

IB015 Neimperativní programování

Řezy, vstup-výstup, všechna řešení

Jiří Barnat

Řez

Pozorování

- Základem výpočtu logického programu je **backtracking**.
- Některé větve výpočtu nevedou k požadovanému cíli.
- Jistá kontrola nad způsobem prohledávání SLD stromu, by byla vhodná.

Dosavadní možnosti ovlivnění výpočtu

- Změna pořadí faktů v databázi.
- Změna pořadí podcílů v definici pravidla.

Operátor řezu – !/0

- Vždy jako podcíl úspěje.
- Ovlivňuje způsob výpočtu (má vedlejší efekt).
- Eliminuje další volby, které by Prolog udělal při procházení výpočetního stromu, a to od okamžiku unifikace podcíle s levou stranou pravidla, ve kterém se predikát ! vyskytuje, až do místa výskytu !.

Důsledky vedlejšího efektu

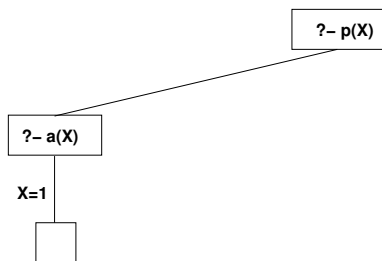
- Prořezává výpočetní strom.
- Rychlejší výpočet.
- Riziko odřezání větví výpočtu, které vedou k dalším (stejným, či jiným) řešením.

Příklad fungování řezu – bez řezu

?- p(X)

```
p(X) :- a(X).  
p(X) :- b(X), c(X),  
        d(X), e(X).  
p(X) :- f(X).  
  
a(1).  
b(1). c(1).  
d(1). d(2). e(2).  
f(3).  
b(2). c(2).
```

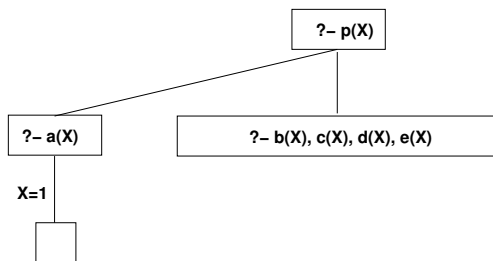
Příklad fungování řezu – bez řezu



```
p(X) :- a(X).  
p(X) :- b(X), c(X),  
         d(X), e(X).  
p(X) :- f(X).
```

```
a(1).  
b(1). c(1).  
d(1). d(2). e(2).  
f(3).  
b(2). c(2).
```

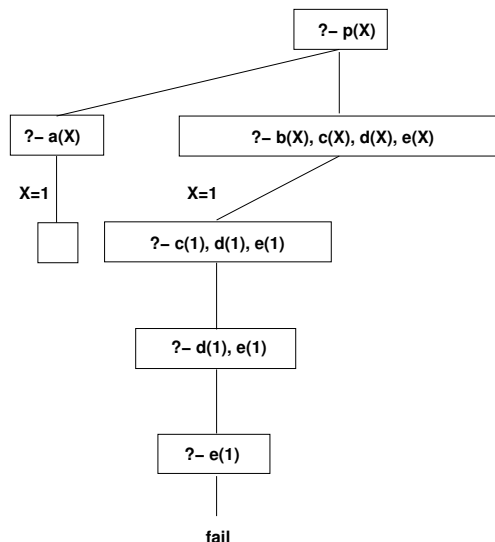
Příklad fungování řezu – bez řezu



```
p(X) :- a(X).  
p(X) :- b(X), c(X),  
         d(X), e(X).  
p(X) :- f(X).
```

```
a(1).  
b(1). c(1).  
d(1). d(2). e(2).  
f(3).  
b(2). c(2).
```

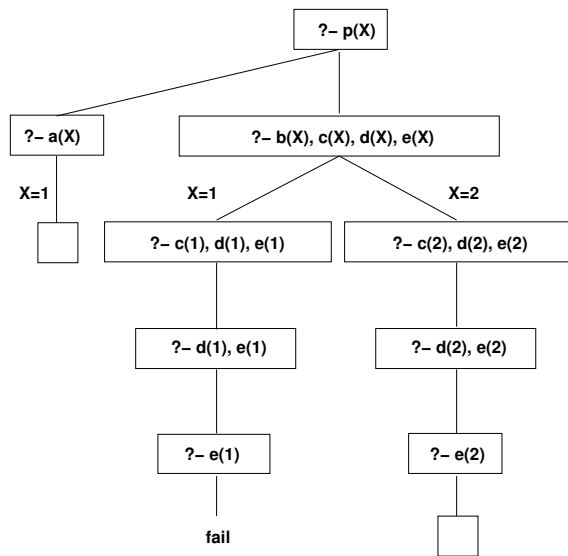
Příklad fungování řezu – bez řezu



```
p(X) :- a(X).
p(X) :- b(X), c(X),
        d(X), e(X).
p(X) :- f(X).

a(1).
b(1). c(1).
d(1). d(2). e(2).
f(3).
b(2). c(2).
```

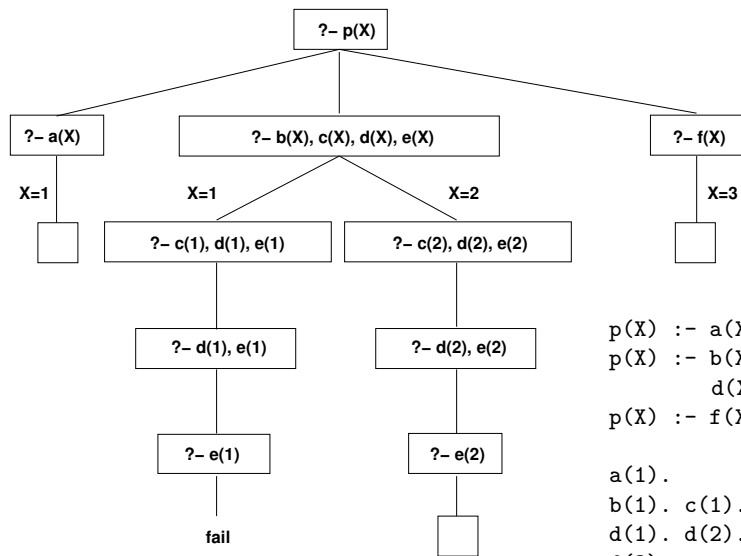

Příklad fungování řezu – bez řezu



```
p(X) :- a(X).
p(X) :- b(X), c(X),
        d(X), e(X).
p(X) :- f(X).

a(1).
b(1). c(1).
d(1). d(2). e(2).
f(3).
b(2). c(2).
```

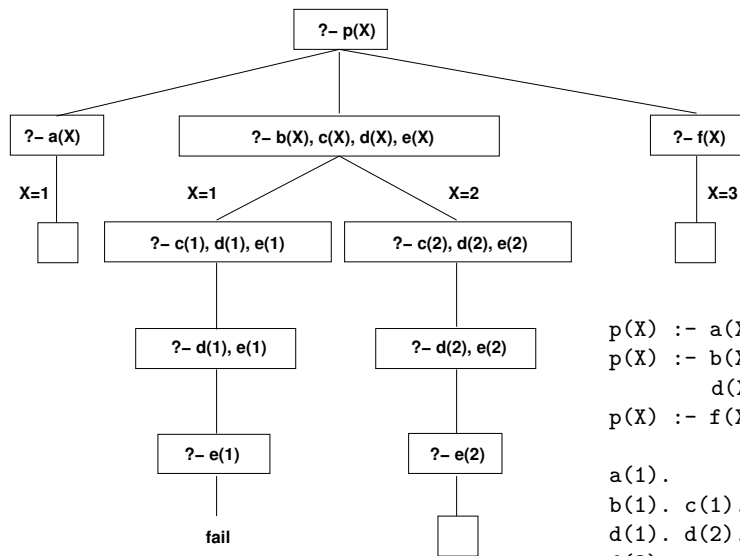
Příklad fungování řezu – bez řezu



`p(X) :- a(X).`
`p(X) :- b(X), c(X),`
`d(X), e(X).`
`p(X) :- f(X).`

`a(1).`
`b(1). c(1).`
`d(1). d(2). e(2).`
`f(3).`
`b(2). c(2).`

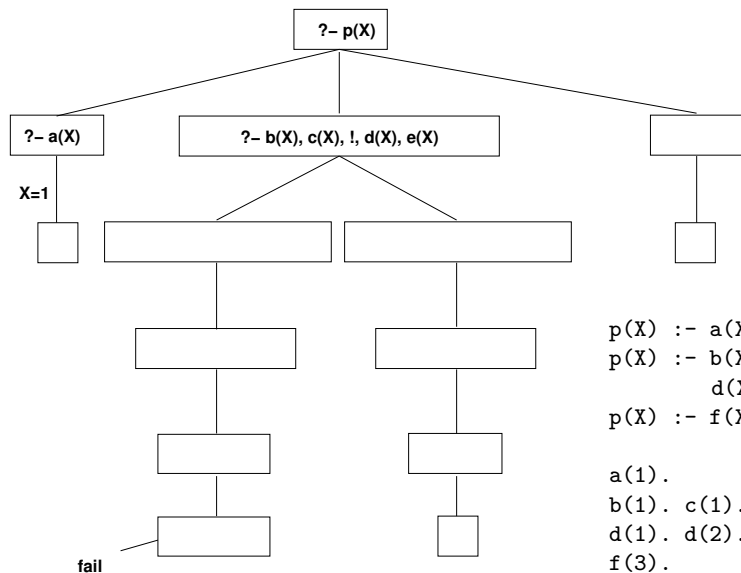
Příklad fungování řezu – s řezem



```
p(X) :- a(X).
p(X) :- b(X), c(X), !,
        d(X), e(X).
p(X) :- f(X).

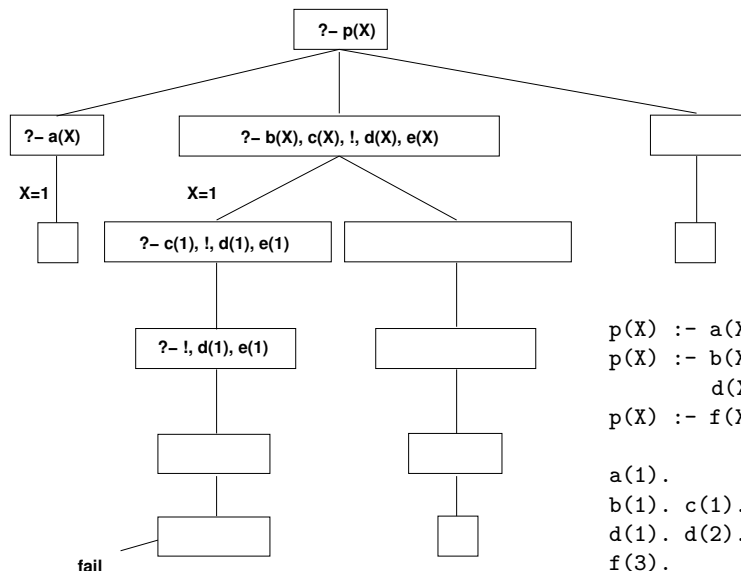
a(1).
b(1). c(1).
d(1). d(2). e(2).
f(3).
b(2). c(2).
```

Příklad fungování řezu – s řezem



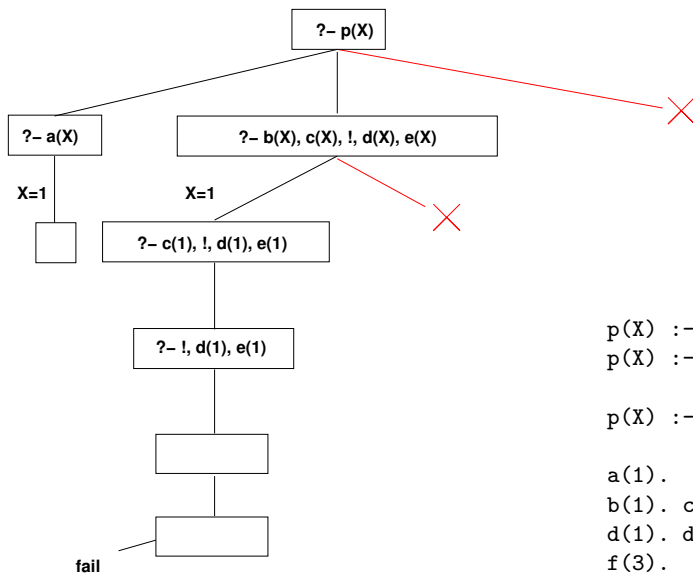
```
p(X) :- a(X).  
p(X) :- b(X), c(X), !,  
         d(X), e(X).  
p(X) :- f(X).  
  
a(1).  
b(1). c(1).  
d(1). d(2). e(2).  
f(3).  
b(2). c(2).
```

Příklad fungování řezu – s řezem



```
p(X) :- a(X).  
p(X) :- b(X), c(X), !,  
        d(X), e(X).  
p(X) :- f(X).  
  
a(1).  
b(1). c(1).  
d(1). d(2). e(2).  
f(3).  
b(2). c(2).
```

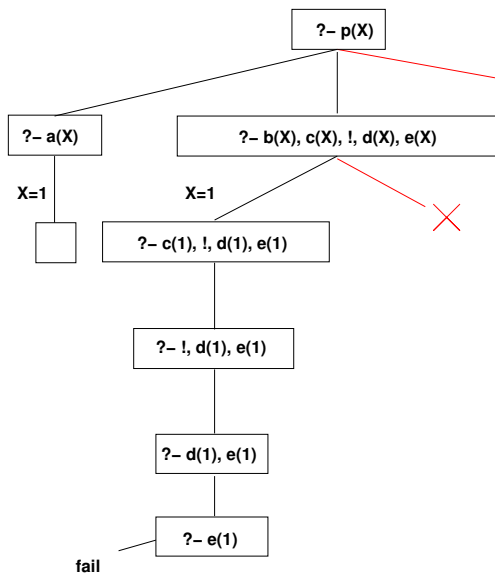
Příklad fungování řezu – s řezem



```
p(X) :- a(X).
p(X) :- b(X), c(X), !,
        d(X), e(X).
p(X) :- f(X).

a(1).
b(1). c(1).
d(1). d(2). e(2).
f(3).
b(2). c(2).
```

Příklad fungování řezu – s řezem



```
p(X) :- a(X).
p(X) :- b(X), c(X), !,
        d(X), e(X).
p(X) :- f(X).

a(1).
b(1). c(1).
d(1). d(2). e(2).
f(3).
b(2). c(2).
```

Popis

- Pokud se při řešení podcíle narazí v těle pravidla na operátor `!`, ostatní fakta a pravidla, se pro právě řešený cíl (ten, který se unifikoval s hlavou pravidla) neberou v potaz.

Příklad

- Porovnej chování následujících programů.

a) `a(X) :- X = 1.`

`a(X) :- X = 2.`

`?- a(X).`

`X = 1 ;`

`X = 2.`

b) `a(X) :- X = 1, !.`

`a(X) :- X = 2.`

`?- a(X).`

`X = 1.`

Popis

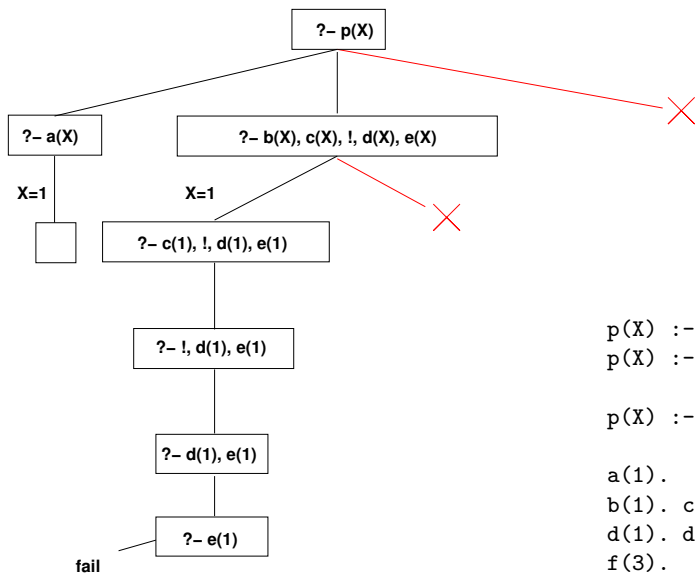
- Pokud se při řešení podcíle narazí v těle pravidla na operátor řezu, všechny unifikace vyplývající z podcílů vyskytujících se v těle pravidla před operátorem ! se fixují (jiné možnosti unifikace těchto podcílů se neuvažují).

Porovnejte

a) $a(X) :- X = 0.$
 $a(X) :- X = 1.$
 $b(X,Y) :- a(X), a(Y).$
 $?- b(X,Y).$
 $X = 0, Y = 0 ;$
 $X = 0, Y = 1 ;$
 $X = 1, Y = 0 ;$
 $X = 1, Y = 1.$

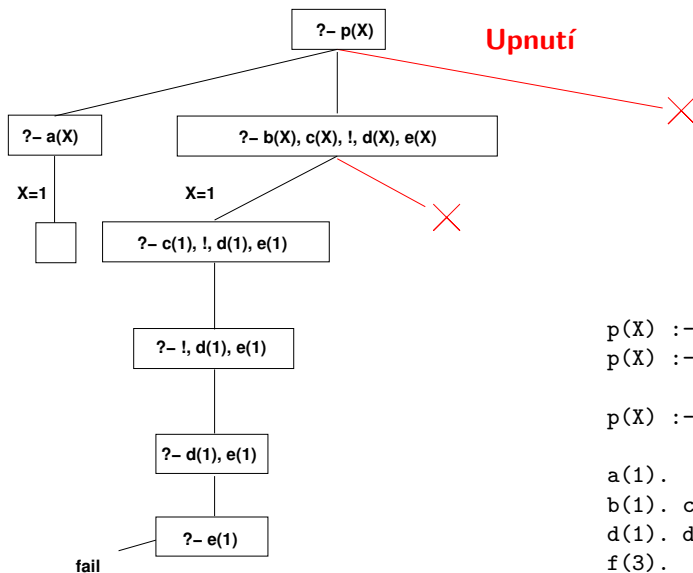
b) $a(X) :- X = 0.$
 $a(X) :- X = 1.$
 $b(X,Y) :- a(X), !, a(Y).$
 $?- b(X,Y).$
 $X = 0, Y = 0 ;$
 $X = 0, Y = 1.$

Příklad fungování řezu – vedlejší efekty



```
p(X) :- a(X).  
p(X) :- b(X), c(X), !,  
        d(X), e(X).  
p(X) :- f(X).  
  
a(1).  
b(1). c(1).  
d(1). d(2). e(2).  
f(3).  
b(2). c(2).
```

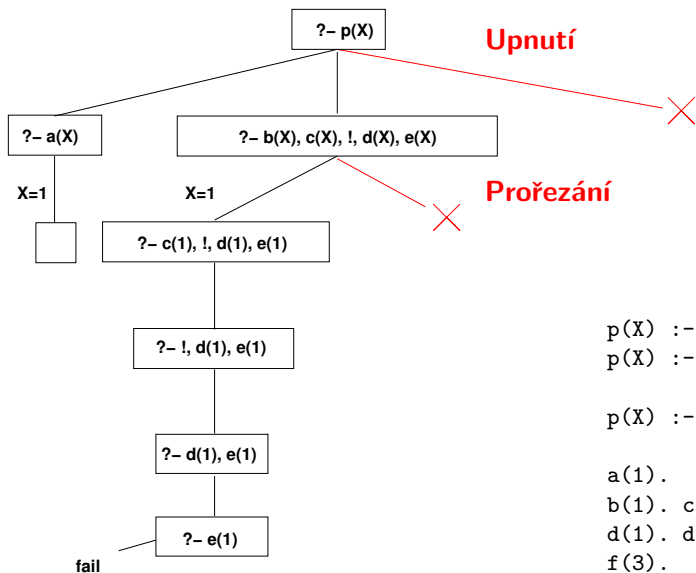
Příklad fungování řezu – vedlejší efekty



```
p(X) :- a(X).
p(X) :- b(X), c(X), !,
        d(X), e(X).
p(X) :- f(X).

a(1).
b(1). c(1).
d(1). d(2). e(2).
f(3).
b(2). c(2).
```

Příklad fungování řezu – vedlejší efekty



```
p(X) :- a(X).  
p(X) :- b(X), c(X), !,  
        d(X), e(X).  
p(X) :- f(X).  
  
a(1).  
b(1). c(1).  
d(1). d(2). e(2).  
f(3).  
b(2). c(2).
```

Úkol

- Určete maximální možný výstup interpretru pro následující kód v Prologu na dotaz `?- b(X,Y)`.

Zadání 1

- `a(X) :- X = 0.`
`a(X) :- X = 1, !.`
`a(X) :- X = 2.`
`b(X,Y) :- a(X), a(Y).`

Úkol

- Určete maximální možný výstup interpretru pro následující kód v Prologu na dotaz `?- b(X,Y)`.

Zadání 1

- `a(X) :- X = 0.`
`a(X) :- X = 1, !.`
`a(X) :- X = 2.`
`b(X,Y) :- a(X), a(Y).`

Řešení 1

- `X = 0, Y = 0 ;`
`X = 0, Y = 1 ;`
`X = 1, Y = 0 ;`
`X = 1, Y = 1.`

Úkol

- Určete maximální možný výstup interpretru pro následující kód v Prologu na dotaz `?- b(X,Y)`.

Zadání 1

- `a(X) :- X = 0.`
`a(X) :- X = 1, !.`
`a(X) :- X = 2.`
`b(X,Y) :- a(X), a(Y).`

Řešení 1

- `X = 0, Y = 0 ;`
`X = 0, Y = 1 ;`
`X = 1, Y = 0 ;`
`X = 1, Y = 1.`

Zadání 2

- `a(X) :- X = 0.`
`a(X) :- X = 1, !.`
`a(X) :- X = 2.`
`b(X,Y) :- a(X), !, a(Y).`

Úkol

- Určete maximální možný výstup interpretru pro následující kód v Prologu na dotaz `?- b(X,Y)`.

Zadání 1

- `a(X) :- X = 0.`
`a(X) :- X = 1, !.`
`a(X) :- X = 2.`
`b(X,Y) :- a(X), a(Y).`

Řešení 1

- `X = 0, Y = 0 ;`
`X = 0, Y = 1 ;`
`X = 1, Y = 0 ;`
`X = 1, Y = 1.`

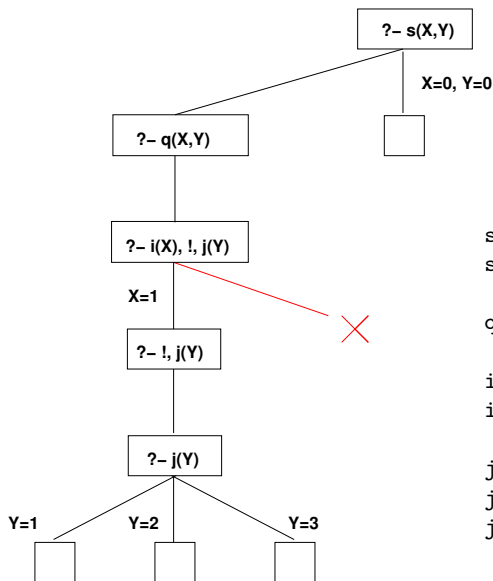
Zadání 2

- `a(X) :- X = 0.`
`a(X) :- X = 1, !.`
`a(X) :- X = 2.`
`b(X,Y) :- a(X), !, a(Y).`

Řešení 2

- `X = 0, Y = 0 ;`
`X = 0, Y = 1.`

Příklad fungování řezu – imunita nadřazených cílů



`s(X,Y) :- q(X,Y) .`

`s(0,0) .`

`q(X,Y) :- i(X), !, j(Y) .`

`i(1) .`

`i(2) .`

`j(1) .`

`j(2) .`

`j(3) .`

Predikát max/3

- Uvažme predikát $\text{max}(+N1, +N2, ?\text{Max})$, který se pro číselné argumenty vyhodnotí na pravda, pokud třetí číslo je maximem prvních dvou.

Řešení bez použití řezu

- $\text{max}(X, Y, Y) :- X \leq Y.$
 $\text{max}(X, Y, X) :- X > Y.$

Pozorování

- Neefektivita v případě dotazu:

```
?- max(3,4,X).  
X = 4 ;      /* následuje úplně zbytečný výpočet */  
false.
```
- Uvedené klauzule jsou vzájemně vylučné, pokud výpočet na jedné uspěje, vyhodnocovat druhou klauzuli je zcela zbytečné.

Řešení s použitím řezu

- $\max(X,Y,Y) :- X \leq Y, !.$
 $\max(X,Y,X) :- X > Y.$
- Korektní řešení, nerealizuje nadbytečný výpočet při rekurzivním prohledávání stromu (díky upnutí).

Pozorování a otázka

- Pokud $x > y$, dochází ke dvěma aritmetickým porovnáním.
- Podmínky jsou vzájemně vylučné.
- Je test $x > y$ v těle druhého pravidla vůbec nutný?

Odpověď

- V módu $(+, +, -)$ je test nadbytečný.
- V módu $(+, +, +)$ je test nutný, kdyby v klauzuli nebyl, tak:
?- $\max(2,3,2).$
true. **/* CHYBA */**

Efektivnější, ale nesprávné řešení

- $\max(X, Y, Y) :- X \leq Y, !.$
 $\max(X, Y, X).$
- Problém: cíl $\max(2, 3, 2)$ se neunifikuje s hlavou prvního pravidla, přestože platí podmínka $2 \leq 3$.

Opravené správné řešení

- $\max(X, Y, Z) :- X \leq Y, !, Z = Y.$
 $\max(X, Y, X).$
- K unifikaci s hlavou prvního pravidla dojde vždy, podmínka je vždy vyhodnocena, a je-li třeba, dojde k upnutí.
- Funguje korektně v módu $(+, +, ?)$.
- Vždy provede pouze jedno aritmetické porovnání.

Zelené řезы

- Odstraněním operátoru řezu se nemění sémantika programu (množina řešení po odstranění řezu je shodná).
- Řez je použit pouze z důvodů efektivity.
- Někdy se jako „modré“ označují řезы eliminující duplicity.

Červené řезы

- Odstraněním operátoru řezu se mění sémantika programu (po odstranění řezu, je možné nalézt další jiná řešení).

Obecná doporučovaná strategie

- Vyrobit funkční řešení bez řезů.
- Zvýšit efektivitu použitím „zelených“ řезů.
- Využít „červené řезы“ pouze pokud není vyhnutí, dobře okomentovat.

Negace v Prologu

Predikát fail/0

- Vestavěný predikát, nikdy neuspěje.
- Pokus o dokazování fail způsobí „backtracking“ ve výpočtu.

Pozorování/Připomenutí

- Pokud všechny větve výpočetního stromu skončí neúspěchem, interpreter ohlásí false, tj. že požadovaný cíl nelze dokázat.

Vysvětlete

- `?- fail.`
`false.`

- `a(_) :- fail.`
`a(_).`
`?- a(cokoliv).`
`true.`

- `a(_) :- !, fail.`
`a(_).`
`?- a(cokoliv).`
`false.`

Kombinace fail a upnutí

- V kombinaci s řezy, konkrétně mechanismem upnutí, může predikát `fail` sloužit jako negace.

Příklad

- V Prologu запиšte: „Hezké je vše, co není škaredé.“

```
hezke(X) :- skarede(X), !, fail.  
hezke(_).  
skarede(strasidlo).
```

- Pokud je možné dokázat podcíl `skarede(X)`, pak predikát `hezke(X)` pro totéž `X` se vyhodnotí na `false`.

```
?- hezke(strasidlo).           ?- hezke(cokoliv_jineho).  
false.                        true.
```


Význam predikátu $\backslash+/1$

- Pokud $\exists(x_1, \dots, x_n)$ takové, že $P(x_1, \dots, x_n)$ je dokazatelné, pak

$?- \backslash+P(x_1, \dots, x_n)$

false.

Definice predikátu $\backslash+/1$

- Definován následujícími pravidly:

$\backslash+(P) :- P, !, fail.$

$\backslash+(_).$

- Známa jako „Negation as failure.“

Pozorování

- Při aplikaci na cíl s proměnnou, je negace vůči faktu, zda pro původní cíl existuje splňující přiřazení.

Neintuitivní chování – neodpovídá logické negaci

- `barva(cervena).`
`barva(modra).`

```
?- X=zelená, \+barva(X).  
X = zelená.
```

```
?- \+barva(X), X=zelená.  
false.
```

Doporučení

- Operátor `\+` používat pouze na podcíle s plně instanciovanými argumenty.

Pozorování

- Negace aplikovaná na termy s volnou proměnnou je nebezpečná zejména pokud se vyskytuje jako podcíl na pravé straně pravidla pro jiný term.

Příklad

- Uvažme následující program:
barva(cervena).
barva(modra).
foo(X) :- \+barva(X).
- Zajímá nás, zda existuje x takové, že platí foo(X), tedy:
?- foo(X).
false.
- Logický závěr by mohl být, že takové X neexistuje, ale:
?- foo(fialova).
true.

If ->Then; Else

- Definováno následovně:
(If -> Then; Else) :- If, !, Then.
(If -> Then; Else) :- !, Else.
- Pokud není větev Else chová se jako:
If -> Then; fail.

Příklad použití podmínky

- `min(X,Y,Z) :- X < Y -> Z = X ; Z = Y.`

Vstup, Výstup

Proud (Stream)

- Místo, odkud program může číst, nebo kam může program zapisovat posloupnost znaků.
- Proudy realizují čtení z klávesnice, výpisy na obrazovku, čtení a zápis do souborů.

Předdefinované proudy `user_*`

- `user_input`, `user_output`, `user_error`
- Pro přímou interakci s uživatelem.
- Iniciálně svázány s proudy `stdin`, `stdout` a `stderr`.

Předdefinované proudy `current_*`

(SWI-Prolog)

- `current_input`, `current_output`
- Iniciálně svázány s odpovídajícími proudy `user_*`.
- Definují místo čtení a zápisu pro predikáty, které neberou konkrétní proud jako svůj argument.

Poznámka

- V Prologu se ustálily dvě sady funkcí pro manipulaci se vstup-výstupními proudy.
- SWI Prolog podporuje oba módy a umí mezi nimi přepínat.

Edinburghský styl

- `tell/1`, `see/1`, ...
- Jednoduché rozhraní, snadné použití.

ISO standard

- `open/3`, `close/1`, ...
- Pro komplexní použití.

see(+SrcDest)

- Otevře `SrcDest` pro čtení a nastaví aktuální vstupní proud.

tell(+SrcDest)

- Otevře `SrcDest` pro zápis a nastaví aktuální výstupní proud.

append(+File)

- Jako `tell/2` ale nastaví pozici místa zápisu na konec souboru.

seeing(-Stream)/telling(-Stream)

- Vrací aktuálně používané proudy pro čtení/zápis.

seen a told

- Uzavírá aktuální vstupní resp. výstupní proud.

Forma čteného/zapisovaného znaku

- `byte` – číslo 0 až 255.
- `char` – znak.
- `code` – ASCII kód znaku.

`put_char(+Char)`

`put_char(+Stream, +Char)`

- Realizuje zápis znaku do aktuálního resp. zadaného proudu.
- Podobně `put_byte` a `put_code`.

Predikáty

- `n1` – zapíše znak nového řádku.
- `get_*` – načte znak v dané formě.
- `peek_*` – znak čekající na přečtení v dané formě.
- `tab(+A)` – zapíše A mezer.
- `flush_output` – vyprázdní buffer operačního systému.

read(-Term)

- Přečte vstup až do další tečky, a přečtené se pokusí unifikovat s argumentem Term.
- Při čtení z konce souboru vrací atom `end_of_file`.

Příklad

- ```
?- read(name:N), read(adresa:[X,Y,Z]).
|: name: jirik. adresa:['u shnile tresne', 42, atlantida].
N = jirik,
X = 'u shnile tresne',
Y = 42,
Z = atlantida.
```

**write(+Term)**

**write(+Stream, +Term)**

- Zápís termu do aktuálního/zadaného výstupního proudu.

**writeln(+Term)**

- Ekvivalentní zápisu `write(Term), nl.`

**read\_term(-Term, +Options)**

**write\_term(+Term, +Options)**

- Komplexní čtení zápis, viz dokumentace.

## repeat/0

- Vždy uspěje, vytváří neomezený počet větvení výpočetního stromu pro „backtrackování“.
- repeat.  
repeat :- repeat.

## Použití repeat

- Typickým použitím predikátu je zpracování vstupů.

```
Head :- repeat,
 ctizeVstupu(X),
 zpracujVstup(X),
 jeKonecVstupu(X), /* X == end_of_file. */
 !.
```

- Mimo toto použití se v podstatě nevyskytuje.

## Seznamy všech řešení

## Pozorování

- Dotazem s volnou proměnnou instruujeme Prolog, aby našel jedno vyhovující přiřazení volným proměnným.
- Uživatel může vynutit systematické hledání dalších řešení.
- Prolog ale umí vrátit seznam všech řešení najednou.

## bagof(+Template, :Goal, -Bag)

- Vrací seznam Bag všech alternativ unifikovaných s Template vyhovujících cíli Goal.
- Vrací false pokud Goal nemá řešení.

## Jednoduchý příklad

- `bagof(X, barva(X), Barvy).`  
Barvy = [modra, cervena].

## Databáze

- `slevy(albert,mleko,leden).`  
`slevy(albert,mleko,unor).`  
`slevy(billa,cukr,cerven).`  
`slevy(billa,cukr,prosinec).`  
`slevy(tesco,cukr,duben).`

## Dotazy

- `?- bagof(Z,slevy(X,Y,Z),R).`  
`X = albert, Y = mleko, R = [leden, unor] ;`  
`X = billa, Y = cukr, R = [cerven, prosinec] ;`  
`X = tesco, Y = cukr, R = [duben].`
- `?- bagof(Z,slevy(_,_ ,Z),R).`  
`R = [leden, unor] ;`  
`R = [cerven, prosinec] ;`  
`R = [duben].`

## Pozorování

- Při použití `bagof` různé hodnoty proměnných, které nejsou součástí výsledného seznamu, vedou na různé varianty výsledku.
- Pro sloučení těchto variant nestačí použít anonymní proměnnou.

## Existenční kvantifikace

- Zápisem  $\text{var}^{\wedge}$  před cíl vyjádříme, že různé hodnoty v této proměnné se nemají rozlišovat, stačí že existuje nějaké vyhovující přiřazení.
- Je možné takto kvantifikovat více proměnných:

$$X^{\wedge}Y^{\wedge}Ci1(X,Y,Z)$$

$$[X,Y]^{\wedge}Ci1(X,Y,Z)$$



## Databáze

- `slevy(albert,mleko,leden).`  
`slevy(albert,mleko,unor).`  
`slevy(billa,cukr,cerven).`  
`slevy(billa,cukr,prosinec).`  
`slevy(tesco,cukr,duben).`

## Existenčně kvantifikované dotazy

- `?- bagof(Z,X^slevy(X,Y,Z),R).`  
`Y = cukr, R = [cerven, prosinec, duben] ;`  
`Y = mleko, R = [leden, unor].`
- `?- bagof(Z,Y^X^slevy(X,Y,Z),R).`  
`R = [leden, unor, červen, prosinec, duben].`
- `?- bagof(Z,[X,Y]^slevy(X,Y,Z),R).`  
`R = [leden, unor, červen, prosinec, duben].`

## **findall(+Template, :Goal, -Bag)**

- Seznam všech vyhovujících řešení.
- V případě že `Goal` nemá řešení vrací prázdný seznam.
- Jinak funguje stejně jako `bagof/3` s tím, že všechny volné proměnné jsou existenčně kvantifikovány.

## **setof(+Template, :Goal, -Set)**

- Využívá predikát `bagof/3`, ale výsledek seřadí s použitím predikátu `sort/2`. Výsledek je tedy seřazený seznam všech možných řešení, s tím že každé řešení je uvedeno pouze jednou (duplicitní řešení jsou odstraněna).