Teoria das Provas

A Teoria das Provas preocupa-se com a *derivabilidade* de frases lógicas. A partir de um conjunto de frases iniciais e usando regras de inferência, é possível derivar novas frases.

As frases iniciais são chamados *axiomas*

As frases derivadas são chamadas *Teoremas* (ou consequências sintácticas)

Regras de inferência operam unicamente sobre a sintaxe das frases (e não sobre o seu significado). Exemplos de regras de inferência vulgarmente encontradas:

Da equação "$X + Y = Z$"

inferir a equação "$X = Z - Y$".

A regra mais popular (usada pela lógica clássica):

Modus Ponens:

"do par de frases $\{B, A \leftarrow B\}$ inferir a frase A".

Os axiomas mais as regras de inferência constituem um *sistema de inferência.*

Os axiomas juntamente com os teoremas derivados constituem uma *Teoria.*

(Definição formal de *Prova*):

Seja A um conjunto de axiomas e R um conjunto de regras de inferência. Uma prova derivada do sistema de inferência $[A, R]$ é a sequência

$$< s_1, s_2, ..., s_n >$$

em que cada s_i é:

- um axioma de A, ou
- é derivado através da aplicação de uma regra de R em axiomas de A, ou
- através da aplicação de uma regra R em frases que precedem s_i.

Exemplo:

A sequência

$$< D, C, A >$$

é a forma de derivar A usando Modus Ponens e o conjunto inicial

$$\{D, D \rightarrow C, C \rightarrow A\}.$$

A sequência é denominada por prova de s_i i.e. $A \vdash s_i$

A *relação de derivabilidade* usando R é definida por:

$$\vdash = \{(A, s) \mid \text{s é provado através de } A \text{ usando } R\}$$

Uma teoria é *inconsistente* se e só se contém conjuntamente uma frase e a sua negação. Caso contrário é *consistente*.

Exemplos de uma teoria Inconsistente:

$$\{A \vee \neg B, \neg A, B\}$$
$$\{\neg B, A \vee B, B\}$$
$$\{A \wedge \neg A\}$$

Exemplos de uma teoria Consistente:

$$\{A \vee \neg B, A\}$$
$$\{A \vee \neg A, B\}$$

Resolução de Problemas e Implicação Lógica

Seja S um conjunto de frases lógicas e s uma conclusão.

Requisito mínimo:

Se S expressa correctamente o nosso problema a resolver

então s expressa uma conclusão correcta.

Se correcto quer dizer "*verdade numa interpretação própria*" então podemos dizer que:

Para *uma* interpretação I:

Se I é um modelo de S

então I é um modelo de s

Mas o computador só manipula símbolos (e nada sabe sobre as nossas intenções). Algo mais forte será:

Para **todas** as possíveis interpretação I:

Se I é um modelo de S

então I é um modelo de s

Por outras palavras: queremos mostrar que $S \models s$

Felizmente, é possível mostrar que $S \models s$ sem ter que verificar todas as interpretações.

Usamos simplesmente um sistema de inferência que estabeleça a relação $S \vdash s$.

Duas propriedades essenciais !

- **Correcto** (Soundness) $S \models s$ sempre que $S \vdash s$
- **Completo** (Completeness) $S \vdash s$ sempre que $S \models s$

Um sistema que não é completo não tem poder inferencial suficiente para resolver todos os problemas que têm solução.

Exemplo de um sistema completo/correcto:

modus ponens ($B, A \leftarrow B$ então A) com

Transporte de literais (e.g. $A \ \vee \ B \leftarrow C \equiv A \leftarrow \neg B \ \& \ C$).

Um problema com o uso de *modus ponens* surge pelo facto da regra ser usada sem particular objectivo de atingir uma conclusão específica. Assim, pode inferir coisas em que não temos interesse. Considerando este detalhe, é geralmente melhor usar um sistema de inferência com a seguinte regra:

$$\{\neg A \ \vee \ literais), (A \leftarrow B)\} \vdash \neg B \ \vee \ literais$$

Um caso especial desta regra é *modus tollens*:

$$\{\neg A, (A \leftarrow B)\} \vdash \neg B$$

Com este novo método, primeiro nega-se a desejada conclusão e depois tenta-se inferir a cláusula vazia ($\square$) (inconsistência).

Generalizando, tendo um programa P (conjunto de cláusulas), uma desejada conclusão A e um sistema de inferência que é correcto e completo para $\vdash$. Então:

$$P \cup \{\neg A\} \models \square \text{ se e só se } P \models (falso \leftarrow \neg A)$$

$$P \cup \{\neg A\} \models \square \text{ se e só se } P \models A$$

$$P \cup \{\neg A\} \vdash \square \text{ se e só se } P \vdash A$$

Muito Importante

Assim, se a conclusão A pode ser inferida directamente de P, então a sua negação $\neg A$ pode ser *refutada* de P.

Qualquer prova de $\square$ vai ser apresentada como uma *Refutação*

A execução de um programa em lógica é baseado no método de provas por refutação. Dado um programa P e uma fórmula negada $\neg A$ (ambas em forma clausal) tentamos mostrar que:

$$P \cup \{\neg A\} \text{ é inconsistente}$$

Nos primórdios da área de Provas Assistidas por Computador (*Automated Reasoning*) os métodos usados para testar a inconsistência de um conjunto de cláusulas eram chamados *Procedimentos de Herbrand.* Usavam o Teorema de Herbrand e a versão *ground* de um programa i.e. a sua versão proposicional.

A versão proposicional do programa

$$P = \{p(X) \leftarrow q(X), q(ana), q(luis)\}$$

tendo em conta o domínio

$$H = \{ana, luis\}$$

é

$$G(P) = \{p(ana) \leftarrow q(ana), p(luis) \leftarrow q(luis), q(ana), q(luis)\}$$

O *teorema de Herbrand* diz que:

$P \cup \{\neg A\}$ é inconsistente

se e só se

um subconjunto *finito* de $G(P \cup \{\neg A\})$ é inconsistente

Estes métodos eram extremamente ineficientes (usavam tabelas de verdade) e foram abandonados em favor de métodos mais eficientes: e.g. *Resolução.*

Resolution: (versão proposicional)

Resolução é uma regra de inferência que, usada isoladamente, é suficiente para testar se um dado conjunto de cláusulas é inconsistente (derivando $\Box$).

Resumidamente resolução é, dado o conjunto de cláusulas P:

1. escolher uma cláusula $A \vee literais \in P$
2. procurar outra cláusula $\neg A \vee mais\_literais \in P$
3. construir a cláusula $literais \vee mais\_literais$
4. adicionar a cláusula construída a P

Exemplo:

1. dado: $mae \vee pai \vee \neg progenitor$
2. com: $homem \vee \neg pai$
3. construir: $mae \vee homem \vee \neg progenitor$

As cláusulas escolhidas para um passo de resolução são chamadas *progenitoras* desse passo. A nova cláusula derivada de um passo de resolução chama-se *resolvente*.

Nota: Cláusulas de Horn são da forma

$$c \leftarrow a_1 \wedge a_2 \wedge \ldots \wedge a_n$$

Propriedades da Resolução:

1. É correcto: cada resolvente é implicado pelos seus progenitores.
2. Um resolvente é a cláusula vazia ($\Box$) se e só se um progenitor é uma fórmula atómica A e o outro a sua negação ($\neg A$).
3. Para qualquer conjunto de cláusulas de Horn, a cláusula vazia pode ser inferida como resolvente. Resolução é um sistema de inferência completo (refutation-complete).
4. Resolução pode ser usada unicamente com frases lógicas na forma clausal.

Uma consequência da resolução ser um sistema de inferência correcto e completo:

$$P \models A \text{ se e só se } P \cup \{\neg A\} \vdash_{res} \Box$$

No caso em que usamos cláusulas não Horn é (e.g. $A \leftarrow \neg B$) necessário recorrer a uma regra adicional chamada *factoring* (a ser explicada conjuntamente com resolução para lógica do primeiro grau).

Por exemplo, tentar obter a cláusula $\Box$ a partir das cláusulas $\{A \vee A, \neg A \vee \neg A\}$

Exemplo com cláusulas de Horn:

1. $\neg A$
2. $A \vee \neg B \vee \neg C$
3. $B \vee \neg C$
4. C

5) de 1. e 2. infere-se $\neg B \vee \neg C$

6) de 3. e 5. infere-se $\neg C \vee \neg C$

7) de 4. e 6. infere-se $\neg C$

8) de 4. e 7. infere-se $\square$

Note-se que outras refutações alternativas existem e podem ser obtidas por resolução.

Resolução em lógica de Predicados

Resolução com cláusulas que contêm variáveis é similar à resolução proposicional mas requer a aplicação de substituições com o intuito de substituir certas variáveis por termos específicos.

Seja a *substituição* $\sigma = \{X_1/t_1, X_2/t_2, ..., X_n/t_n\}$

Uma *aplicação* de σ numa fórmula W é a substituição de todas as ocorrências da variável X_i pelo termo t_i.

Exemplo:

$W = gosta(X, Y)$ e $\sigma = \{X/ana, Y/logica\}$

então $W\sigma = gosta(ana, logica)$.

mas também

$\sigma_1 = \{X/t(Z), Y/logica\}$

então $W\sigma_1 = gosta(t(Z), logica)$.

Resolução em lógica de Predicados (cont)

Noção importante: Um *unificador* para as fórmulas $W1$ e $W2$ é a substituição θ tal que $W1\theta = W2\theta$.

O *unificador mais geral* entre duas fórmulas $W1$ e $W2$ é o unificador θ desta duas fórmulas tal que, para qualquer outro unificador das duas fórmulas σ, podemos ter

$$W1\sigma = (W1\theta)\lambda$$

(sendo *lambda* uma substituição em $W1$).

Exemplo:

$\sigma = \{X/a, Y/a\}$ é um unificador para $P(X)$ e $P(Y)$, mas não é o unificador mais geral. O unificador mais geral é $\theta = \{X/Y\}$.

Mas já sabemos como fazer instâncias de fórmulas através de substituições.

Unificação

O processo de calcular unificadores mais gerais chama-se *unificação* e é um passo importante que ocorre no sistema dedutivo de resolução. Em termos gerais, podemos dizer que funciona como um sistema de resolução de equações.

O algoritmo de unificação calcula o unificador mais geral entre dois termos se ele existir ou retorna insucesso (*failure*) no caso contrário.

O algoritmo usa uma stack para guardar as "equações" a serem resolvidas. Guarda também uma localização de memória θ com uma substituição que é a resposta do algoritmo.

Inicialmente θ está vazia e a stack contém a equação $T1 = T2$ (sendo $T1$ e $T2$ os termos a unificar). O algoritmo consiste num ciclo de processamento de equações retiradas da stack. O ciclo pára quando o processamento de uma equação acaba em "failure" ou quando a stack fica vazia.

Unificação (cont)

Algoritmo Unify

Dados dois termos $T1, T2$ a serem unificados
devolve θ, o unificador mais geral, ou $failure$.

```
    θ ← ∅
    push(stack, T1 = T2)
    failure ← false
    while stack ≠ ∅ & ¬failure do
       pop X = Y da stack
       case
          X é uma variável que não ocorre em Y
(*)          substituir Y por X na stack e em θ
(*)          adicionar X = Y a θ
          Y é uma variável que não ocorre em X
(*)          substituir X por Y na stack e em θ
(*)          adicionar Y = X a θ
          X e Y sao constantes ou variáveis idênticas
             continue
          X é f(X_i, ..., X_n) e Y é f(Y_1, ..., Y_n)
          para um functor f e n > 0
             push(X_i = Y_i, stack) para i = 1, ..., n
          otherwise
             failure ← true

    if failure then return failure
    else return θ
```

Notar que as afirmações (*) podem ser substituidas por

fazer uma referência de X para Y

O teste *não ocorre* é conhecido por *occurs check* e é necessário para prevenir unificações infinitas. Isto é, unificações de termos onde não existe instâncias finitas comuns e.g. $s(X)$ e X.

Resolução (com variáveis)

Consideremos as seguintes cláusulas:

C1 $A(arg1) \ \vee \ literais$

C2 $\neg A(arg2) \ \vee \ mais\_literais$

Se existir um unificador mais geral θ para os átomos A tal que:

$$A(arg1)\theta = A(arg2)\theta$$

então estas duas cláusulas podem ser progenitoras do passo de resolução, em que o resolvente é:

C3 $(literais \ \vee \ mais\_literais)\theta$

Exemplo

C1 $gosta(ana, X) \leftarrow gosta(X, logica)$

C2 $gosta(luis, U) \leftarrow alegre(U)$

$\theta = \{X/luis, U/logica\}$, o que dá o resolvente:

C3 $gosta(ana, luis) \leftarrow alegre(logica)$

Factoring

Regra necessária para que a resolução seja completa para cláusulas não Horn.

Esta regra simples diz que dada a cláusula:

$$L1 \lor L2 \lor outros\_literais$$

podemos correctamente inferir a cláusula

$$(L1 \lor outros\_literais)\theta$$

se e só se $\exists\theta \;\; \theta = m.g.u(L1, L2)$

Exemplo

$$\mathsf{gosta(marcelo, X)} \leftarrow \mathsf{apoia(X, Y)} \;\&\; \neg\mathsf{gosta(Y, regionalizacao)}$$

inferir

$$\mathsf{gosta(marcelo, regionalizacao)} \leftarrow \mathsf{apoia(regionalizacao, marcelo)}$$

Programação em Lógica e Resolução:

Dado um programa P, uma interrogação (query) tem a forma:

$$?A_1 \ \& \ ... \ \& \ A_n$$

que é lida em lógica como:

$$\neg\exists(A_1 \ \& \ ... \ \& \ A_n)$$

Para responder à query usando o programa P, aplica-se resolução para testar se:

$$P \cup \{\neg\exists(A_1 \ \& \ ... \ \& \ A_n)\} \vdash \Box$$

e assim concluir

$$P \models \exists(A_1 \ \& \ ... \ \& \ A_n)$$

Resolução é um principio construtivo em que são retornados valores para as variáveis presentes na query. Estes valores são gerados pelo processo de unificação.

Exemplo:

C1 $antepassado(X, Y) \leftarrow pai(X, Y)$

C2 $antepassado(X, Y) \leftarrow antepassado(X, Z)\ \&\ pai(Z, Y)$

C3 $pai(ze, luis)$

C4 $pai(luis, joao)$

Query inicial Q1: $?antepassado(ze, W)$

Q1 resolvido com C1 dá o resolvente Q2:$?pai(ze, W)$

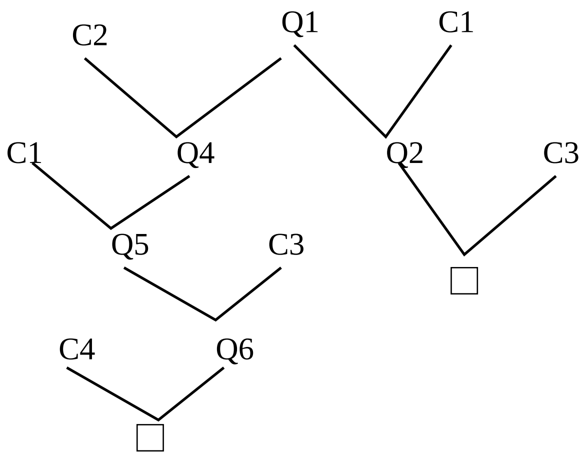
em que $\theta_1 = \{X/ze, Y/W\}$

Q2 resolvido com C3 dá o resolvente Q3:$\Box$

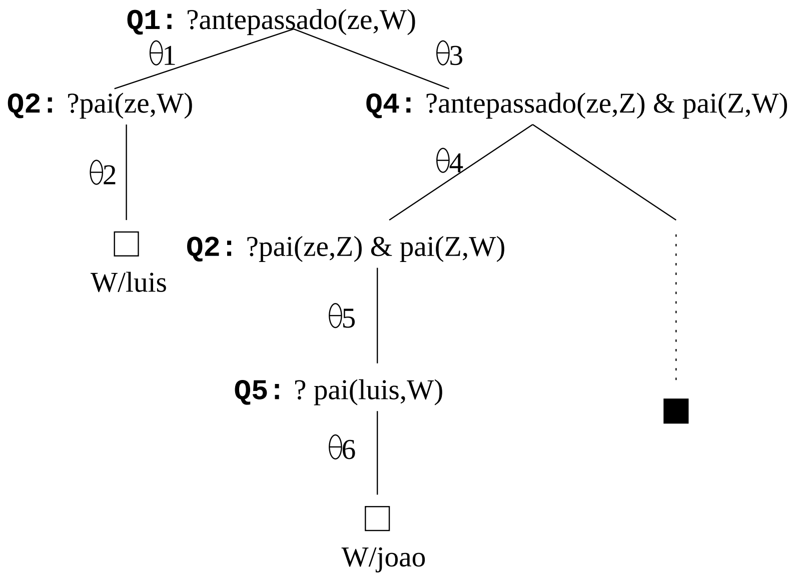
em que $\theta_2 = \{W/luis\}$

Dizemos que Q1 sucede através da refutação $\{Q1, Q2, \Box\}$
Também sucede pela refutação $\{Q1, Q4, Q2, Q5, Q6, \Box\}$.

As várias formas de executar Q1 podem ser mostradas através de um grafo de resolução.



Uma forma mais compacta de representar a execução de Q1 é através de uma arvore de resolução (aquilo a que vamos chamar árvore SLD ou árvore de pesquisa). Aqui, aparecem só os resolventes e os unificadores mais gerais.



em que:

$$\theta_1 = \{X/ze, Y/W\},$$

$$\theta_2 = \{W/luis\},$$

$$\theta_3 = \{X/ze, Y/W\}$$

$$\theta_4 = \{X/ze, Y/W\},$$

$$\theta_5 = \{Z/luis\},$$

$$\theta_6 = \{W/joao\}$$

Computação em Lógica

Computação = Um ramo da árvore, que pode ser:

- com sucesso,
- com insucesso finito e
- com insucesso infinito

Noção de resposta derivada:

é a composição dos unificadores mais gerais gerados durante os passos de resolução dessa computação

isto é

$$\theta^* = \theta_1 \circ \theta_2 \circ \ldots \circ \theta_n$$

A resposta à query $?A_1 \; \& \; \ldots \; \& \; A_n$ é

$$\forall((A_1 \; \& \; \ldots \; \& \; A_n)\theta^*)$$

que é uma consequência lógica do programa

Resolução **SLD**

S para "Selecção"

L para "Linear"

D para "Definite Clauses"(cláusulas de Horn)

O que é importante aqui é explicar o significado do **S**:

Se $?A_1$ & ... & A_n é a query ou qualquer outra cláusula que é usada como progenitora no próximo passo de resolução então SLD-resolução faz uso de uma política de **S**elecção. Ou seja, primeiro selecciona um dos literais A_i.

Se a query não pode ser resolvida nesse literal seleccionado com alguma cláusula do programa, então a computação falha finitamente. Caso contrário a query é resolvida usando o literal seleccionado que nunca mais é usado nessa computação

Este principio de selecção (introduzido por *Kowalski*) faz o espaço de pesquisa mais pequeno do que o que existiria se todos os passos possíveis de resolução fossem permitidos.

Esta ideia de seleccionar um literal de entre os presentes numa query dá origem à noção de *regra de computação* i.e. estratégia de escolha do literal a resolver.

Teorema da independência da regra de computação

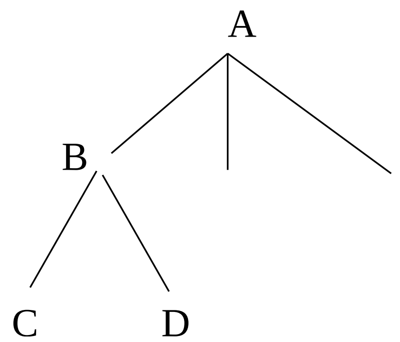
As respostas a uma query são as mesmas independentemente da regra de computação escolhida.

Noção de *estratégia de pesquisa*

Pesquisa Depth-first =

pesquisa sequencial + pesquisa top-down

Quando uma computação acaba, a próxima tem grandes possibilidades de ter nós comuns, na árvore de resolução, com a computação anterior. Estes nós podem ir desde a raiz até um ponto de ramificação. Assim, em vez de começar a próxima computação da raiz é mais eficiente localizar o ponto de ramificação e prosseguir a partir desse ponto. Esta estratégia de pesquisa chama-se *backtracking*.



1ª resolução: A,B,C

2ª resolução sem backtracking: A,B,D

2ª resolução com backtracking: D

O compromisso de usar pesquisa *depth-first* com *backtracking* não determina a ordem relativa na qual as computações são exploradas. Esta ordem é fixada independentemente pela escolha de uma regra de pesquisa (*search rule*).

A regra de pesquisa atribui prioridades às cláusulas do programa que se podem resolver com a query actual.

Exemplo:

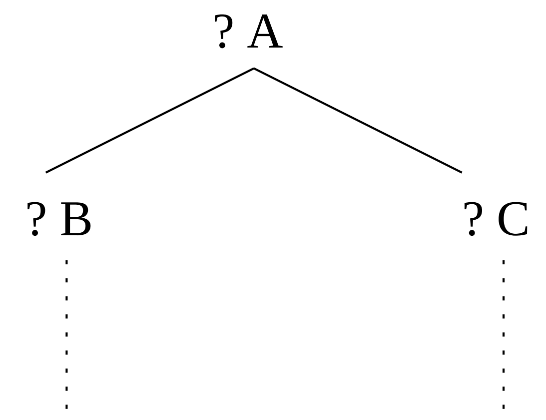
query: $?A$

programa:

C1) $A \leftarrow B$

C2) $A \leftarrow C$

árvore SLD:



Se C1) tem maior prioridade do que C2) então a computação da sub-árvore esquerda é explorada antes da computação da direita. Em PROLOG a prioridade é atribuída de cima para baixo, pela ordem de leitura das cláusulas (exactamente como acontece neste exemplo).

Uma desvantagem associada ao facto da pesquisa ser sequencial é que é *injusta*. Injusta no sentido em que ao explorar uma computação atrasa a exploração de outras computações alternativas. Assim, se a computação corrente é infinita e se outra computação alternativa tem sucesso, então a pesquisa pode não encontrar todas as respostas para a query associada.

Assim, embora a resolução SLD seja um sistema de inferência completo, o procedimento de prova automático obtido da conjugação da SLD com um estratégia de pesquisa injusta é potencialmente incompleto. e.g. PROLOG

No caso de árvores SLD finitas, a regra de pesquisa injusta é a melhor opção.

Exemplo: O seguinte programa

$$p(X, Z) \leftarrow a(X, Z)$$

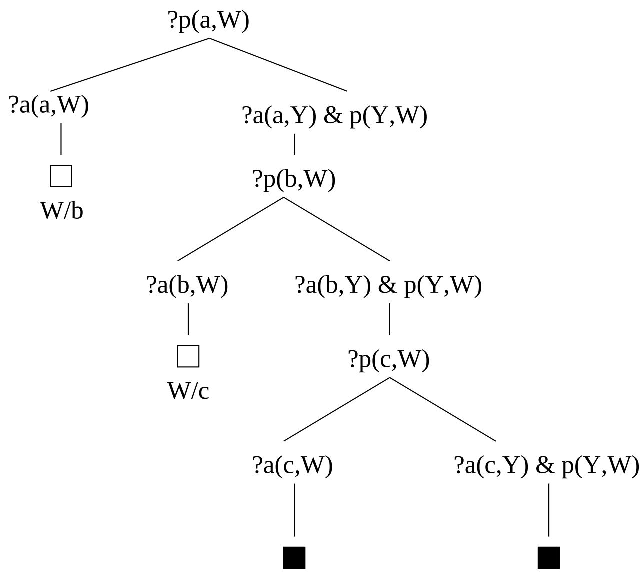
$$p(X, Z) \leftarrow a(X, Y) \;\&\; p(Y, Z)$$

$$a(a, b)$$

$$a(b, c)$$

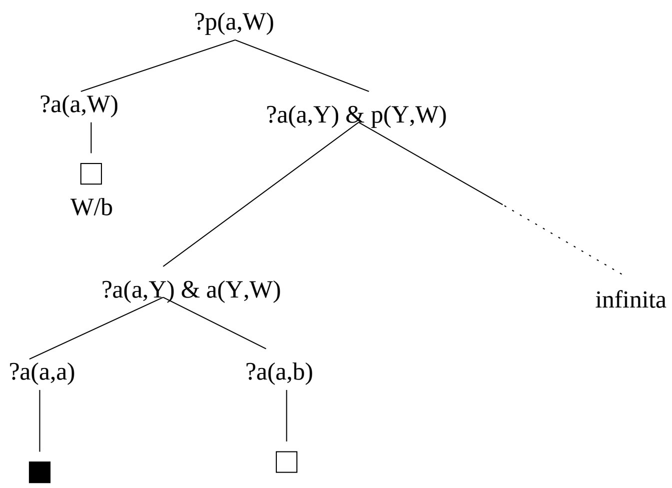
Regra de computação: esquerda-para-direita

Regra de Pesquisa: ordem textual



Regra de computação: direita-para-esquerda

Regra de Pesquisa: ordem textual



Note-se o número de soluções para a interrogação inicial.

Pruning (poda) da pesquisa usando o operador "cut"

Supor que uma interrogação ao predicado A usa uma cláusula da forma:

$$A \leftarrow \textbf{calls} \;\&\; ! \;\&\; \textbf{more\_calls}$$

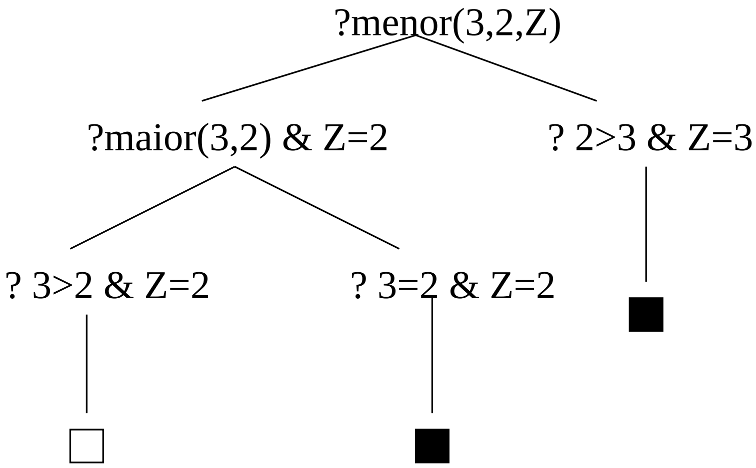
O símbolo "cut"(!) é escalonado como sendo uma normal subquery. No entanto, quando é executado faz a poda (pruning) na árvore de pesquisa do ramo correspondente a:

- todas as formas ainda não tentadas de resolver A
- todas as formas ainda não tentadas de resolver **calls**

Exemplo ("Z é o menor valor entre X e Y"):

$$menor(X, Y, Z) \leftarrow maior(X, Y) \;\&\; Z = Y$$
$$menor(X, Y, Z) \leftarrow Y > X \;\&\; Z = X$$
$$maior(X, Y) \leftarrow X > Y$$
$$maior(X, Y) \leftarrow X = Y$$

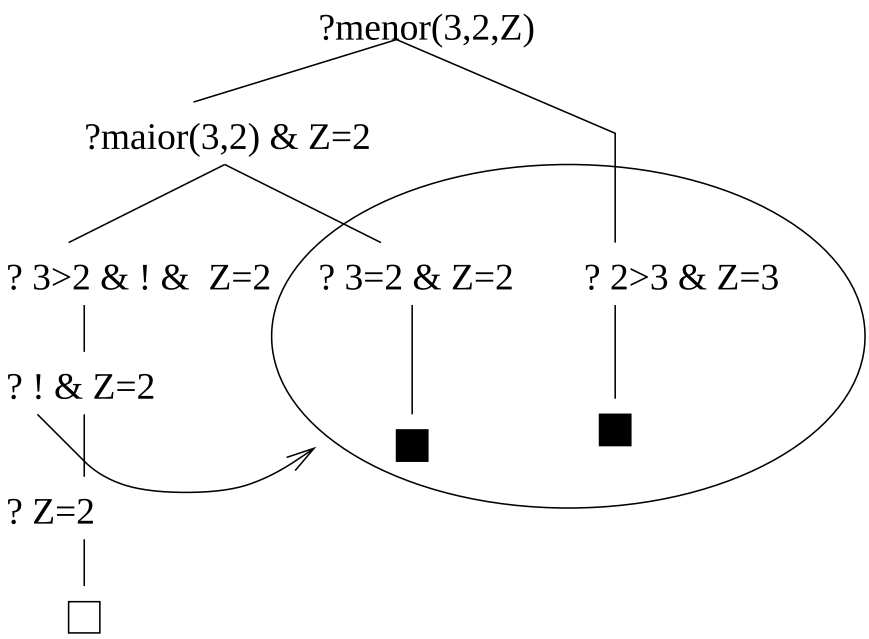
Nota: O programa é correcto para todas as queries mas podem existir computações redundantes



Sabemos que a computação do meio falha pois se $?3 > 2$ pode ser solucionado então $?3 = 2$ não pode ser verdade. A computação mais à direita é também reconhecida que falha porque sabemos que se $?maior(3, 2)$ pode ser solucionado então $?2 > 3$ não pode ser solucionado.

O problema aqui prende-se com o facto do interpretador não estar a par destas exclusões mútuas. Podemos no entanto forçar este facto através do uso de "cuts". Substituindo a primeira cláusula na definição de *menor* pela cláusula:

$$\mathsf{menor}(\mathsf{X}, \mathsf{Y}, \mathsf{Z}) \leftarrow \mathsf{maior}(\mathsf{X}, \mathsf{Y}) \ \& \ ! \ \& \ \mathsf{Z} = \mathsf{Y}$$



O que o “cut” na cláusula anterior previne é o uso de qualquer outra cláusula *menor* e *maior* não tentadas anteriormente.

Isto corresponde a duas computações falhadas (que são podadas do espaço de pesquisa). O novo programa é correcto para todas as possíveis queries, mas mais eficiente.

Exemplos de mau uso do *cut*

Se um programa contendo "cuts" mantém-se *declarativamente* correcto quando se retira os "cuts" então o pior que pode acontecer quando se usa este programa com os seus "cuts" é incompletude não desejada.

Se o programa é *declarativamente* incorrecto então o uso deste programa com os seus "cuts" pode levar à obtenção de respostas incorrectas (unsoundness).

Exemplo:

"Z é o mínimo múltiplo comum entre X e Y"

$\mathsf{mmc(X, Y, Z)} \leftarrow \mathsf{num(Z)}\ \&\ \mathsf{div(X, Z)}\ \&\ \mathsf{div(Y, Z)}\ \&\ !.$

$\mathsf{num(1)}$

$\mathsf{num(Z)} \leftarrow \mathsf{num(W)}\ \&\ \mathsf{Z\ is\ W + 1}$

Para a query

$\mathsf{?mmc(12, 15, Z)}$

este programa calcula a solução correcta $\mathsf{Z/60}$

Para a query

$\mathsf{?mmc(12, 15, 120)}$

o programa calcula a solução <u>errada</u> yes !

Isto deve-se ao facto da primeira cláusula não ser uma correcta definição de mínimo múltiplo comum. Para as queries a mmc cujos primeiros dois argumentos estão instanciados (são ground) a cláusula "trabalha" correctamente e o seu "cut" serve para terminar a execução do programa assim que a primeira solução é encontrada.

Outro Exemplo

"Z é o mínimo entre X e Y"

$\mathsf{minimo(X, Y, X)} \leftarrow \mathsf{X} \leq \mathsf{Y}$ & !.

$\mathsf{minimo(X, Y, Y)}$.

Para a query

?minimo(2, 5, 5)

este programa deriva a solução YES !!!!!

Conclusão

- O operador "cut" serve para *falhar o mais cedo possível*,
- Mas evitar situações de remoção de testes de igualdade
- O "cut" tem uma intrínseca possibilidade de trazer incompletude !!