

Řezy, akumulátory, tail-rekurze

IB015 Neimperativní programování

Tomáš Szaniszlo, Vladimír Štill, Martin Ukrop

Fakulta informatiky, Masarykova univerzita

Podzim 2017

Operátor řezu: !

- podcíl, který vždy uspěje
- eliminuje další volby ve výpočetním stromu: od okamžiku unifikace hlavy klauzule po výskyt !

```
1 ?- member(X, [1,2,3]).
```

```
2 X = 1 ;
```

```
3 X = 2 ;
```

```
4 X = 3.
```

```
1 ?- member(X, [1,2,3]), ! .
```

```
2 X = 1.
```

```
3 % zadna dalsi reseni
```

Použití:

- zefektivnění výpočtu
- omezení duplicitních řešení
- negace

Operátor řezu: příklad

Odstranění zbytečné možnosti dotazu na další řešení:

```
1 powertwo(1) :- ! .  
2 powertwo(X) :- X mod 2 == 0,  
3               Y is X div 2,  
4               powertwo(Y).
```

Efekty řezu na výpočetní strom

Upnutí: Další pravidla a fakta pro aktuální cíl se ignorují.

Ořezání: Fixování voleb provedených při vyhodnocování těla pravidla až po operátor řezu.

1 $f(1).$

2 $f(6).$

3 $g(X) :- f(X), !, X > 5.$

4 $g(X) :- X =< 2.$

$?- g(X).$

Efekty řezu na výpočetní strom

Upnutí: Další pravidla a fakta pro aktuální cíl se ignorují.

Ořezání: Fixování voleb provedených při vyhodnocování těla pravidla až po operátor řezu.

1 $f(1).$

2 $f(6).$

3 $g(X) :- f(X), !, X > 5.$

4 $g(X) :- X \leq 2.$

$?- g(X).$

$?- f(X), !, X > 5.$



Efekty řezu na výpočetní strom

Upnutí: Další pravidla a fakta pro aktuální cíl se ignorují.

Ořezání: Fixování voleb provedených při vyhodnocování těla pravidla až po operátor řezu.

1 $f(1).$

2 $f(6).$

3 $g(X) :- f(X), !, X > 5.$

4 $g(X) :- X \leq 2.$

$?- g(X).$

$?- f(X), !, X > 5.$

$?- !, 1 > 5.$

Efekty řezu na výpočetní strom

Upnutí: Další pravidla a fakta pro aktuální cíl se ignorují.

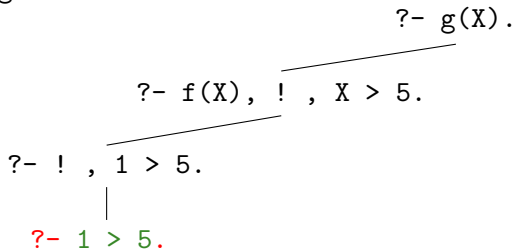
Ořezání: Fixování voleb provedených při vyhodnocování těla pravidla až po operátor řezu.

1 $f(1).$

2 $f(6).$

3 $g(X) :- f(X), !, X > 5.$

4 $g(X) :- X \leq 2.$



Efekty řezu na výpočetní strom

Upnutí: Další pravidla a fakta pro aktuální cíl se ignorují.

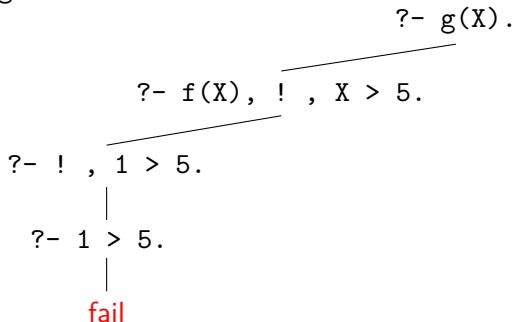
Ořezání: Fixování voleb provedených při vyhodnocování těla pravidla až po operátor řezu.

1 $f(1).$

2 $f(6).$

3 $g(X) :- f(X), !, X > 5.$

4 $g(X) :- X \leq 2.$



Efekty řezu na výpočetní strom

Upnutí: Další pravidla a fakta pro aktuální cíl se ignorují.

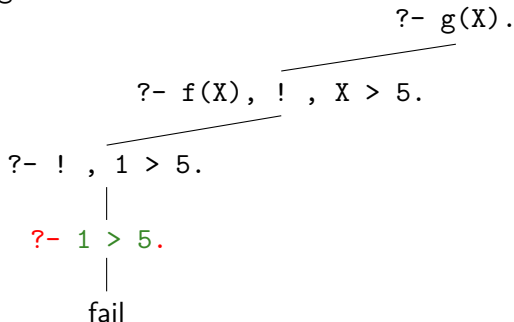
Ořezání: Fixování voleb provedených při vyhodnocování těla pravidla až po operátor řezu.

1 $f(1).$

2 $f(6).$

3 $g(X) :- f(X), !, X > 5.$

4 $g(X) :- X \leq 2.$



Efekty řezu na výpočetní strom

Upnutí: Další pravidla a fakta pro aktuální cíl se ignorují.

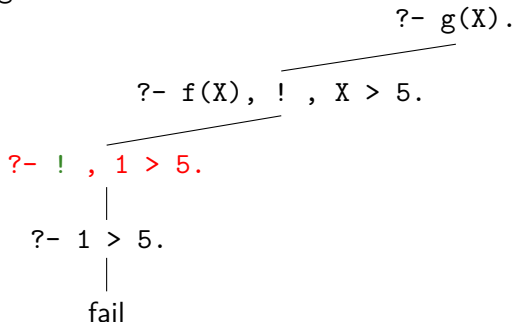
Ořezání: Fixování voleb provedených při vyhodnocování těla pravidla až po operátor řezu.

1 $f(1).$

2 $f(6).$

3 $g(X) :- f(X), !, X > 5.$

4 $g(X) :- X \leq 2.$



Efekty řezu na výpočetní strom

Upnutí: Další pravidla a fakta pro aktuální cíl se ignorují.

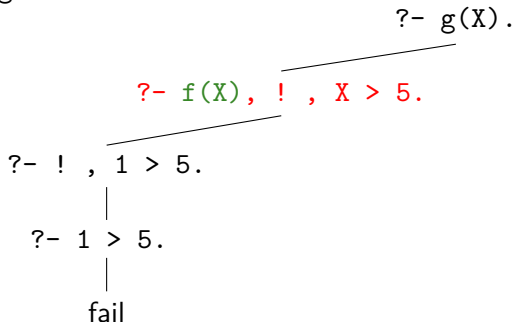
Ořezání: Fixování voleb provedených při vyhodnocování těla pravidla až po operátor řezu.

1 $f(1).$

2 $f(6).$

3 $g(X) :- f(X), !, X > 5.$

4 $g(X) :- X \leq 2.$

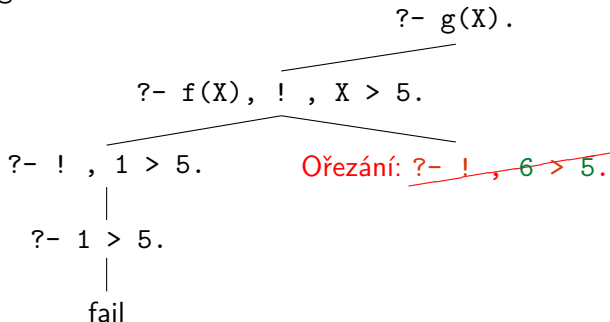


Efekty řezu na výpočetní strom

Upnutí: Další pravidla a fakta pro aktuální cíl se ignorují.

Ořezání: Fixování voleb provedených při vyhodnocování těla pravidla až po operátor řezu.

- 1 $f(1).$
- 2 $f(6).$
- 3 $g(X) :- f(X), !, X > 5.$
- 4 $g(X) :- X \leq 2.$

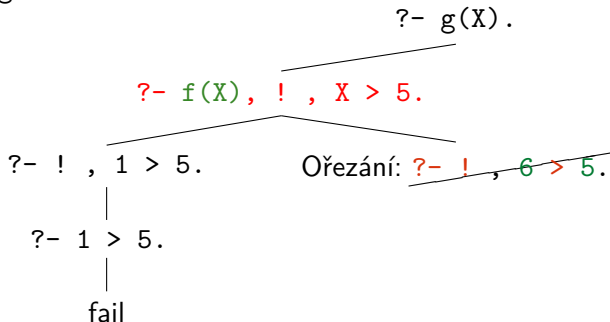


Efekty řezu na výpočetní strom

Upnutí: Další pravidla a fakta pro aktuální cíl se ignorují.

Ořezání: Fixování voleb provedených při vyhodnocování těla pravidla až po operátor řezu.

- 1 $f(1).$
- 2 $f(6).$
- 3 $g(X) :- f(X), !, X > 5.$
- 4 $g(X) :- X \leq 2.$

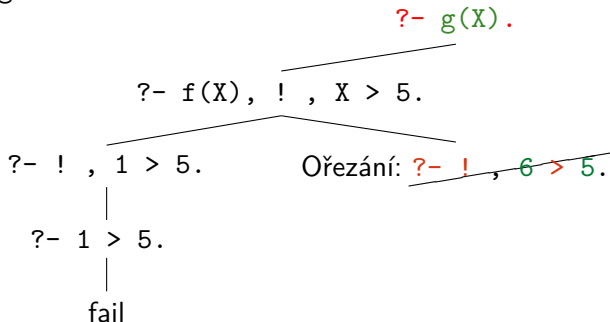


Efekty řezu na výpočetní strom

Upnutí: Další pravidla a fakta pro aktuální cíl se ignorují.

Ořezání: Fixování voleb provedených při vyhodnocování těla pravidla až po operátor řezu.

- 1 $f(1).$
- 2 $f(6).$
- 3 $g(X) :- f(X), !, X > 5.$
- 4 $g(X) :- X \leq 2.$

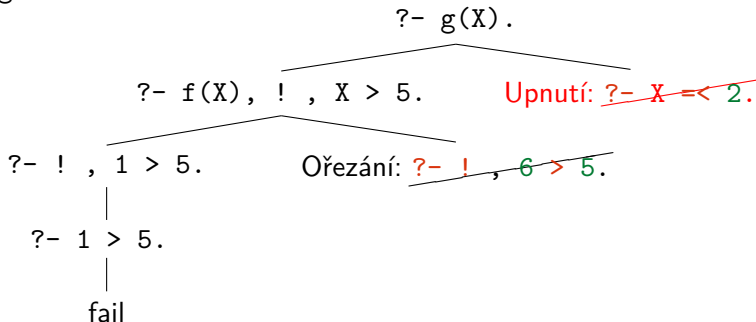


Efekty řezu na výpočetní strom

Upnutí: Další pravidla a fakta pro aktuální cíl se ignorují.

Ořezání: Fixování voleb provedených při vyhodnocování těla pravidla až po operátor řezu.

- 1 $f(1).$
- 2 $f(6).$
- 3 $g(X) :- f(X), !, X > 5.$
- 4 $g(X) :- X \leq 2.$

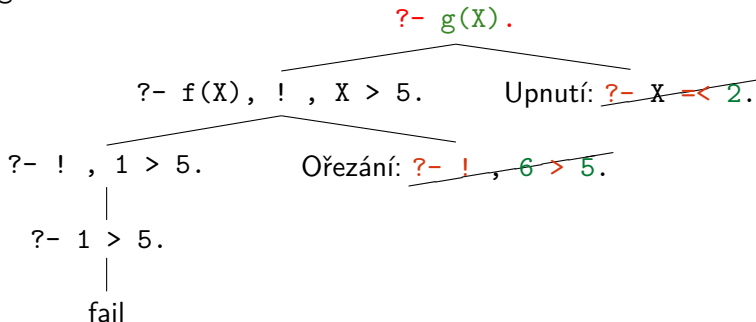


Efekty řezu na výpočetní strom

Upnutí: Další pravidla a fakta pro aktuální cíl se ignorují.

Ořezání: Fixování voleb provedených při vyhodnocování těla pravidla až po operátor řezu.

- 1 $f(1).$
- 2 $f(6).$
- 3 $g(X) :- f(X), !, X > 5.$
- 4 $g(X) :- X \leq 2.$



Příklad 11.1.9: Uvažte následující program:

```
1 f(X) :- g(X), h(X).  
2 f(3).  
3 g(0).  
4 g(1).  
5 g(2).  
6 h(1).  
7 h(2).
```

Vyhodnoťte dotaz $?- f(X).$ pro výše uvedený program a pro jeho 4 variace, které vzniknou výměnou příslušného pravidla za verzi s řezem uvedenou níže. Pro každý nakreslete kompletní SLD strom výpočtu se všemi řešeními.

- a) $f(X) :- !, g(X), h(X).$
- b) $f(X) :- g(X), h(X), !.$
- c) $f(X) :- g(X), !, h(X).$
- d) $g(1) :- !.$

Příklad 11.1.2: Jaký je rozdíl mezi následujícími definicemi predikátů `member/2`? Ve kterých odpovědích se budou lišit?

a) `mem1(H, [H|_]).`

`2 mem1(H, [_|T]) :- mem1(H, T).`

b) `mem2(H, [H|_]) :- !.`

`2 mem2(H, [_|T]) :- mem2(H, T).`

c) `mem3(H, [K|_]) :- H == K.`

`2 mem3(H, [K|T]) :- H \== K, mem3(H, T).`

Příklad 11.1.1: Navrhněte predikát `max/3`, který uspěje, jestliže je číslo ve třetím argumentu maximem čísel z prvních dvou argumentů. Uveďte řešení bez použití řezu i s ním.

Příklad 11.1.5: Napište predikát `remove/3`, který odstraní všechny výskyty prvního argumentu ze seznamu ve druhém argumentu a výsledný seznam unifikuje do třetího argumentu.

Koncová rekurze: akumulátory

Při použití rekurze si musíme pamatovat všechnen kontext volání „nad námi“:

```
1 length [] = 0
2 length (_:xs) = 1 + length xs
```

```
1 length [1,2,3]  $\rightsquigarrow$  1 + length [2,3]  $\rightsquigarrow$ 
2   1 + (1 + length [3])  $\rightsquigarrow$ 
3   1 + (1 + (1 + length []))
4    $\rightsquigarrow^*$  3
```

Koncová rekurze: akumulátory

Výpočet lze zrychlit, pokud je nejvýše jedno rekurzivní volání, a to je poslední operací výpočtu v tomto zanoření:

- při výpočtu můžeme předešlé záznamy na zásobníku volání (*call stack*) rovnou smazat
- typicky pomocná funkce s **akumulátorem** (extra parametr, v němž střeďáme mezivýsledek)

Koncová rekurze: akumulátory

Výpočet lze zrychlit, pokud je nejvýše jedno rekurzivní volání, a to je poslední operací výpočtu v tomto zanoření:

- při výpočtu můžeme předešlé záznamy na zásobníku volání (*call stack*) rovnou smazat
- typicky pomocná funkce s **akumulátorem** (extra parametr, v němž střeďáme mezivýsledek)

```
1 lengthAcc xs = lengthAcc2 xs 0
2   where lengthAcc2 [] len = len
3         lengthAcc2 (_:xs) a = lengthAcc2 xs (1 + a)
```

```
1 lengthAcc [1,2,3] ~> lengthAcc2 [1,2,3] 0 ~>
   lengthAcc2 [2,3] 1 ~> lengthAcc2 [3] 2 ~>
   lengthAcc2 [] 3 ~> 3
```

Koncová rekurze: akumulátory (porovnání)

Délka výpočtu délky seznamu s 7 000 000 prvky:

- klasická rekurzivní definice funkce `length` – 224 ms
- pomocí akumulátoru – 165 ms
- pomocí akumulátoru se striktním vyhodnocováním – 18 ms

Situace v Prologu je stejná:

```
1 myLength([], 0).
2 myLength(_|XS, L) :- myLength(XS, L1), L is L1 + 1.

1 lengthAcc(XS, L) :- lengthAcc2(XS, 0, L).
2
3 lengthAcc2([], Len, Len).
4 lengthAcc2(_|XS, Acc, Len) :-
5     Acc2 is Acc + 1,
6     lengthAcc2(XS, Acc2, Len).
```

Příklad 10.2.12: Definujte následující predikáty s pomocí akumulátoru:

- a) `listSum/2`, který sečte seznam čísel a na nečíselném seznamu neuspěje (použijte `number/1`),
- b) `fact/2`, který spočítá faktoriál zadaného čísla,
- c) `fib/2`, který (efektivně) spočítá n -tý člen Fibonacciho posloupnosti,
- d) `reverseAcc/2`, který obrátí pořadí prvků v seznamu.