Cvičení 2

Programování s omezujícími podmínkami

Roman Barták
Katedra teoretické informatiky a matematické logiky

`roman.bartak@mff.cuni.cz`
`http://ktiml.mff.cuni.cz/~bartak`

Prolog ve zkratce

Prologovský **„program"** se skládá z **pravidel** a **faktů.**
Každé **pravidlo** má strukturu **Hlava:-Tělo.**
- **Hlava** je (složený) term
- **Tělo** je dotaz (konjunkce termů)
  - Tělo typicky obsahuje všechny proměnné z Hlavy
- sémantika: **pokud je Tělo pravda potom odvoď Hlavu**

Fakty jsou vlastně pravidla s prázdným tělem (true).

Dotaz je konjunkce termů: Q = **Q1,Q2,…,Qn.**
- **Najdi pravidlo**, jehož hlava odpovídá dotazu Q1.
  - Pokud je takových pravidel více, vytvoř „bod volby" a použij první pravidlo.
  - Pokud neexistuje žádné pravidlo, potom se vrať na poslední bod volby a zkus zde alternativní pravidlo.
- **Použij tělo pravidla**, kterým v dotazu nahraď část Q1.
  - Pro pravidla, kde Tělo=true, dotaz Q1 zmizí.
- **Opakuj, dokud nezískáš prázdný dotaz.**

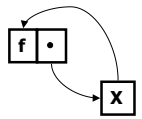
Technologie Prologu

Prolog = Unifikace + Backtracking

- **Unifikace** (matching) slouží pro
  - výběr vhodného pravidla (dle hlavy)
  - vytvoření odpovědní substituce
  - Jak?
    - substitucí se snaží udělat termy syntakticky identické
- **Backtracking** (prohledávání do hloubky) je pro
  - prozkoumání alternativ
  - Jak?
    - nejdříve vyřeš první část (zleva) dotazu
    - použij první pravidlo (shora)

Unifikace

- základní mechanismus pro **předávání informací**
- vlastně syntaktická rovnost termů po substituci termů do proměnných
- `?-X=f(a).` **-> X/f(a)**
- `?-f(X,a)=f(g(b),Y).` **-> X/g(b), Y/a**
- `?-f(X,b,g(a))=f(a,Y,g(X)).` **-> X/a, Y/b**
- `?-X=f(X).` **-> infinite term**
  - **kontrola výskytu** může zakázat nekonečné struktury
  - ale cyklické struktury se někdy hodí, protože vlastně modelují klasické ukazatele



Výběr pravidel

- **Unifikace se používá pro výběr pravidel.**

```
?-path(f(a),G).
```

  - pravidlo: path(X,Y):-arc(X,Y).
  - **unifikuj hlavu**: `X=f(a),Y=G`

```
?-arc(f(a),G).
```

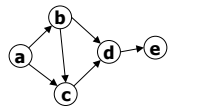
    - pravidlo(fakt): arc(a,b).
    - **unifikuj**: f(a)=a, G=b **-> fail**
    - pravidlo (fakt): arc(a,c).
    - **unifikuj**: f(a)=a, G=c **-> fail**
    - ...

Skládání odpovědi

- **Unifikace se používá pro složení odpovědi.**

```
path(X,Y,path(X,Y)):-
  arc(X,Y).
path(X,Y,path(X,PathZY)):-
  arc(X,Z),
  path(Z,Y,PathZY).

?-path(a,d,P).
P=path(a,path(b,d));
P=path(a,path(b,path(c,d)));
P=path(a,path(c,d));
no
```




```
node(a).
node(b).
node(c).
node(d).
node(e).

arc(a,b).
arc(a,c).
arc(b,c).
arc(b,d).
arc(c,d).
arc(d,e).
```


Předávání informace

- Jak Prolog „počítá" výstup?
- **Akumulátor**
  - Částečný výsledek je akumulovaný v atributu pravidla.
  - Požaduje počáteční nastavení atributu.
- **Skládání substitucí**
  - Složí výsledek z částí, které se vybudují později.
  - Specifické pro Prolog a použití substituce.

Akumulátor

Symbolický součet unárně reprezentovaných čísel (`0`, `s(0)`, `s(s(0))`, ...).
Výsledek je **akumulovaný** v parametru pravidla.

```
plus(0,X,X).
plus(s(X),Y,Z):-plus(X,s(Y),Z).
```

akumulátor

```
?-plus(s(s(s(0))),      s(0)    ,Sum).
?-plus(  s(s(0))  ,    s(s(0))  ,Sum).
?-plus(    s(0)   ,  s(s(s(0))) ,Sum).
?-plus(      0    ,s(s(s(s(0)))),Sum).
```

Skládání

Symbolický součet unárně reprezentovaných čísel.
Výsledkem je **složený term,** jehož úplný obsah bude naplněn později.

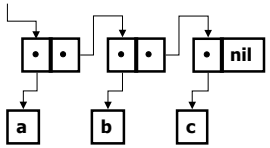
```
plus2(0,X,X).
plus2(s(X),Y,s(Z)):-plus2(X,Y,Z).
```

argument pro skládání výsledku

```
?-plus2(s(s(s(0))),s(0),S1). %S1=s(S2)
?-plus2(  s(s(0))  ,s(0),S2). %S2=s(S3)
?-plus2(    s(0)   ,s(0),S3). %S3=s(S4)
?-plus2(      0    ,s(0),S4). %S4=s(0)
```


Seznamy

- Jak reprezentovat **seznam prvků**?
- Použitím termů:
  - struktura s ukazateli
  - list(a,list(b,list(c,nil)))



- Prolog tuto strukturu poskytuje přímo:
  - `[Head|Tail]`
  - `[a,b,c] =[a|[b|[c|[]]]]`
  - prvkem seznam může být cokoliv, třeba jiný seznam
    - `[[q,2], 12, f(a,b), [[]]]`
- Jedná se pouze o „syntaktický cukr"!

Příslušnost do seznamu

- Jak zkontrolovat **příslušnost do seznamu**?
- Projitím seznamu od začátku, dokud není prvek nalezen.

```
member(X,[X|_]).
member(X,[_|T]):- member(X,T).
```

```
?-member(a,[a,b,a]). -> yes
?-member(X,[a,b,a]). -> X=a; X=b; X=a
?-member(a,L).       -> L=[a|_]; L=[_,a|_], …
```

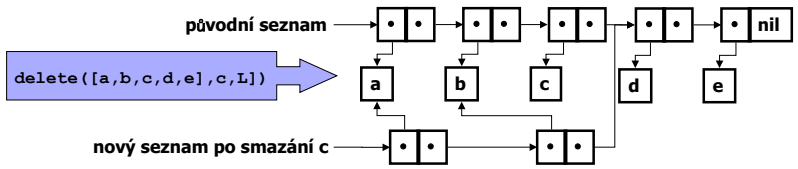

Smazání prvku

- Smazání prvního výskytu X ze seznamu List.
  `delete(List,X,ListWithoutX)`

```
delete([],_,[]).
delete([X|T],X,T).
delete([Y|T],X,[Y|NewT]):-
    X\=Y, delete(T,X,NewT).
```

X a Y nelze unifikovat

- Část seznamu před X je zduplikována!



Vložení prvku

- Vložení X před seznam `insert(L,X,LStartWithX)`:

```
insert(L,X,[X|L]).
```

- Přidání X na konec seznamu `add(L,X,LEndWithX)`:

```
add([],X,[X]).
add([Y|T],X,[Y|NewT]):-
  add(T,X,NewT).
```

  - Opět je celý seznam duplikován!
  - Stejná procedura může také odebrat poslední prvek!

```
?-add(NewList,X,[a,b,c,d]).
NewList=[a,b,c]
X=d
```


Spojení seznamů

- spojení dvou seznamů
  - `concat(L1,L2,L)`
  - L1=[a,b,c], L2=[d,e] -> L=[a,b,c,d,e]

```
concat([],L,L).
concat([H|T],L2,[H|NewT]):-
  concat(T,L2,NewT).
```

- Časová a prostorová složitost záleží na délce prvního seznamu!
- Proceduru lze použít také na rozdělení seznamů.

```
?-concat(List1,List2,[a,b,c,d]).
List1=[], List2=[a,b,c,d] ;
List1=[a], List2=[b,c,d] ;
...
```


Obrácení seznamu

- Obrať seznam
  - `revert(L,Rev)`
  - L=[a,b,c] -> Rev=[c,b,a]

```
revert([],[]).
revert([H|T],Rev):-
  revert(T,RT),
  add(RT,H,Rev).
```

Lepší technika je užitím **akumulátoru**!

```
revert1(List,Rev):-
  rev(List,[],Rev).
rev([],L,L).
rev([H|T],Acc,Rev):-
  rev(T,[H|Acc],Rev).
```

Pomalé a spotřebovává paměť!

Vyhýbejte se `add` (`concat`) ve svých kódech.

délka seznamu	`revert`	`revert1`
50000	**39 s.**	**0 s.**

Operátory

- zapisovat vše formou termu není moc komfortní
  - srovnejme '='(X,'+'(2,3)) a X=2+3
- pro lidi je vhodnější jiný zápis termů
  - např. **infixová notace „standarních" operací** (posktovaných Prologem)
- uživatel navíc může definovat **vlastní operátory**

```
:- op(precedence, type, name).
```

- opět se jedná pouze o **"syntaktický cukr"**

Aritmetické operace

```
?-X=1+2.   -> X=1+2
?-3=1+2.   -> no
```

Číslo je speciálním tvarem atomu. Má vlastní sémantiku (je to číslo)!

- Term 1+2 je různý od termu 3.
  - S termy totiž není spojena žádná sémantika!
- Potřebujeme speciální proceduru pro vyhodnocování numerických výrazů: "is"

```
?-X is 1+2.
X=3
```

- `X is Expr` funguje jako **aritmet. vyhodnocení**:
  - vyhodnoť Expr a srovnej (unifikuj) výsledek s X
- Pozor: **"is" není přiřazovací příkaz**!

```
?-X is 1+2, X is 7.
```


Aritmetické porovnání

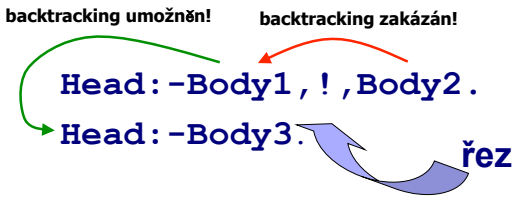
- Pokud máme čísla, můžeme je porovnávat?
- Prolog poskytuje standardní dotazy na porovnání čísel:
  - `X < Y`
    - numerická hodnota X je menší než num. hodnota Y

```
?-1<2.     -> yes
?-1+1<3.   -> yes
?-3<1+2.   -> no
```

  - `X > Y, X =< Y, X >= Y`

Řez

- Prolog používá **prohledávání do hloubky** pro výběr alternativních pravidel.
  - je-li více alternativ, udělá bod volby
- Můžeme explicitně **některé alternativy zakázat**?
  - **Řez** je metoda odstranění posledního bodu volby.



Řez v praxi

```
test1(X,Y):-
  member(Y,[[1,2],[3,4]]),member(X,Y).
test1(0,[]).

test2(X,Y):-
  !,member(Y,[[1,2],[3,4]]),member(X,Y).
test2(0,[]).

test3(X,Y):-
  member(Y,[[1,2],[3,4]]),!,member(X,Y).
test3(0,[]).

test4(X,Y):-
  member(Y,[[1,2],[3,4]]),member(X,Y),!.
test4(0,[]).
```

Příklady červeného řezu
Tento typ řezu není doporučen, protože mění běh výpočtu!

X	1	2	3	4	0
Y	[1,2]	[1,2]	[3,4]	[3,4]	[]

"Efektivní" řez

- Odstranění nenavštívených větví výpočtu (**zelený** řez).

Příklad:
rozdělte seznam na prvky menší než X a prvky, které nejsou menší než X (`split(List,X,LessX,NoLessX)`)

```
split([],_,[],[]):-!.
split([H|T],X,[H|T1],T2):-
  H<X,!,
  split(T,X,T1,T2).
split([H|T],X,T1,[H|T2]):-
  split(T,X,T1,T2).
```


Negace

- Jak ukázat neexistenci řešení?
- Užitečné pro negativní dotazy typu "není prvkem".

```
\+ :Goal
```

- nesvazuje proměnné!
- **Uvnitř negace:**

```
not(Query):-
  call(Query),!,fail.
not(_Query):-
  true.
```

META-PREDICATE
Prologovský cíl je term, takže každý term lze použít jako dotaz.

Pokud Query uspěje potom neúspěch (řez zakáže použití alternativního pravidla), jinak úspěch díky alternativnímu pravidlu.

Negace v praxi

- Negace v Prologu je **negace-jako-nespěch.**
  - To není stejné jako logická negace!

```
p(a).
p(b).
q(a).
?- \+ (p(X),q(X)), X=b.    -> fail
?- X=b, \+ (p(X),q(X)).    -> X=b
```

- **Zvláštní pozornost je potřeba věnovat negovaným cílům obsahujícím proměnné!**

Všechna řešení

- Jak získat všechna řešení dotazu Query?

```
findall(?Template,:Query,?List)
```

Posbírá všechny odpovědi na Query ve tvaru Template do seznamu List.

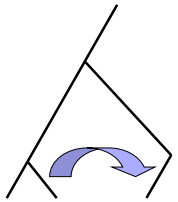
Příklad:
Najděte všechny sousedy uzlu "a".

```
?-findall(X,edge(a,X),Neigborhood).
?-findall(f(X),edge(a,X),Neigborhood).
?-findall(dzzz,edge(a,X),Neigborhood).
```

[b,c]
[f(b),f(c)]
[dzzz,dzzz]

Blackboard

- Jak předávat informaci při neúspěšném návratu (backtracking)?
- Jak předávat informaci mezi větvemi výpočtu?
- Lze přímo použít Prologovsko databázi!
  - `assert` pro uložení informace v jedné větvi
  - která budu dostupná i dále
- Lepší je používat **blackboard**!
  - čisté a efektivní řešení



Práce s blackboardem

- Informace uložená na blackboardu je jednoznačně identifikována atomem, kterém se říká **klíč** (atom si volí uživatel).

- `bb_put(:Key, +Term)`
- `bb_get(:Key, ?Term)`
- `bb_delete(:Key, ?Term)`
- `bb_update(:Key, ?OldTerm, ?NewTerm)`

SICStus 4

Blackboard-příklad

- Otestujte splnitelnost dotazu Query bez svázání proměnných.

```
sat(Query, _Answer):-
  bb_put(sat,no),
  once(Query), % finds one solution (if any)
  bb_put(sat,yes),
  fail.
sat(_Query,Answer):-
  bb_delete(sat,Answer).
```

Jiné řešení je použitím negace a if-then-else:

```
sat2(Query,Answer):-
  (\+ call(Query) -> Answer=no ; Answer=yes).
```


Blackboard v praxi

- Zjistěte počet odpovědí na dotaz Query
  `sat_num(:Query,-NumAnswers)`

```
sat_num(Query,_NumAnswers):-
  bb_put(counter,0),
  call(Query),
  bb_get(counter,N),
  N1 is N+1,
  bb_put(counter,N1),
  fail.
sat_num(_Query,NumAnswers):-
  bb_delete(counter,NumAnswers).
```

```
arc(a,b).
arc(a,c).
arc(a,d).

?-sat_num(arc(a,X),N).
N=3;
no
```

- Jiná metoda pomocí `findall`:

```
sat_num(Query,NumAnswers):-
  findall(x,Query,List),
  length(List,NumAnswers).
```

Blackboard-vlastnosti

- Blackboard funguje jako globální proměnná.
- **Pozor při "zanoření"!**
  - Pokud Query z předchozího slajdu volá `sat`, potom se data na blackboardu promíchají.
- Blackboard zachovává strukturu termu, ale "ztrácí" vazby na proměnné!!

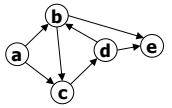
```
?-A=term(X,f(X)), bb_put(test,A), X=a,
  bb_get(test,B).
A = term(a,f(a)),
B = term(_A,f(_A)),
X = a ? ;
no
```


Domácí úkol

Vytvořte program pro nalezení jedné z nejkratších cest mezi zadanou dvojicí uzlů.

- **Databáze (graf):**

```
arc(a,b).
arc(a,c).
arc(b,c).
arc(b,e).
arc(c,d).
arc(d,b).
arc(d,e).
```



- **Očekávané odpovědi:**

```
?-shortest_path(a,a,P).
  P = [a]

?- shortest_path(a,e,P).
  P = [a,b,e]

?- shortest_path(e,b,P).
  no
```