parcial. Após a sua execução, a regra estará na seguinte ordem:

[fat(N,X), N1 vale N-1, fat(N1,F1), X vale N*F1]

5º passo: "colocar na forma usual"

Finalmente devemos colocar a regra no formato usual. Após sua execução, a regra estará na seguinte forma usual:

fat(N,X) se N1 vale N-1 & fat(N1,F1) & X vale N*F1

3.4.3 IMPLEMENTAÇÃO DO ALGORITMO

Antes de apresentarmos os predicados correspondentes aos processos da figura 3.1, vejamos o significado dos seguintes símbolos que serão utilizados:

Símbolos	Significado
X = Y	; sucede se X e Y se unificam
X \= Y	; sucede se X e Y não se unificam
\+ X	; sucede se X falha
nonvar(X)	; sucede se X não for variável

```
atomic(X)              ; sucede se X é um tipo de dado atômico
%                      ; comentário
```

Iremos mostrar que nossos algoritmos estão corretos. Uma expressão P {D} Q, onde P são as pré-condições e Q são as pós-condições, é chamada especificação parcialmente correta se a execução de D falha quando P é falso e, quando a execução de D termina, Q é verdadeiro. A especificação é totalmente correta se D sempre pára.

Para facilitar as provas, estamos utilizando definições D puramente lógicas. Isto faz com que os algoritmos se tornem grosseiramente ineficientes. Uma vez que tenhamos certeza de que eles estão corretos, podemos aumentar-lhes a eficiência, introduzindo controle.

Vejamos a definição do predicado correspondente ao processo de transformar_em_lista.

Exemplo:

```
?- transformar_em_lista( fat(N,N*F1) se fat(N-1,F1), R1).
   R1 = [fat(N,N*F1), fat(N-1,F1)]

% transformar_em_lista(R,R1)
% especificação
%  1- pré:   R  = M se Y1 & Y2 & ...
%     pós:   R1 = [M,Y1,Y2,...]
%  2- pré:   R  \= M se Y
%     pós:   R1 = [R]


transformar_em_lista(R, R1) :-
   R = (M se Cd),          % pré_condição
   listar_cauda(Cd, C),
   R1 = [M|C].             % pós_condição
```

```
transformar_em_lista(R, R1) :-
   R \= (X se Y),      % pré_condição
   R \= (V & W),       % pré_condição
   R1 = [R].           % pós_condição
```

O predicado listar_cauda transformará a cauda da regra em lista. Vejamos sua definição:

Exemplo:

```
?- listar_cauda( fat(N-1,F1), C)
   C = [fat(N-1,F1)]

% listar_cauda(Cd, C)
% especificação: se Cd = N1 & N2 & ... então C = [N1,N2, ...]
%  1- pré: Cd = X & Y
%     pós: C = [X|RC], RC é a lista correspondente a Y
%  2- pré: Cd = X,  X \= V & W,
%     pós: C = [X]

listar_cauda(Cd, C) :-
   Cd \=  (X & Y),        % pré_condição
   Cd \=  (V se W),       % pré_condição
   C   =  [Cd].           % pós_condição
listar_cauda(Cd, C) :-
   Cd = (X & Y),          % pré_condição
   listar_cauda(Y, Y1),
   C = [X|Y1].            % pós_condição
```

Vamos demonstrar rapidamente que o algoritmo transformar_em_lista está correto e faz o que pretendemos. Se as especificações forem satisfeitas, o algoritmo faz o que pretendemos. Vamos, portanto, verificar se elas estão satisfeitas. Analisando a listagem de transformar_em_lista, temos que ela nada mais é do que a verificação das pré-condições e das pós-condições. Todos os predicados das pré e pós_condições, exceto "listar_cauda(Cd,C)", são primitivas do PROLOG e dispensam maiores comentários. Temos que mostrar que este predicado está correto, ou seja, que ele garante que C é a lista correspondente a Cd. Isto significa que,

se Cd = N1 & N2 & N3 .... então C = [N1,N2,N3,...]. Vamos provar por indução.

No caso de Cd ser constituído por um único predicado, ele estará correto pois a especificação 2 é forçosamente satisfeita pela primeira sentença da definição. Admitamos que Cd = N1 & N2, e que o algoritmo está correto para N2. Neste caso, a primeira especificação é satisfeita.

Antes de analisarmos o algoritmo de isolar expressoes devemos, em primeiro lugar, reconhecer uma expressão. Isto pode ser feito pelas seguintes regras:

```
expressao(E) :- nonvar(E), E = (X+Y),
                expressao_simples(X), expressao_simples(Y).

expressao(E) :- nonvar(E), E = (X-Y),
                expressao_simples(X), expressao_simples(Y).

expressao(E) :- nonvar(E), E = (X*Y),
                expressao_simples(X),expressao_simples(Y).

expressao(E) :- nonvar(E), E = X/Y,
                expressao_simples(X),expressao_simples(Y).


expressao_simples(X) :- var(X).

expressao_simples(X) :- atom(X).

expressao_simples(X) :- number(X).

expressao_simples(X) :- nonvar(X), \+ atomic(X),  expressao(X).
```

Vejamos o algoritmo isolar expressoes, que irá substituir cada expressão Ei de um predicado por uma variável Vi. Além disto, ele deve construir uma lista contendo todas as estruturas da forma

"Vi vale Ei".

Exemplo:

```
?- isolar_expressoes( fat(N,N*F1), PSE, Exs).
        PSE = fat(N,X),
        Exs = X vale N*F1
```

```
% isolar_expressoes(P, PSE, Exs)
% especificação
%
% pré :  P é uma estrutura na forma Fn(A1,A2,...),ou seja,
%        P =.. [Fn, A1, A2, ...]
%        (não_expressão(Ai) ou expressão(Ai))
%
% pós :  PSE =.. [Fn, V1, V2, ...]
%       Exs = [At1, At2,...],Ati = Vj vale Ej, expressao(Ej)
```

```
isolar_expressoes(P, PSE, Exs) :-
     P =.. [Fn|Args],
     isola_expressoes(Args, As, Exs),
     PSE =.. [Fn|As].
```

isolar expressoes estará parcialmente correto dependendo da correção de isola expressoes. Vejamos o algoritmo de isola expressoes.

Exemplo:

```
?- isola_expressoes([N, N*F1], As, Exs).
   As = [N,X],
   Exs = [ X vale N*F1]
```

```
% isola_expressoes(Args, As, Exs)
% especificação
%
% 1- pré : Args = []
%    pós : As = [], Exs = []
%
% 2- pré : Args = [Arg|R], expressão(Arg)
%    pós : As = [X|RAs], Exs = [X vale Arg|RExs],
%          (R, RAs, RExs) satisfazem a especificação
%
% 3- pré : Args = [Arg|R], não_expressão(Arg)
%    pós : As = [Arg|RAs],
%          (R, RAs, Exs) satisfazem a especificação
```

```
isola_expressoes(Args, As, Exs) :-
    Args = [],              % pré_condição
    As = [],                % pós_condição
    Exs = [].               % pós_condição

isola_expressoes(Args, As, Exs) :-
    Args = [Arg|R],                     % pré_condição
    expressao(Arg),                     % pré_condição
    As = [X|RAs],                       % pós_condição
    Exs = [X vale Arg|RExs],            % pós_condição
    isola_expressoes(R, RAs, RExs).     % pós_condição

isola_expressoes(Args, As, Exs) :-
    Args = [Arg|R],                     % pré_condição
    nao_expressao(Arg),                 % pré_condição
    As = [Arg|RAs],                     % pós_condição
    isola_expressoes(R, RAs, Exs).      % pós_condição
```

Vamos provar que o algoritmo está correto usando indução. No caso do predicado P não ter argumentos, ou quando todos os argumentos já foram analisados (Args = []), ele está correto, pois a especificação 1 é forçosamente satisfeita pela primeira sentença da definição. Admitamos que o predicado contenha um argumento, Args = [Arg]. Se o argumento for uma expressão temos que o algoritmo está correto, pois a especificação 2 é satisfeita pela segunda sentença da definição. Se o argumento não for uma expressão, temos que o algoritmo está correto, pois a especificação 3 é satisfeita pela terceira sentença da definição. Admitamos que Args = [Arg|R] e que o algoritmo está correto para R. Neste caso, as especificações 2 e 3 estão corretas.

O predicado separar expressoes dos predicados irá, como o próprio nome diz, separar expressões dos predicados.

Exemplo:

```
?- separar_expressoes_dos_predicados( [fat(N,N*F1),fat(N-1,F1)],
                                      PredEIs)
   PredEIs = [fat(N,X), X vale N*F1, fat(N1,F1),N1 vale N-1]

% separar_expressoes_dos_predicados(Pred, PredEIs)
% especificação
%
%  1- pré : Pred = []
%     pós : PredEIs = []
%
%  2- pré : Pred = [P|R], P = Fn(A1, A2, Ai,...)
%           (não_expressão(Ai) ou expressão(Ai ))
%     pós : PredEIs = [PSE, Ati,...|RSExs],
%           PSE = Fn(V1, Vj, ... ), não_expressão(Vj)
%           Ati = Vj vale Ej , expressão(Ej)
%           Exs = [At1, ...]
%           (R, RSExs) satisfaxem a especificação
%           RSExs é correspondente a R


  separar_expressoes_dos_predicados( [], []) .

  separar_expressoes_dos_predicados( [P|R], S) :-
      isolar_expressoes(P, PSE, Exs),
      separar_expressoes_dos_predicados(R, RSExs),
      append( [PSE|Exs], RSExs, S).
```

Temos que separar expressoes dos predicados estará parcialmente correto se provarmos a corretude de isolar expressoes. Como este já foi provado, concluimos então que separar expressoes dos predicados está parcialmente correto.

O próximo predicado, regra com expr isol, colocará a regra na forma de manipulação. Nesta forma, a regra será transformada em uma lista de predicados com todas as expressões isoladas.

Exemplo:

```
?- regra_com_expr_isol( fat(N,N*F1) se fat(N-1,F1), REI).
   REI = [fat(N,X),X vale N*F1,fat(N1,F1),N1 vale N-1].
```

```
% regra_com_expr_isol(R, REI)
% especificação
%
% 1- pré :   R = Fn1(A11, A12, ...) se Fn2(A21, A22, ...) &
%                                        Fn3(A31, A32, ...) & ...
%            (não_expressão(Aij) ou expressão(Aij) )
%
% 2- pós :   REI = [Fn (V1j, ...),  At1j, ...,
%                   Fn (V2j, ...),  At2j, ...,
%                   Fn (V3j, ...),  At3j, ...,
%                   ...],
%            não_expressão(Vij), Atij = Vik  vale Eik ,
%                                onde expressão(Eik)


   regra_com_expr_isol(Regra, REI) :-
       transformar_em_lista(Regra, LR),
       separar_expressoes_dos_predicados(LR, REI).
```

Temos que regra com expr isol estará parcialmente correto se transformar em lista e separar expressoes dos predicados estiverem corretos. Como estes dois predicados já foram provados, podemos concluir que regra com expr isol está correto.

A especificação informal da definição em PROLOG que calcula o padrão de fluxo de uma regra é a seguinte:

E1) São fêmeas ('-') todas as constantes da cabeça da regra.
E2) Sao fêmeas ('-') todas as variáveis a esquerda de vale.
E3) Sao fêmeas ('-') todas as variáveis que aparecem como tal nos predicados da cauda.
E4) Todas as outras variáveis são machos ('+').

Este algoritmo deve ser aplicado a cada regra de uma definição (conjunto de regras que definem um predicado), obtendo-se uma lista de padrões de fluxo. Devemos reduzir esta lista (combinando-a) a um único padrão. O seguinte algoritmo fará esta redução.

```
        reduz_padrões( [], P,P).
```

```
reduz_padrões( [P1|RP], P2, P) :-
    reduz_um_padrão(P1, P2, P3),
    reduz_padrões(RP, P3, P).

reduz_um_padrão(P1, P2, P) :-
    P1 =.. [Fn|Args1],
    P2 =.. [Fn|Args2],
    fluxos_combinados(Args1, Args2, Args),
    P =.. [Fn|Args].


fluxos_combinados([], [], []).
fluxos_combinados([X R1], [Y R2], [Z R]) :-
    combina_fluxos(X, Y, Z),
    fluxos_combinados(R1, R2, R).


combina_fluxo('+', '-', '+') .
combina_fluxo('-', '+', '+') .
combina_fluxo('-', '-', '-') .
combina_fluxo('+', '+', '+') .
```

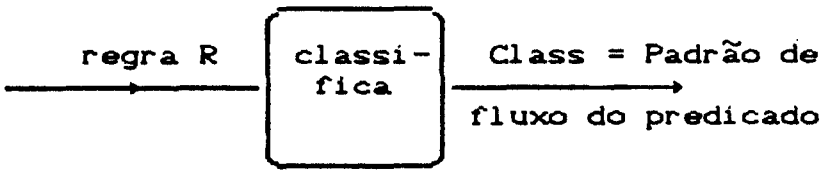
Os fluxos são combinados de forma que se o fluxo for '+' (exige-se que tenha uma constante) em um dos predicados que está sendo combinado, então o resultado deve ser '+'. Por outro lado, o '-' pode conter tanto uma variável quanto uma constante.

Vamos apresentar o algoritmo do predicado classifica que descobre o padrão de fluxo de dados de uma regra. Observe que o algoritmo deve ser aplicado recursivamente aos predicados da cauda e, portanto, corre-se o risco de ficar em um laço eterno. Para evitar o laço, cada chamada deve receber uma lista com seus antecessores. Uma nova chamada recursiva somente deve ser realizada se não pertencer a lista de antecessores. A manutenção e uso da lista de antecessores, embora não apresente dificuldades, tornam o algoritmo confuso, cheio de detalhes

irrelevantes para a questão principal, que é a ordenação dos predicados da cauda. Deste modo, não será incluida a recursividade.

O predicado classifica fornece o padrão de fluxo de dados de uma regra de uma definição.

Vejamos o DFD do predicado classifica .



Expandindo classifica teremos o seguinte DFD:

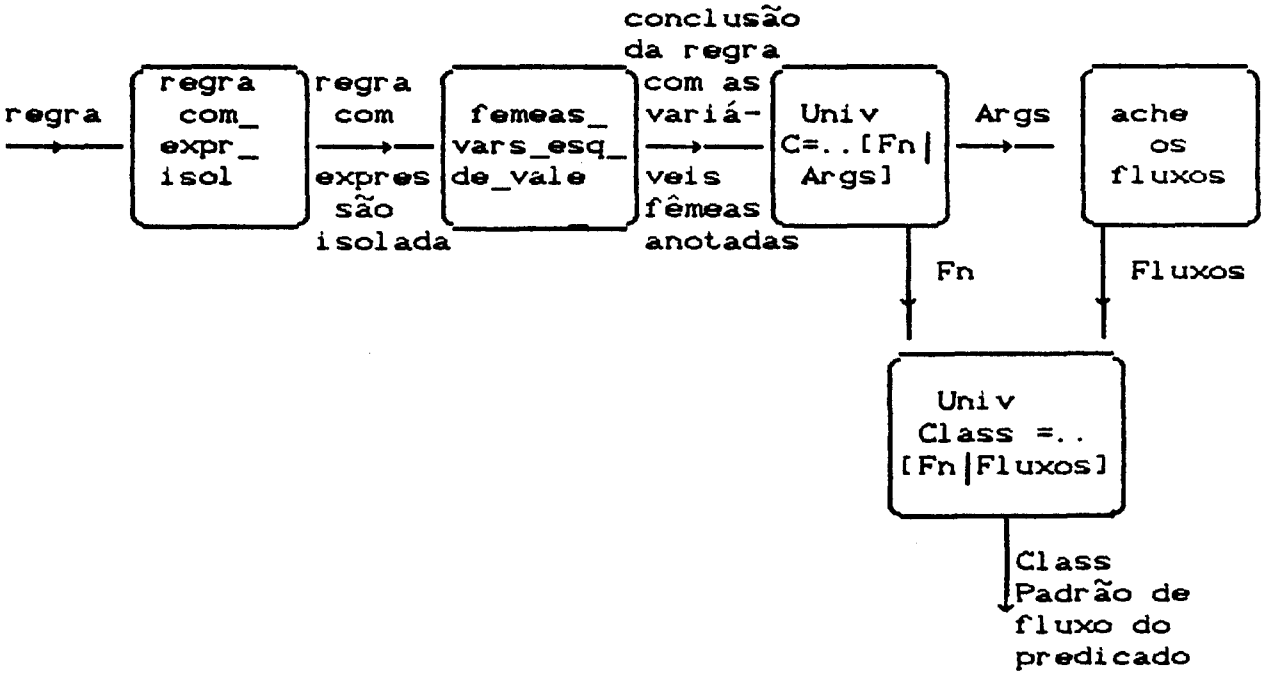


Figura 3.2 - O DFD do processo classifica

Exemplo:

```
?- classifica( fat(N,N*F1) se fat(N-1,F1), Class).

   Class = fat('+','-')

   Temos que:

   R = fat(N,N*F1) se fat(N-1,F1),

   após a execução de regra_com_expr_isol, temos que:
       [C|H] = [fat(N,X), X vale N*F1, fat(N1,F1), N1 vale N-1]

   após femeas_vars_esq_de_vale, temos que:
         Fn = fat,
         C = fat(N, femea(1)),
         Args = [N, femea(1)],

   após ache_os_fluxos, temos que:
         Fn = fat,
         Fluxos = ['+','-'],
         Class = fat('+','-')


   classifica(R, Class) :-
         regra_com_expr_isol(R, [C|H]),
         femeas_vars_esq_de_vale(H,1),
         C =.. [Fn|Args],
         ache_os_fluxos(Args,Fluxos),
         Class =.. [Fn|Fluxos].
```

Aplica-se o predicado femeas vars esq de vale às condições da regra para anotar as variaveis a esquerda de vale como fêmeas.

Exemplo:

```
?- femeas_vars_esq_de_vale([X vale N*F1,fat(N1,F1),N1 vale N-1],
                           1).
```

Após sua execução temos que X vale N*F1 torna-se femea(1) vale N*F1 e N1 vale N-1 torna-se femea(2) vale N-1.

```
%  femeas_vars_esq_de_vale(H, N)
%  especificação: H é uma lista de predicados (H= [P|R])
%                 N é um inteiro, contador de fêmeas
```

```
%   1- pré : H = [P|R], P = V vale E, N inteiro
%      pós : V = fêmea(N)
%            M is N + 1, M inteiro
%            (R,M) satisfazem a especificação

%   2- pré : H = [P|R], P \= X vale E, N inteiro
%            (R,N) satisfazem a especificação


  femeas_vars_esq_de_vale( [], N).

  femeas_vars_esq_de_vale([V vale E|R], N) :-
       V = femea(N),                        % pós-condição
       M is N + 1,                          % pós-condição
       femeas_vars_esq_de_vale(R,M).

  femeas_vars_esq_de_vale([P|R], N) :-
       P \= X vale Y,                       % pré-condição
       femeas_vars_esq_de_vale(R,N).
```

Vamos mostrar que o algoritmo femeas vars esq de vale está correto usando indução. No caso de H ser constituído por um único predicado do tipo V vale E, a variável V a esquerda do vale será anotado como fêmea. Isto está correto, pois a especificação 1 é forçosamente satisfeita pela segunda sentença da definição. Se o predicado não for do tipo V vale E, nada precisa ser feito. Está correto pois a especificação 2 é forçosamente satisfeita pela terceira sentença da definição. Admitamos que H = [P1|P2] e que o algoritmo esteja correto para P2. Neste caso, a primeira e segunda especificações estão satisfeitas.

Depois de anotadas as variáveis, convém achar os fluxos. O predicado que achará os fluxos chama-se ache os fluxos. Vejamos sua definição.

Exemplo:

```
?- ache_os_fluxos( [N, femea(1)], Padrao_de_Fluxo).
   Padrão_de_Fluxo = ['+,'-'].
```

```
% ache_os_fluxos(Args, Padrao_de_Fluxo)
% especificação

%  1- pré :  Args = [X|R], var(X)
%     pós :  Padrao_de_Fluxo = ['+'|RF],
%            (R, RF) satisfazem a especificação

%  2- pré :  Args = [X|R], nonvar(X), X = fêmea(N)
%     pós :  Padrao_de_Fluxo = ['-'|RF],
%            (R, RF] satisfazem a especificação

%  3- pré :  Args = [X|R], nonvar(X), X \= fêmea(N)
%     pós :  Padrao_de_Fluxo = ['-'|RF],
%            (R, RF] satisfazem a especificação



    ache_os_fluxos( [],[]).

    ache_os_fluxos( [X|R], ['+'|RF]) :-
           var(X),                  % pré-condição
           ache_os_fluxos(R, RF).

    ache_os_fluxos( [X|R], ['-'|RF]) :-
           nonvar(X),               % pré-condição
           X = femea(N),            % pré-condição
           ache_os_fluxos(R, RF).

    ache_os_fluxos( [X|R], ['-'|RF]) :-
           nonvar(X),               % pré-condição
           X \= femea(N),           % pré-condição
            ache_os_fluxos(R, RF).
```

No caso de Args ser constituído por um único argumento, temos que se o argumento for uma variável, o algoritmo está correto pois a especificação 1 é forçosamente satisfeita pela segunda sentença da definição. Se o argumento for não variável e do tipo fêmea(N), o algoritmo está correto, pois a especificação 2 está satisfeita pela terceira sentença da definição. Caso o argumento não seja variável, nem do tipo fêmea, o algoritmo está correto pois a especificação 3 está satisfeita pela quarta sentença da

definição. Admitamos que Args = [Arg1 | Arg2] e que o algoritmo esteja correto para RArgs. Neste caso, a primeira, segunda, e terceira especificações estão satisfeitas.

Vamos agora examinar cuidadosamente o funcionamento do predicado que achará fluxos de uma definição. Na figura 3.3 mostramos o DFD de padroes_de_fluxo.

Expandindo padroes_de_fluxo, temos:

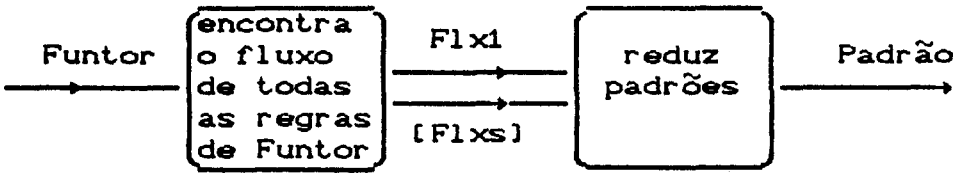


Figura 3.3 O DFD de padrões_de_fluxo

Temos duas regras de funtor fat, ou seja, compondo a definição de fat. A primeira é "fat(0,1)" tendo como fluxo "fat('-','-')". A segunda regra é "fat(N,N*F1) se fat(N-1,F1)" tendo como fluxo fat('+','-'). Temos, neste caso que a lista de fluxos é [fat('-','-'), fat('+','-')]. No DFD quebramos a lista pois reduz padroes precisa de dois argumentos de entrada: a cabeça (fat('-','-')) e a cauda da lista de fluxos ([fat('+','-')]). A redução feita por reduz padrões será a seguinte: Sendo o

primeiro argumento da regra '-' , e da segunda '+', o resultado da combinação é '+'. Como o segundo argumento das duas regras é '-' deve-se, portanto, produzir resultado '-'. Desta forma, o padrão final da definição fat é fat('+','-').

Os processos da figura 3.4 são:

```
padroes_de_fluxo( Funtor, Padrao) :-
    findall(Flx, fluxo_de_uma_regra(Funtor,Flx),[Flx1|Flxs]),
    reduz_padroes(Flxs, Flx1, Padrao).


fluxo_de_uma_regra(Fn, Flx) :-
    regra(Fn, Nro, R),
    classifica(R, Flx).
```

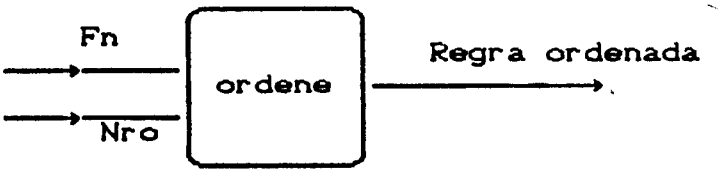
Exemplo:

```
?- padroes_de_fluxo( fat, Padrao).
   Padrao = fat('+','-')

   Temos que:
      Funtor = fat,

   após findall temos:
      Flx1 = fat('-','-),
      [Flxs] = [fat('+''-')],

   após reduz_padroes temos:
      Padrao = fat('+','-')
```

Suponhamos que o fluxo de todos os predicados tenha sido determinado conforme discutido e (automaticamente) colocado na base de dados da seguinte forma:

```
fluxo(fat,['+','-']).
```

Vamos analisar como a regra pode ser ordenada. Na figura 3.4 temos o DFD do predicado ordene.



Expandindo ordene temos:

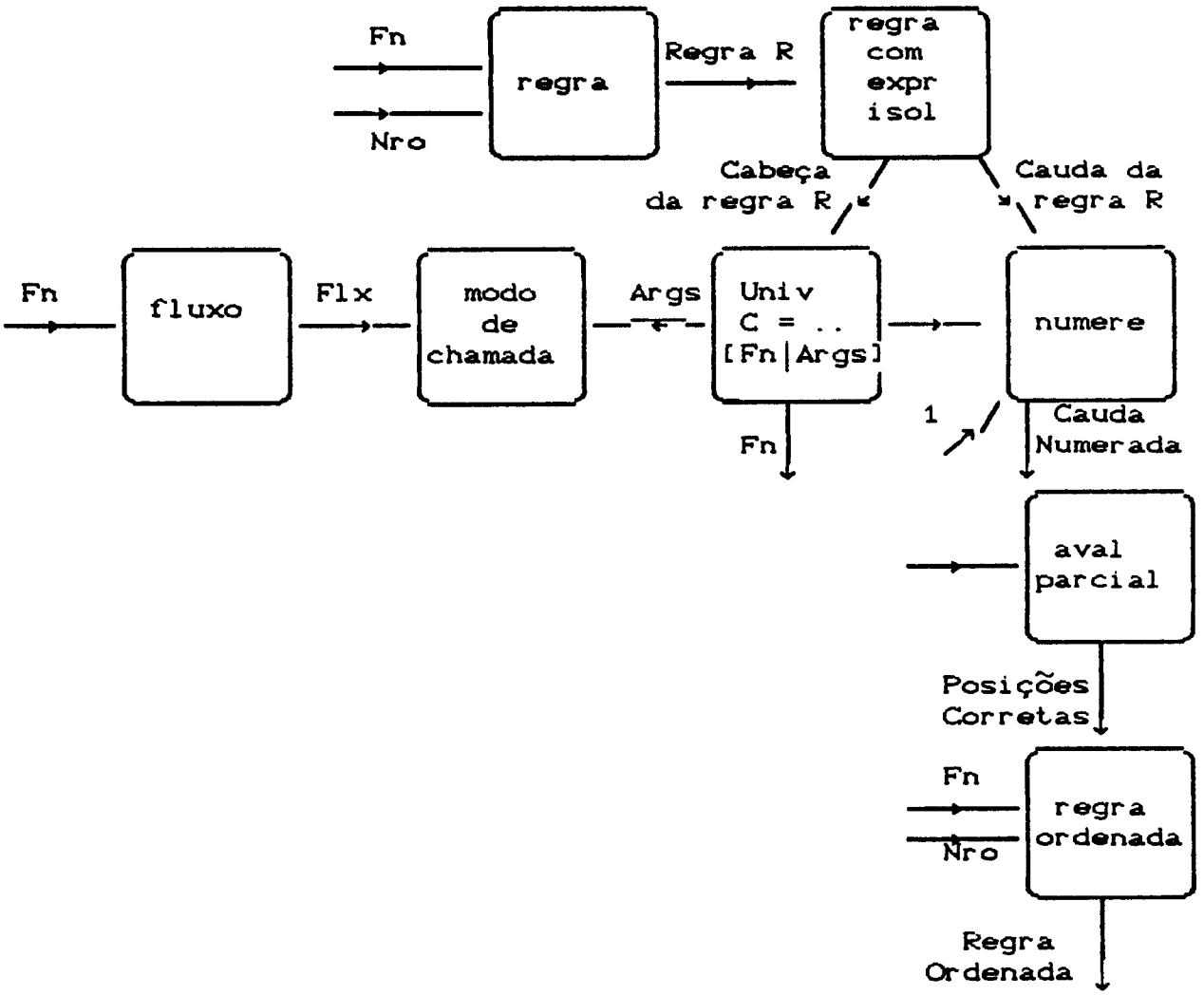


Figura 3.4 O DFD de ordene

O processo numere do DFD da figura 3.4 anota cada predicado com sua posição na cauda. Por exemplo, o predicado X vale N*F1 ocupa a primeira posição e, portanto, é anotado com o número 1. Temos também que fat(N1,F1) é anotado com 2 e N1 vale N-1 é anotado com 3. Em seguida, aval_parcial fornecerá as posições que cada predicado deveria ocupar após ordenado. A lista [3,2,1] indica que o predicado anotado com 3 deve ocupar a primeira posição e o predicado anotado com 2 deve ficar na segunda posição. Finalmente, o predicado anotado com 1 fica na última posição. O predicado ordene é o seguinte:

```
ordene(Fn, Nro, Regra) :-
     regra(Fn, Nro, R),
     regra_com_expr_isol(R, [C|H]),
     fluxo(Fn, Flx),
     C =.. [Fn|Args],
     modo_de_chamada(Args, Flx),
     numere(H, 1, HNumerado),
     aval_parcial(HNumerado, NH),
     regra_ordenada(Fn, Nro, NH, Regra).
```

Exemplo:

```
?- ordene(fat, 2, Regra).
   Regra = fat(N,X) se N1 vale N-1 & fat(N1,F1) & X vale N*F1.

   Temos que:

      Fn = fat,
      Nro = 2,
      R = fat(N,N*F1) se fat(N-1, F1),

   após regra_com_expr_isol, temos que:
       [C|H] = [fat(N,X), X vale N*F1, fat(N1,F1),N1 vale N-1]

   após fluxo, temos que:
       Flx = ['+','-'],
       Args = [N,X],

   após numere, temos que:
       HNumerado = [X vale N*F1:1, fat(N1,F1):2, N1 vale N-1:3]

   após aval_parcial, temos que:
```

```
NH = [3,2,1],
```

após regra_ordenada, temos que:

```
Regra = fat(N,X) se N1 vale N-1 & fat(N1,F1) & X vale N*F1
```

O predicado numere é muito simples e, como dissemos, vai anotar cada predicado com a posição que ele ocupa na cauda.

Exemplo:

```
?- numere([X vale N*F1, fat(N1,F1), N1 vale N-1], HNumerado).
   HNumerado = [X vale N*F1:1, fat(N1,F1):2, N1 vale N-1:3]

    numere( [], N, []).

    numere( [X|Y], N, [X:N|NY]) :-
         M is N + 1,
         numere(Y,  M, NY).
```

Vejamos o predicado modo de chamada. Temos que cada argumento macho ("entrada obrigatória") deve receber uma constante na hora da chamada. A questão é que não sabemos que constante é esta mas sabemos que, na hora da chamada, não teremos uma variável no argumento. Podemos usar este fato para fazer uma avaliação parcial. O resultado desta avaliação parcial será a ordem dos predicados da cauda. Vejamos como será feita esta avaliação.

O 1º passo da avaliação parcial será a unificação da cabeça da regra. Acontece que, para fazer isto, ainda não conhecemos os argumentos da chamada. Sabemos apenas quais argumentos serão conhecidos, ou seja, aqueles marcados com '+' (os de entrada obrigatória). Podemos, então, preparar um unificador "parcial" que coloque a constante 'g' nas variáveis de fluxo '+'. O unificador parcial é o predicado modo de chamada e tem a seguinte

definição.

Exemplo:

```
?- modo_de_chamada( [N,X], ['+','-']).
```

Pelo fato de que o argumento N é de entrada obrigatória, após a execução de modo de chamada, temos que:

N = 'g'

```
modo_de_chamada( [], []).

modo_de_chamada( [X|Y], ['+'|Flxs]) :-
        nonvar(X),
        modo_de_chamada(Y, Flxs).

modo_de_chamada( [X|Y], ['+'|Flxs]) :-
        var(X),
        X = g,
        modo_de_chamada(Y, Flxs).

modo_de_chamada( [X|Y], ['-'|Flxs]) :-
        modo_de_chamada(Y, Flxs).
```

Neste unificador "parcial" toda variável em argumento macho recebeu uma constante 'g', exceto quando já existia uma constante no argumento.

O 2º passo da avaliação parcial é continuar a avaliação parcial pela resolução da cauda. O que sabemos sobre os dados da cauda é o mesmo que sabemos sobre a cabeça, ou seja, sabemos que os marcados com '+' terão valores conhecidos no momento da unificação. Isto é suficiente para realizar a "escolha" dos predicados a serem resolvidos em cada passo da prova em tempo de compilação. O predicado que fará a escolha é o seguinte:

```
escolha( [X|Y], X, Y).

escolha( [X|Y], D, [X|Y1]) :-
        escolha(Y,D,Y1).
```

Este predicado produz todas as escolhas possíveis quando não se tem informações sobre os dados. Por exemplo:

?- escolha([a,b,c], D, F). Produzirá as seguintes soluções:

```
D = a
F = [b, c]  →  ;

D = b
F = [a, c]  →  ;

D = c
F = [a, b].
yes.
```

Vamos examinar como deve ser o processo de avaliação parcial. O predicado é aval_parcial, o qual irá descobrir a ordem dos literais na cauda da regra.

Exemplo:

```
?- aval_parcial([X vale N*F1:1, fat(N1,F1):2, N1 vale N-1:3],P)
   P = [3,2,1]

        aval_parcial( [], []).

        aval_parcial( R, [N|Y]) :-
             R = [_|_],
             escolha(R, C:N, R1),
             resol_parcial(C),
             aval_parcial(R1,Y).
```

Temos que cada literal da cauda está anotado com seu número (posição). O predicado escolha(R, C:N, R1) retira da cauda R, o literal anotado por C:N para ser resolvido parcialmente. R1 é a

cauda R sem o literal escolhido C:N. Em seguida, tenta-se realizar a resolução parcial de C. Se a resolução parcial não tiver êxito, ou seja, se resol_parcial(C) falhar, escolha fornecerá outro C:N. Caso resol_parcial(C) tenha êxito, o número do literal retirado (que teve êxito) da cauda R é colocado na lista [N|Y]. Tenta-se, então, realizar a avaliação parcial do restante da cauda, ou seja, de R1. No final da execução, teremos uma lista [N|Y] com os números indicando a ordem em que os predicados devem ser escolhidos.

Vejamos como se processa a resolução parcial. O predicado é seguinte:

```
resol_parcial( V vale Expr) :-
        unif_atr(V),
        unif_expr(Expr).

resol_parcial(F) :-
        F \= V vale Expr,
        F =.. [Fn|Args],
        fluxo(Fn, Modos),
        unif_parcial(Args, Modos).
```

Primeiro vejamos como se faz a resolução parcial de expressões (unif_expr). A variável a esquerda do vale deve receber um valor. Não sabemos ainda que valor é este. Devemos, portanto, colocar um 'g' na variável.

```
unif_atr(X) :- nonvar(X).

unif_atr(X) :- var(X), X = g.
```

Todo lado direito do vale deve ser conhecido, ou seja, conter constantes no momento da resolução. Isto é feito pelo predicado unif_expr(E).

```
unif_expr(E) :-
        nonvar(E),
        E = X + Y,
        unif_expr(X),
        unif_expr(Y).

unif_expr(E) :-
        nonvar(E),
        E = X - Y,
        unif_expr(X),
        unif_expr(Y).

unif_expr(E) :-
        nonvar(E),
        E = X * Y,
        unif_expr(X),
        unif_expr(Y).

unif_expr(E) :-
        nonvar(E),
        E = X / Y,
        unif_expr(X),
        unif_expr(Y).

unif_expr(E) :- atomic(E).
```

Vejamos agora como se faz resolução parcial de predicados que não são expressões (unif_parcial). A unificação parcial deve verificar cada argumento e certificar-se de que todos os machos tem uma constante. Supõe-se que o predicado produza valores nos argumentos fêmeas. Como não sabemos que valores são estes, unificamos as variáveis com 'g'. O unificador parcial é o seguinte:

```
unif_parcial( [], []).

unif_parcial( [B|C], ['+'|Y]) :-
      nonvar(B),
      unif_parcial(C,Y).

unif_parcial( [B|C], ['-'|Y]) :-
      nonvar(B),
      unif_parcial(C,Y).
```

```
unif_parcial( [B|C], ['-'|Y]) :-
        var(B),
        B = g,
        unif_parcial(C,Y).
```

Tendo, através de resol_parcial, obtido a ordem correta dos predicados da cauda, podemos ordená-los com ponha_em_ordem, o qual possui a seguinte definição.

Exemplo:

```
?- ponha_em_ordem([X vale N*F1,fat(N1,F1),N1 vale N-1],[3,2,1],
                  ROrdenada).
   ROrdenada = [N1 vale N-1, fat(N1,F1), X vale N*F1]

      ponha_em_ordem(R, [], []).

      ponha_em_ordem(R, [M|RM], [X|Y]) :-
            indice(R,1,M,X),
            ponha_em_ordem(R, RM, Y).


      indice( [X|Y], N, M, X) :- N = M.

      indice( [_|Y], N, M, X) :-
            N \= M ,
            N1 is N + 1,
            indice(Y, N1, M, X).
```

Finalmente, dado o nome do funtor, o número da regra, e a numeração da cauda, o predicado regra ordenada irá remontar a cauda de modo que ela obedeça à numeração. Esta numeração foi obtida, como vimos, pelo aval_parcial.

Exemplo:

```
?- regra_ordenada(fat,2,[3,2,1], C se Cauda) .
   C se Cauda = fat(N,X) se N1 vale N-1 & fat(N1,F1) &
                            X vale N*F1

   Temos que:
        Fn = fat,
        Nro = 2,
        NH = [3,2,1],
        R = fat(N,N*F1) se fat(N-1,F1),
```

```
após regra_com_expr_isol, temos que:
     Cab = fat(N,X),
     H = [X vale N*F1, fat(N1,F1), N1 vale N-1],

apos ponha_em_ordem, temos que:
     H_Ordenado = [N1 vale N-1, fat(N1,F1), X vale N*F1],

apos remonte_cauda, temos que:
     Cauda = N1 vale N-1 & fat(N1,F1) & X vale N*F1
     C = fat(N,X)

regra_ordenada(Fn, Nro, NH, Cab se Cauda) :-
       regra(Fn, Nro, R),      % obtém regra da base de dados
       regra_com_expr_isol(R, [Cab|H]),
       ponha_em_ordem(H, NH, H_Ordenado),
       remonte_cauda(H_Ordenado, Cauda).
```

A cauda depois de ordenada pelo predicado ponha em ordem estará na forma [N1,N2,N3,...]. O predicado remonte cauda irá colocá-la na forma N1 & N2 & N3 & ....

Exemplo:

```
?- remonte_cauda([N1 vale N-1, fat(N1,F1), X vale N*F1], R).
   R = N1 vale N-1 &  fat(N1,F1) & X vale N*F1

          remonte_cauda([], true).
          remonte_cauda([X], X).
          remonte_cauda([X,Z|Y], X & R) :-
               remonte_cauda( [Z|Y], R).
```

O exemplo apresentado refere-se a um sistema extremamente simplificado. É possível extender as idéias apresentadas para sistemas bem mais sofisticados.

Há vários artigos escritos sobre o uso de avaliação parcial para aumentar a eficiência de programas. Não sabemos, entretanto, de nenhum trabalho que fale do uso de avaliação parcial para introdução de controle. Aliás, são poucos os trabalhos sobre a introdução de controle em sistema especialista para aumentar-lhes

a eficiência, conforme o trabalho de Smith [Smith-89].

Smith [Smith-89] apresentou um método de controlar inferências regressivas independente do domínio, no qual computa o custo esperado e probabilidade de sucesso para diferentes estratégias regressivas. Estas informações são usadas para selecionar quais os passos de inferência e computar a melhor ordem para o processamento das conjunções. Para o cálculo do custo e da probabilidade, considera informações simples sobre o conteúdo da base de dados, tais como: o número de fatos de uma dada forma e tamanho do domínio para os predicados e relações envolvidas. Em seu artigo cita apenas os trabalhos de Natarajan os quais usam abordagens semelhantes, porém que não se aplicam ao caso em que estudamos. Para maiores detalhes ver [Natarajan, K. S, "On Optimizing backtrack search for all solutions to conjuntive problems", Research Report, 11982, IBM (1986) e "Optimizing depth-first search of AND-OR trees, Research Report, 11842, IBM (1986)].

3.5 CONCLUSÃO

Neste capítulo apresentamos as dificuldades de introduzir automaticamente controle explícito em sistemas especialistas, ou seja, o problema da escolha do literal a ser provado. Usando o exemplo do fatorial mostramos o funcionamento do algoritmo.

CAPÍTULO 4

CONCLUSÕES E EXTENSÕES

Apresentamos neste trabalho um algoritmo que ordena as condições de cada regra da base de conhecimentos, de modo que os literais fiquem em ordem de prioridade para resolução. Após esta ordenação, o mecanismo de resolução do sistema especialista deve sempre pegar o literal mais a esquerda para fazer a resolução, eliminando-se assim o problema da escolha do literal da hipótese.

O critério usado para se obter a prioridade dos literais foi o seguinte: quando um mesmo argumento ocorrer em mais de um literal, os literais nos quais ele ocorre como entrada obrigatória, devem aparecer após os literais nos quais ele ocorre como entrada_saída. Nos casos em que o argumento ocorre somente uma vez, ou ocorre mais de uma vez, porém sempre com o mesmo tipo de entrada, não há diferenças de prioridade. Ao ordenarmos estamos introduzindo controle na base de conhecimentos.

Observemos que neste caso, a introdução do controle é efetuada na fase da aquisição. Desta forma, o sistema especialista ficará muito mais rápido na fase da consulta, pois a escolha da estratégia já foi determinada na fase da aquisição. Obtemos

assim, sistemas especialistas mais eficientes e cujo controle é obtido automaticamente.

O protótipo implementado é restrito para linguagens de representação do conhecimento da forma P se Q_1 & Q_2 & ... & Q_n, cujos argumentos são variáveis, constantes, ou expressões (com operadores + , - , * , /). Podemos extendê-lo para tratar outros tipos de argumentos tais como: estruturas de dados, outros operadores, etc.

Uma possível extensão seria na escolha da melhor estratégia (ordem), em termos de eficiência. Neste caso, seria necessário uso de métricas que auxiliassem na decisão. No nosso protótipo, apenas fornecemos as estratégias possíveis.

Tratamos apenas da introdução de controle sobre a escolha da hipótese a ser provada. Não introduzimos o controle sobre a escolha da regra (disjunção).

Usando o mesmo algoritmo o protótipo pode ser extendido para tratar regras contendo negação.

Podemos aplicar esta técnica, introdução automatica de controle e avaliação parcial, em sistemas nos quais deseja-se que o controle seja fornecido pelo próprio sistema com base em regras previamente dadas.

O protótipo foi implementado em Arity-Prolog para micros compatiiveis ao IBM-PC XT.

BIBLIOGRAFIA

[ARARIBOIA-89] Araribóia G., "Inteligência Artificial", LTC, 1989.

[CLOCKSIN-84] Clocksin, W. F. & Mellish C. S., "Programming in Prolog", second edition, cap. 10, appendix B, Springer Verlag, 1984.

[CHANG-73] Chang C. L. & Lee R. C.T.,"Symbolic Logic and Mechanical Theorem Proving", Academic Press, New York, 1973.

[DELAHAYE-86] Delahaye, J., "Outils Logiques pour L'Inteligence Artificielle", Eyrolles, 1986.

[DUDA-76] Duda, R. O., Hart. P. E., & Nilsson, N. J., "Subjective Bayesian Methods for Rule-Based Inference Systems", Report TR-124, Artificial Intelligence Center, SRI International, 1976.

[ERSHOV-77] Ershov, A. P., "On the Partial Computation Principle", Information Processing Letters, Vol.6, N.2, p. 38-41, April, 1977.

[ERSHOV-82] Ershov, A. P., "Mixed Computation: Potential Applications and Problems for Study", Theoretical Computer Science,Vol. 18, p. 41-67, North Holland Publishing Company, 1982.

[FUTAMURA-83] Futamura Y., "Partial Computation of Programs", Journal of IECE of Japan, Vol.66, No. 2, 1983.

[GRIES-87] Gries D., "The Science of Programming", caps. 1 a 4, Springer Verlag, 1987.

[HAYES-73] Hayes Roth F., Waterman D. A. & Lenat D. D., "Building Expert Systems", Addison-Wesley, 1973.

[JAEGER-86] Jaeger W., "Paidéia", Ed.Universidade de Brasília, 1986.

[KAHN-84] Kahn, K. M., "The Compilation of Prolog Programs without the use of a Prolog Compiler", Tech. Report, UPMAIL, Uppsala, University, Sweeden, 1984.

[LOMBARDI-67] Lombardi L.A.,"Incremental Computation Advances Computers", 8, Academic Press, New York, 1967.

[NEGOITA-85] Negoita V.C., "Expert Systems and Fuzzy Systems", The Benjamin/Cummings Publishing Company Inc,1985.

[NILSSON-82] Nilsson Nils J., "Principles of Artificial Intelligence", Springer-Verlag, 1982.

[PAGAN-88] Pagan,F. G., "Converting Interpreters into Compilers", Software - Practice and Experience, Vol. 18, No.6, p. 509-527, June, 1988.

[ROBINSON-65] Robinson, J.A. "A Machine-Oriented Logic Based on the Resolution Principle", Journal of the ACM, Vol.12, January, 1965.

[SHAFFER-87] Shaffer G & Logan R., "Implementing Dempster's Rule for Hierarquical Evidence", Artificial Intelligence, Vol. 33, No. 3, p. 271-298, November, 1987.

[SHOENSIELD-67] Shoensield J. R., "Mathematical Logic", Addison-Wesley, 1967.

[SMITH-89] Smith, D. E., "Controlling Backward Inference", Artificial Intelligence, No. 39, p. 245-208,1989.

[STERLING-86] Sterling L., & Beer R. D., "Incremental Flavor-Mixing of Meta-Interpreters for Expert System Construction", IEEE, 1986.

[TAKEUCHI-85] Takeuchi, A. & Furukawa, K., "Partial Evaluation of Prolog Programs and its Application to Meta Programming", ICOT Tech. Report, 1985.

[TANIMOTO-87] Tanimoto, S. L., "The Elements of Artificial Intelligence - An Introduction Using LISP", Computer Science Press, 1987.

[VENKEN-85] Venken, R.,"A Prolog Meta-interpreter for Partial Evaluation and its Apllication to Source Transformation and Query-Optimisation", Advances in Artificial Intelligence, p. 347-357, 1985.