Negação em Programação em Lógica

- Regra do Mundo fechado
- Negação por Falhanço
- Implementação
- SLD com negação por falhanço (SLDNF)
- Problemas e sistemas correctos

Motivação:

Eficiente manipulação de informação negativa,

Implementação de raciocínio por omissão.

Desejada não monotonia na inferência.

Inferência Monótona: (Se $P \vdash p$ também $P \cup \{q\} \vdash p$)

Há necessidade de fornecer mecanismos de inferência por omissão.

Por exemplo, mecanismos que permitem implementar o seguinte principio:

O que não se conseguir provar assumir como falso

Exemplo da vida real:

Se nada for dito em contrário, hoje há aula!

Regra do Mundo Fechado

Uma noção importante:

A *Base de Herbrand* de um programa P (que designamos por $B(P)$) é o conjunto de todos as fórmulas atómicas *ground* que podemos construir usando os símbolos dos predicados que constam em P.

Exemplo:

$$P = \{p(X) \leftarrow q(X), p(b), q(a)\}$$

$$\text{então } B(P) = \{p(a), q(a), p(b), q(b)\}$$

$B(P)$ é sempre um modelo de P. (na verdade o maior modelo)

Mundo Aberto versus Mundo Fechado

(*Mundo Fechado*): Assumir que aquilo que não observamos ou tomamos conhecimento é falso

(*Mundo Aberto*): Admitir que no nosso mundo existem coisas que não tomamos conhecimento.

Regra do Mundo Fechado (cont)

Regra do Mundo Fechado:

Dada um conjunto de frases lógicas P, a regra do mundo fechado assume que todas as frases atómicas que não são consequências lógicas de P são falsas.

Formalmente, e em termos de mundos completos:

$$CWA(P) = P \cup \{\neg q : q \in B(P), P \not\models q\}$$

em que $B(P)$ é o tudo que é possível construir com o "vocabulário" de P.

Exemplo em que P é um programa em lógica:

$\mathsf{simpatico(X)} \leftarrow \mathsf{ajuda(X, Y)}$

$\mathsf{ajuda(ana, luis)}$

Com a $CWA(P)$ podemos inferir:

$\neg\mathsf{ajuda(ana, ana)}$

$\neg\mathsf{ajuda(luis, luis)}$

$\neg\mathsf{ajuda(luis, ana)}$

$\neg\mathsf{simpatico(luis)}$

Nota: Estas cláusulas negativas são exactamente as interrogações que falham finitamente usando P.

Formalizar a CWA é no fundo tentar formalizar a noção de *inferência por omissão.*

i.e. inferir $\neg q$ por omissão do facto que P implica q

Esperamos ter sempre:

$CWA(P) \models \neg q$ se $P \vdash \neg q$ (pela regra anterior)

Mas podem surgir problemas:

- CWA pode ser inconsistente: e.g. sendo P= $\{A \leftarrow \neg B\}$ então $CWA(P) = \{\neg A, \neg B, A \leftarrow \neg B\}$
  mas $CWA(P) \models \Box$.
- $CWA(P)$ muitas vezes não pode ser simplesmente escrita tendo em conta P. É muitas vezes intratável (indecidibilidade da lógica).

Negação por Falhanço (Finito)

Em vez de implementar a regra de inferência CWA num programa podemos adicionar um novo "operador" à nossa linguagem:

not (negação por falhanço).

Regra do falhanço finito:

a query ?not q sucede
se e só se
a sua subquery ?q falha finitamente

Adicionando este novo "operador" temos de utilizar uma nova semântica (Completion) para explicar as consequências lógicas de um programa. Assim, interpretamos todas as ocorrências do novo operador como negação clássica e passamos a usar a nova semântica para determinar as consequências lógicas de um programa P.

not em Prolog pode ser implementado como:

```
not(P):- P, !, fail.
not(P).
```

Este novo operador fornece não monotonia:

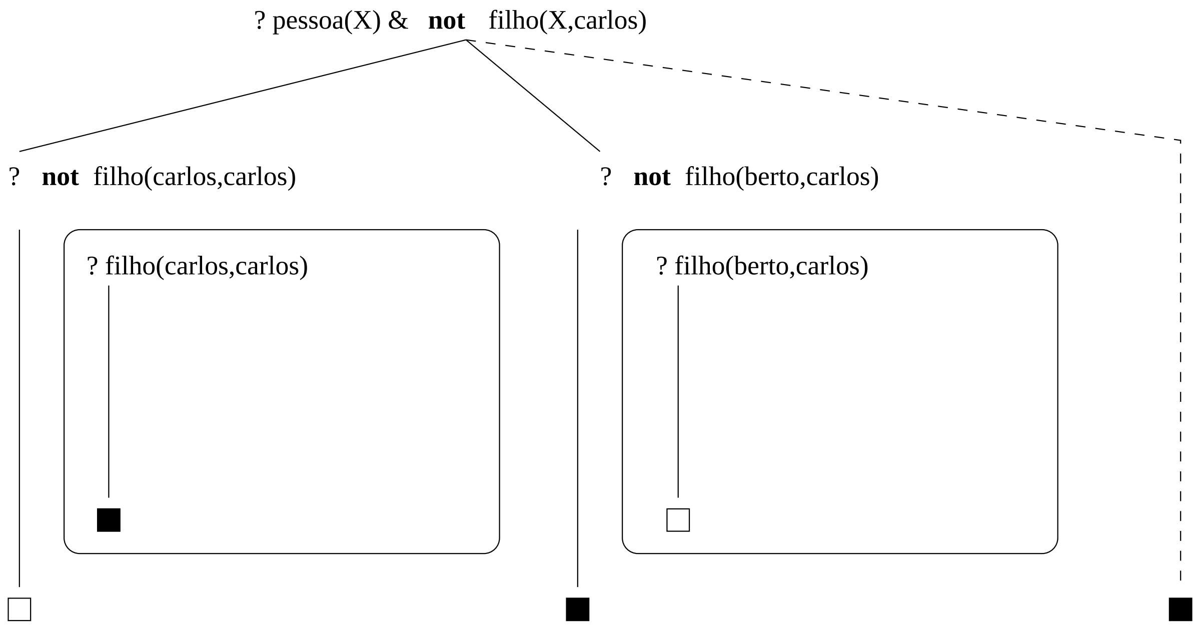
e.g. $P = \{p \leftarrow not\ q\}$

$P \vdash p$ mas $P \cup \{q\} \not\vdash p$

Exemplo de uma árvore de prova com subcomputações not

pessoa(carlos) filho(berto, carlos)
pessoa(berto) filho(emilio, carlos)
pessoa(emilio)

Exemplo de computação SLD com falhanço finito para o programa anterior



Neste exemplo, tudo foi calculado com o mecanismo standard SLD (com a computação especial para o operador not). A sub-árvore mais à esquerda gerada pela sub-query com not é de uma computação de falhanço finito, resultando numa subcomputação com sucesso. As outras duas sub-árvores contêm computações com sucesso resultando em duas subcomputações de falhanço finito.

O operador not só pode ser usado para inferir átomos *ground* i.e. sem variáveis, caso contrário a inferência pode ser incompleta e/ou *incorrecta.*

Exemplos, em que $P = \{$

$$\mathsf{filho(joao, luis)}$$

$$\mathsf{filho(rui, luis)}\ \}$$

A interrogação

$$\mathsf{?not\ filho(X, luis)}$$

dá origem à subquery

$$\mathsf{?filho(X, luis).}$$

Duas possíveis respostas existem para esta subquery, nomeadamente X/joao e X/rui. Assim, a interrogação inicial falha finitamente. No entanto existe um valor para X (X/luis) em que a subquery falha finitamente o que implica que a query inicial deve suceder.

Este sistema de inferência é *incompleto* .

Supor que adicionamos a P as seguintes cláusulas:

$$\mathsf{pai(Y)} \leftarrow \mathsf{not\ esteril(Y)}$$

$$\mathsf{esteril(Y)} \leftarrow \mathsf{not\ filho(X, Y)}$$

A query

$$\mathsf{?pai(Y)}$$

executada usando a regra do falhanço finito sucede, mas sem nenhuma substituição para o Y (not não retorna substituições). Mas pela leitura das cláusulas (e aplicando a regra do mundo fechado) existe um $\mathsf{pai(Y)}$ se e só se não existir um X tal que $\mathsf{filho(X, Y)}$ não for verdade.

Mas o que acontece é que existe um X ($\mathsf{X} = \mathsf{luis}$) o que implica que $\mathsf{pai(Y)}$ não pode ser verdade.

Assim o sistema de inferência é *incorrecto.*

Nota: Normalmente lê-se

$$\mathsf{p(X)} \leftarrow \mathsf{not\ q(Y)}$$

como

$$\forall \mathsf{X}(\mathsf{p(X)} \leftarrow \exists \mathsf{Y}\mathsf{not\ q(Y)})$$

em vez de

$$\forall \mathsf{X}(\mathsf{p(X)} \leftarrow \mathsf{not}\ \exists \mathsf{Y}\mathsf{q(Y)}) \qquad \text{(negação existencial)}$$

O procedimento SLDNF

SLD + Negação por falhanço finito

- Restrição da regra de computação seleccionar subqueries com not que estejam totalmente instanciadas ("ground")
- Para além disso, é necessário forçar que a regra de computação não seleccione outra subquery até que a interrogação com not seja totalmente calculada.
- Dizemos que uma regra de computação que respeita as duas condições acima é *safe*.
- No entanto há casos que a aplicação de uma regra de computação safe leva a que nenhuma call (subquery) possa ser seleccionada. Dizemos que a computação bloqueou (*floundered*) e não pode ser explorada mais longe.
- Assim, uma regra de computação safe garante-nos que o nosso sistema é solido e correcto (*sound*) mas a possibilidade de uma computação bloquear implica que o nosso sistema é inevitavelmente incompleto!

A linguagem Prolog como uma implementação de um sistema de Programação em Lógica

- Linguagem: cláusulas de Horn com operador de negação por falhanço
- Resolução SLD
- Regra de computação: da esquerda para a direita
- Regra de selecção: de cima para baixo
- Estratégia de pesquisa: sequencial top-down com *backtracking*
- Unificação sem o teste de *occurs check*
- Negação por falhanço sem regra de computação *safe*.
- Inclui operador de pruning (cut)