Složitější domény

Petr Štěpánek

S využitím materialu Krysztofa R. Apta

2006

V této části se budeme zabývat seznamy a binárními stromy. Naším cílem není tyto datové struktury podrobně rozebírat, spíše nám jde o to, ukázat, co všechno je možné v čistém Prologu s jeho prostředky naprogramovat.

Seznamy

Obecně řečeno, datová struktura, která pro posloupnosti podporuje jen jedinou operací - vložení nové položky na začátek - se obvykle nazývá seznam.

Seznamy patří mezi základní datové struktury Prologu, a proto si zasloužily uživatelsky přátelské prostředí, které obsahuje speciální vestavěné funkce pro různé formy označování seznamů.

Základem označování a definice seznamů je (jediná) konstanta [] a binární funkční symbol [. | ..] .

Formálně se seznamy definují induktivně

- [] je seznam ,
- je-li `xs` seznam , potom [`x` | `xs`] je také seznam; `x` se nazývá *hlavou* a `xs` *tělem* seznamu.
- [] se nazývá prázdný seznam.

Například `[s(0)|[]]` a `[0|[x|[]]]` jsou seznamy, zatímco `[0|s(0)]` není seznam, protože `s(0)` není seznam .

Tento základní způsob označování seznamů není příliš přehledný, a proto se pro označování seznamů zavadějí synonyma. Také zde se postupuje nduktivně , pro $n \geq 1$

- $[s_0|[s_1,\ldots,s_n|t]]$ se zkracuje na $[s_0,s_1,\ldots,s_n|t]$,
- $[s_0,s_1,\ldots,s_n|[]]$ se zkracuje na $[s_0,s_1,\ldots,s_n]$.

Tedy

`[a|[b|c]]` se zkracuje na `[a,b|c]`, a
`[a|[b,c|[]]]` se zkracuje na `[a,b,c]`.

Použijeme-li vestavěný predikát rovnosti `=/2` s infixní notací, jakoby byl vnitřně definován jedinou klauzulí:

```
% X = Y ← X a Y   se unifikují
```

Prolog generuje následující konverzace

```
?- X = [a|[b,|c]]
X - [a,b|c]
?- [a,b|c] = [a|[b|c]]
yes
?- X = [a|[b,c|[]]]
X = [a,b,c]                atd.
```

Tyto konvence usnadňující čtení seznamů přidáme k čistému Prologu, také budeme přidávat koncovky ´s´ k proměnným za které se mají dosazovat seznamy.

Ačkoliv uvedené příklady to neuvádějí, prvky seznamů nemusí být jen základní termy.

Následuje směska programů, které používají seznamy.

LIST (SEZNAM)

```
% list(Xs) ← Xs je seznam.
  list([]).
  list[_|Ts]) ← list(Ts).
```


Stejně jako u programu *NUMERAL* platí

- je-li `t` seznam, dotaz `list(t)` končí úspěšně ,
- je-li `t` základní term, který není seznamem, dotaz `list(t)` konečně selhává,
- pro proměnnou `X` , dotaz `list(X)` generuje nekonečně mnoho odpovědí.

Délka seznamu se definuje induktivně

- délka prázdného seznamu `[]` je 0 ,
- je-li n délka seznamu `Xs,` potom délka seznamu `[X|Xs]` je n+1 .

Program *LENGTH*

```
len([],0).
len([_|Ts],s(N)) ← len(Ts,N).
```

Pomocí programu *LENGTH* můžeme počítat délku seznamů pomocí numerálů.

```
 ?- len([a,b,c,d],N).
N = s(s(s(s(0))))
```

Program může také generovat seznam dané délky, sestávající z proměnných.

```
  ?- len(Xs,s(s(s(s(0))))).
N =  [_A,_B,_C,_D]
```

kde `_A,_B,_C,_D` jsou proměnné generované sytémem Prologu. Později ukážeme, k čemu jsou takové proměnné užitečné.

Program *MEMBER*

`% member(Element,List)` ← `Element` je prvkem `List`(u).

```
member(X,[X|_]).
member(X,[_|Xs]) ← member(X,Xs).
```

```
 ?- member(X,[1,2]).
X = 1 ;
X = 2 ;
no
 ?- member_both(X,[1,2,3],[2,3,5]).
X = 2 ;
X = 3 ;
no
```

Špatně typované dotazy mohou vést k nečekaným odpovědím.

```
?- member(0,[0,s(0)].
yes
```

Tak jako u programů *LIST* a *NUMERAL* si odpověď ´no´ na takový dotaz vynutíme vložením testu do první klauzule programu *MEMBER* .

```
member(X,[X|Xs]) ← list(Xs).
```

Program *SUBSET*

```
% subset(Xs,Ys) ← každý prvek seznamu Xs je prvkem seznamu Ys .
subset([],_).
subset([X|Xs],Ys) ←  member(X,Ys),subset(Xs,Ys).
```


V programu *SUBSET* nejde vlastně o množiny ale o multimnožiny, v seznamech je dovoleno opakování položek. Můžeme tedy dostat

```
?- subset([a,a],[a]).
yes
```

APPEND je známý program pro zřetězení (spojování) seznamů.

```
app([],Ys,Ys).
app([X|Xs],Ys,[X|Zs]) ← app(Xs,Ys,Zs).
```

Program *APPEND* má mnoho použití, můžeme s jeho pomocí vynechat určitou položku ze seznamu. To může být jedna z verzí programu *SELECT* (*DELETE*) .

```
select(X,Xs,Ys) ← app(X1s,[X|X2s],Xs),
                  app(X1s,X2s,Zs)
                + APPEND .
```

Elegantnější verze programu *SELECT* definuje predikát `select` induktivně.

`% select(X,Xs,Zs)` ← `Zs` vznikne vynecháním jednoho výskytu položky `X` ze seznamu `Xs`.

```
select(X,[X,Xs],Xs).
select(X,[Y|Xs],[Y,Zs]) ← select(X,Xs,Zs).
```

Pokud se položka `X` v seznamu `Xs` nevyskytuje, oba programy skončí výpočet neúspěchem. Přidáním klauzule

```
select(X,Xs,Xs).
```

na konec programu skončí výpočet i v takovém případě úspěšně.

Pomocí programu *SELECT* je možné sestrojit program, který postupně generuje všechny permutace daného seznamu.

PERMUTACE

```
perm([],[]).
perm(Xs,[X|Ys]) ← perm(Xs,[X|Ys]),
                      select(X,Xs,Zs),
                      perm(Zs,Ys).
```

Jinou variantu programu *PERMUTACE* získáme rozepsáním definice predikátu `select` pomocí `app(end)`.

Pomocí programu *APPEND* můžeme snadno definovat počáteční i koncový úsek seznamu.

```
prefix(Xs,Ys) ← app(Xs,_,Ys).
suffix(Xs,Ys) ← app(_,Xs,Ys).
```

a program *APPEND*.

Podseznam nějakého seznamu pak lze formálně definovat takto

• Seznam `as` je podseznamem seznamu `bs,` jestliže `as` je prefixem nějakého sufixu seznamu `bs` .

Predikát `sublist` je pak definován jedinou klauzulí:

```
sublist(Xs,Ys) ← app(_,Zs,Ys),app(Xs,_,Zs).
```

V této klauzuli je `Zs` sufixem `Ys` a `Xs` je prefixem `Zs.`
Abychom dostali program *SUBLIST* přidáme ještě program *APPEND.*

Obracení seznamů. Krátce připomeneme dvě verze programů na obracení seznamů, *Naivní revers* a *Revers s akumulátorem.*

Naivní REVERS

`% reverse(Xs,Ys)` ← `Ys` vznikne obracením seznamu `Xs.`

```
reverse([],[]).
reverse([X|Xs],Ys) ← reverse(Xs,Zs),
                     app(Zs,[X],Ys).
```

Tento program je oblíbený pro svou neefektivnost jako testovací program pro implementace. Délka výpočtu je kvadratická vzhledem k délce vstupního seznamu.

Vyjádříme-li klauzule rekurzivními rovnicemi podle délky x daného seznamu, dostaneme pro první klauzuli

$r(0) = 1$

a pro druhou klauzuli

$r(x + 1) = r(x) + a(x)$,
$a(x) = x + 1$.

Celkem tedy $r(x) = x \,.\, (x + 1) / 2 + 1$.

REVERSE s akumulátorem je algoritmus lineární vzhledem k délce seznamu.

```
reverse(X1s,X2s) ← reverse(X1s,[],X2s),

reverse([],Xs,Xs).
reverse([X|X1s],X2s,Ys) ← reverse(X1s,[X|X2s],Ys).
```

Anonymní proměnné se mohou používát nejenom v klauzulích, ale i v dotazech, takže

```
  ?- reverse([a,b,a,d],[X|Ls]).

 X = d
Ls = [a,b,a]
```

ale také

```
?- reverse([a,b,a,d],[X|_]).
X = d
```

Připomeňme, že každý výskyt anonymní proměnné zastupuje jinou proměnnou, proto v databázi *CENTRAL_AMERICA* ,kde predikát sousedství je definován jen pro různé země, dotaz `neighbour([X,X])` neuspěje, ale dotaz `neighbour([_,_])` ano.

Přitom anonymní proměnné nelze chápat jako existenční kvantifikátory, jak by to naznačoval poslední dotaz, ale v dotazu

```
?- app(X1s,[X|X2s],[a,b,a,c]), app(X1s,X2s,Zs).
```

musíme pomocné proměnné `X1s` a `X2s` vyjmenovat, protože se každá z nich v dotazu vyskytuje dvakrát.

Pomocí programu *REVERSE* můžeme jednoduše vyjádřit test, zda daný seznam tvoří palindrom.

Program *PALINDROM*

`% palindrom(Xs)` ← seznam `Xs` čtenýoběma směry dává stejné slovo.

`palindrom(Xs)` ← `reverse(Xs,Xs).`

a program *REVERSE.*

```
?- palindrom([t,a,b,a,k,a,b,a,t]).
yes.
```

Následující program ukazuje jak anonymni proměnné mohou zjednodušit čitelnost programu.

Problém: vytvořit seznam, ve kterém budou tři jedničky, tři dvojky, ... , tři devítky uspořádány tak, že pro i = 1, 2, ... , 9 mezi každými dvěma následujícími výskyty čísla i bude právě i čísel.

Program *SEKVENCE*

```
% sequence(Xs) ← Xs je seznam 27 položek.

sequence([_,...27krát...]).

question(Ss) ← sublist([1,_,1,_,1]),
               sublist([2,_,_,2,_,_,2],
               sublist([3,_,_,_,3,_,_,_,3]),
                  ...
               sublist([9,_,9krát,9,_,9krát,_,9]).
```


a program SUBLIST. Následující konverzace s Prologem vydá všech šest řešení problému.

```
  ?- question(Ss).

Ss = [1,9,1,2,1,8,2,4,6,2,7,9,4,5,8,6,3,4,7,5,3,9,6,8,3,5,7] ;
Ss = [1,8,1,9,1,5,2,6,7,2,8,5,2,9,6,4,7,5,3,8,4,6,3,9,7,4,3] ;
Ss = [1,9,1,6,1,8,2,5,7,2,6,9,2,5,8,4,7,6,3,5,4,9,3,8,7,4,3] ;
Ss = [3,4,7,8,3,9,4,5,3,6,7,4,8,5,2,9,6,2,7,5,2,8,1,6,1,9,1] ;
Ss = [3,4,7,9,3,6,4,8,3,5,7,4,6,9,2,5,8,2,7,6,2,5,1,9,1,8,1] ;
Ss = [7,5,3,8,6,9,3,5,7,4,3,6,8,5,4,9,7,2,6,4,2,8,1,2,1,9,1] ;
no
```

Při podrobnějším prohlédnutí nabízených řešení zjistíme, že poslední tři seznamy (řešení) vzniknou obrácením prvních tří seznamů.

Není těžké ukázat, že je-li seznam řešením tohoto problému, pak jeho obrácením vznikne také řesení.

Domény závislé na aplikacích

Budeme se zabývat doménami, které používají funkčních symbolů závislých na aplikacích. Takové domény odpovídají složeným typům dat v imperativních a funkcionálních jazycích.

Obarvení mapy.

V tomto případě se hodí již uvedný příklad Střední Ameriky, kde na obarvení stačí jen tři barvy. Program i řešení tak budou přehlednější.

Podle definice, mapa je správně obarvena danou množinou barev, pokud žádné dvě sousedící země nejsou obarveny stejnou barvou.

Volba vhodné reprezentace dat může podstatně zjednodušit řešení daného problému.

Mapu budeme reprezentovat seznamem oblastí a seznamem barev.

Hlavní váha tedy spočívá na *reprezentaci oblastí*. Každá oblast je určena svým jménem, barvou a barvami jejich sousedů, tedy termem

`region(name,colour,neighbours),`

kde `neighbours` je seznam barev sousedících oblastí.

Vlastní program je jen odrazem následující definice správného obarvení

- Mapa je správně obarvená, jsou-li správně obarveny všechny oblasti.
- Oblast `region(name,colour,neighbours)` je správně obarvená, jestliže `colour` a položky seznamu `neighbours` patří do seznamu použitelných barev a `colour` není v seznamu `neighbours` .

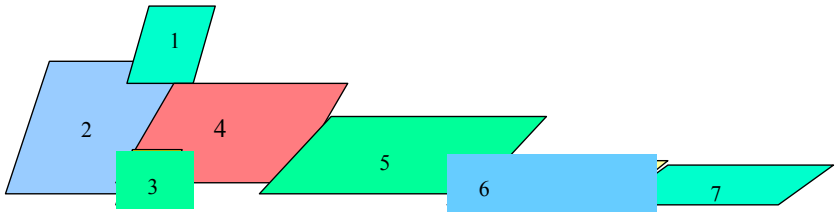
```
% colour_map(Map,Colours)... Map(a) je správně obarvena
barvami ze seznamu Colours.
colour_map([],_).
colour_map([Region|Regions],Colours) ←
                    colour_region,Colours),
                    colour_map(Regions,Colours).
% colour_region(Region,Colours) ← Region a její sousedé
                                  jsou správně obarveni
                                  barvami z Colours.
colour_region(region(_,Colour,Neighbours),Colours)
                  select(Colour,Colours,Colours1),
                  subset(Neighbours,Colours1).
                  +SELECT a SUBSET .
```

Nejprve je třeba vhodná reprezentace mapy Střední Ameriky, vyjádřená jediným atomem u predikátu `map`:


```
map([
   region(belize,Belize,[Guatemala]),
   region(guatemala,Guatemala,[Belize,El_Salvador,Honduras]),
   region(el_salvador,El_Salvador,[Guatemala,Honduras]),
   region(honduras,Honduras,[Guatemala,El_Salvador,Nicaragua]),
   region(nicaragua,Nicaragua,[Honduras,Costa_Rica]),
   region(costa_rica,Costa_Rica,[Nicaragua,Panama]),
   region(panama,Panama,[Costa_Rica])
   ]).

?- map(Map),colour_map(Map,[green,blue,red]).

Map = [
region(belize,green,[blue]),
region(guatemala,blue,[green,green,red]),
region(el_salvador,green,[blue,red]),
region(honduras,red,[blue,green,green]),
region(nicaragua,green,[red,blue]),
region(costa_rica,blue,[green,green]),
region(panama,green,[blue])
]).
```



1 Belize
2 Guatemala
3 El Salvador
4 Honduras
5 Nicaragua
6 Costa Rica
7 Panama

Binární stromy

Binární stromy jsou jinou fundamentalní datovou strukturou. Prolog nemá žádné vestavěné prostředky k těmto strukturám, na rozdíl od seznamů. V literatuře o Prologu najdeme řady různých definic binárních stromů. Musíme si tedy posloužit definici vlastní.

Ingredience.

- konstanta `void` označující prázdný strom,
- ternarní funkce `tree` , která konstruuje binární strom přidáním levého a pravého podstromu k listu dosud sestrojeného binárního stromu.

Binární stromy budeme definovat induktivně podle následující definice.

Nejprve neformální definice:

- `void` je prázdný binární strom ,
- jsou-li `left` a `right` binární stromy, potom term

`tree(x,left,right)` je binární strom. Říkáme, že `x` je jeho kořenem a `left` je jeho levým a `right` pravým podstromem.

Prázdné binární stromy " zaplňují " místa (podstromy), ve kterých nejsou uložena žádná data.

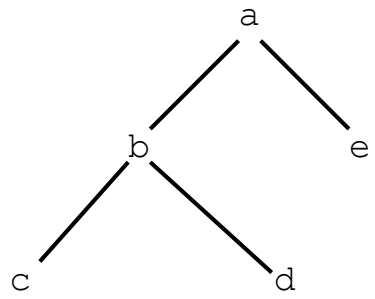
Při vizualizaci je praktické jejich přítomnost ignorovat.

Tak například binární strom `tree(c,void,void)` potom odpovídá stromu jehož jediným uzlem je kořen `c` .

Naproti tomu binární strom:

```
tree(a,tree(b,tree(c,void,void),tree(d,void,void)),
tree(e,void,void))
```

lze znázornit následovně



takže uzly tohoto stromu jsou reprezentovány termy tvaru

`tree(s,void,void)`

kde `s` nabývá hodnot `c, d, e` .

Z pragmatických důvodů budeme mluvit krátce o *stromech* místo o *binárních stromech* .

Musíme si ovšem vždy uvědomovat, že je rozdíl mezi termem, který je (binární) strom podle naší definice, a jeho vizualizací, která je stromem.

Jako obvykla začneme programem, který testuje zda daný term je binárním stromem.

```
% bin_tree(T) <-  T je strom.
bin_tree(void).
bin_tree(tree(_,Left,Right)) <-
                                    bin_tree(Left),
                                    bin_tree(Right).
```


Tak jako vždy konstatujeme:

- pro strom `t` dotaz `bin_tree(t)` končí úspěšně,
- pro *základní* term `t` , který není stromem dotaz `bin_tree(t)` konečně selhává,
- pro proměnnou `x` dotaz `bin_tree(x)` generuje nekonečně mnoho odpovědí .

Jak známo, stromy lze použít k ukládání dat a udržování různých operací s těmito daty.

Prvky stromu: je přirozené, že náležení prvků `x` do stromu `T` odpovídá induktivní definici stromu.

Tedy
položka `x` náleží do stromu `T` , právě když

- `x` je kořenem stromu `T` nebo
- `x` je prvkem levého podstromu `T` nebo
- `x` je prvkem pravého podstromu `T` .

Program TREE_MEMBER

```
% tree_member(E,T) <-  E je prvkem stromu Tree.
tree_member(X,_,_)).
tree_member(X,tree(_,Left,_)) <-
                              tree_member(X,Left).
tree_member(X,tree(_,_,Right)) <-
                              tree_member(X,Right).
```


Program TREE_MEMBER lze použít například k těmto účelům:

- k testování, zda daná položka `x` náleží do stromu `t` dotazem `tree_member(x,t)`,
- nebo k postupnému generování všech prvků daného stromu `t` dotazem `tree_member(X,t)`.

Procházení stromem se obvykle provádí třemi způsoby

- *pree_order* - v každém podstromu je navštíven nejprve kořen, potom uzly levého a nakonec pravého podstromu,
- *in_order* - nejprve jsou navštíveny všechny uzly levého podstromu, potom kořen stromu a nakonec všechny uzly pravého podstromu,
- *post_order* - nejprve jsou navštíveny uzly levého postromu, potom pravého a nakonec kořen stromu.

Každý se jednoduše překládá do programu v Prologu.

Například:

`% in-order(Tree,List) <- List` je seznam uzlů stromu `Tree` v pořadí, které odpovídá průchodu *in_order* .

```
in-order(void,[]).
in-order(tree(X,Left,Right),Xs) <-
                              in-order(Left,Ls),
                              in-order(Right,Rs),
                              app(Ls,[X|Rs],Xs).
```

\+ program APPEND

Frontier (hranice) stromu je seznam vytvořený z jeho listů. Připomeňme, že listy jsou vyjádřeny termy

`tree(a,void,void).`

Při programování výpočtu hranice stromu rozeznáváme tři typy stromů:

- prázdný strom - `void` ,
- list - `tree(a,void,void)`,
- neprázdný strom, který není listem (krátce *nel-tree*) , reprezentovaný termem `tree(x,l,r)`.

Program FRONTIER

(nejprve definujeme pomocnou relaci `nel_tree`.

`% nel_tree(t) <- t` je *nel_tree* .

```
nel_tree(tree(_,tree(_,_,_),_)).
nel_tree(_,_,tree(_,_,_))).
```

```
% front(Tree,List) <- List je hranicí stromu Tree.

front(void,[]).
front(tree(X,void,void),[X]).
front(tree(X,L,R),Xs) <-
                          nel_tree(tree(X,L,R)),
                          front(L,Ls),
                          front(R,Rs),
                          app(Ls,Rs,Xs).
                        + APPEND
```

Dotaz `nel-tree(t)` může končit úspěchem i pro termy, které nejsou stromy, následující (zdánlivě jednodušší) program proto není korektní.


```
% front(Tree,List) <- List je hranicí stromu Tree.

front(void,[]).
front(tree(X,void,void),[X]).
front(tree(_,L,R),Xs) <-
                          front(L,Ls),
                          front(R,Rs),
                          app(Ls,Rs,Xs).
                        + APPEND
```

není korektní. Dotaz `front(tree(X,void,void),Xs)` dává dvě různé odpovědi `{Xs/[X]}` podle druhé klauzule a `{Xs/[]}` podle třetí klauzule.

Ovšem v předchozí verzi byl podprogram NEL-TREE používán jen na termy, které jsou stromy.

Shrnutí

Doména	*Inventář*
prázdná	žádné funkce ani predikáty
konečná	jen konstanty
Seznamy	konstanta a binární funkční symbol
Binární stromy	konstanta a ternární funkční symbol