Prírodovedecká fakulta Univerzity Pavla Jozefa Šafárika
Ústav informatiky

Diplomová práca

IMPLEMENÁCIA FORMÁLNYCH ŠTRUKTÚR V DATABÁZOVÝCH SYSTÉMOCH

Košice 2003 Peter Gurský

1

Univerzita Pavla Jozefa Šafárika v Košiciach, Prírodovedecká fakulta

Ústav informatiky

Zadanie diplomovej práce

Meno a priezvisko diplomanta: Peter Gurský

Študijný odbor: Informatika
Zameranie: Informačné a znalostné systémy

Názov: Implementácia formálnych štruktúr v databázových systémoch.

Cieľ:
Vytvoriť systém umožňujúci uloženie množiny termov v databáze a nasledovnú prácu s nimi. Analyzovať rôzne možnosti implementácie s využitím metód indexovania termov.

Odporúčaná literatúra:

1. F. Baader, W. Snyder, Unification Theory, in: A. Robinson, A. Voronkov (eds.), Handbook of automated reasoning, Elsevier, 2001.
2. R. Sekar, I. V. Ramakrishnan, A. Voronkov, Term indexing, in: A. Robinson, A. Voronkov (eds.), Handbook of automated reasoning,Elsevier, 2001.
3. P. Singleton, O. P. Brereton, Storage and retrieval of first-order terms using a relational database, in: M. F. Worboys, A. F. Grundy (eds.), Advances in Databases, Proceedings of BNCOD 11 (Keele, 1993),Lecture notes in computer science, vol. 696, Springer, 1993.

Poznámka:

Vedúci diplomovej práce: RNDr. Peter Eliaš, PhD.
Konzultant diplomovej práce:
Oponent diplomovej práce: RNDr. Stanislav Krajči, PhD.

Dátum zadania dipl. práce: 15.06.2001
Dátum odovzdania dipl. práce:

prof. RNDr. Peter Vojtáš, DrSc.
vedúci ústavu

Dátum potvrdenia 18.01.2002

Vyhlásenie

Vyhlasujem, že túto diplomovú prácu som vypracoval samostatne na základe štúdiom získaných poznatkov. Použitú literatúru som uviedol v práci.

Košice, Apríl 2003 Peter Gurský

Dovoľujem si touto cestou poďakovať vedúcemu mojej diplomovej práce RNDr. Petrovi Eliašovi, PhD. za všestrannú pomoc, cenné rady a usmerňovanie v práci.

Obsah

Úvod

V mojej diplomovej práci sa venujem problému uchovávania formálnych štruktúr v databáze. Konkrétne ide o uchovávanie množín termov a nasledujúcu prácu s nimi. S termami sa pracuje v mnohých odvetviach matematiky aj informatiky. Vyskytujú sa napríklad v logickom a funkcionálnom programovaní, alebo dokazovačoch viet výrokového či predikátového počtu. Pri práci s termami sa často narazí na problém uchovávania množiny termov. Jedno z možných riešení tohoto problému ponúka moja diplomová práca.

Okrem samotného uchovania množín termov som riešil aj problém rýchleho výberu termov spĺňajúcich určité podmienky. Ide o také podmienky, ako sú generalizácia, unifikácia, inštancia, premenovanie premenných alebo identita, ktoré sa najčastejšie vystytujú pri práci s termami.

Formálnemu riešeniu problému unifikácie je venovaná kapitola teória unifikácie. Spomína sa v nej niekoľko unifikačných algoritmov s rôznou mierou zložitosti. V niektorých prípadoch je rozvedený aj problém korektnosti a úplnosti unifikačných algoritmov.

K výraznému zrýchleniu výberu termov spĺňajúcich spomínané podmienky prispieva indexovanie termov. Niekoľkým indexovacím technikám a ich použiteľnosti pre rôzne aplikácie sa venuje kapitola indexovanie termov. V tejto kapitole je rozvedený aj problém výberu dátových štruktúr pre uchovanie termov ako aj indexov.

Na základe popísanej teórie som vytvoril knižnicu v jazyku php umožňujúcu uchovávanie a prácu s množinami termov. K tejto knižnici je vytvorená aj webová aplikácia, v ktorej sú ukázané všetky možnosti tejto knižnice. V Praktickej časti je popísaná knižnica aj jej obslužná aplikácia.

K tejto práci je priložený CD-ROM, ktorom sa nachádzajú inštalačné programy pre webový server Apache, databázový server MySQL a programovací jazyk PHP pre Linux aj Windows, ktoré sú potrebné pre spustenie celého systému, a pravdaže samotný program. Okrem toho obsahuje aj všetku použitú literatúru a elektronickú verziu tejto diplomovej práce.

1.Teória unifikácie

Unifikácia je proces na ktorom ja založené množstvo metód automatizovanej dedukcie. Využíva sa napríklad v logickom a funkcionálnom programovaní, dokazovačoch viet, deduktívnych databázach a iných aplikáciách. Teória unifikácie abstrahuje od týchto aplikácií a poskytuje formálne definície, ich vlastnosti a hlavne konkrétne unifikačné algoritmy ktoré sa môžu použiť vo výsledných aplikáciách.

1.1. Základné definície

Na to aby sme mohli dôjsť k prvému unifikačnému algoritmu, musíme sa dohodnúť na nejakých označeniach a vedieť sa pohybovať v niekoľkých pojmoch. Začneme pojmom term:

Definícia 1.1. *Term* je štruktúra, ktorá spĺňa jednu z nasledujúcich podmienok:

1. ak x je premenná, tak x je term
2. ak f je n-árny funkčný symbol pre $n \geq 0$ a $t_1,\ldots,t_n$ sú termy tak $f(t_1,\ldots,t_n)$ je term

Je potrebné poznamenať, že ak f je 0-árny funkčný symbol, tak budeme hovoriť, že f je konštanta.

Označenie 1.1. Algebru termov označujeme *T(F,V)*, kde *F* sú funkčné symboly, ktoré ju generujú a *V* je spočítateľná nekonečná množina premenných.

Označenie 1.2. Množinu premenných vyskytujúcich sa v terme t označujeme *Vars(t)*.

Definícia 1.2. *Substitúcia* je zobrazenie z množiny premenných do množiny termov.

Substitúcia je teda množina usporiadaných dvojíc <premenná,term>. Pre väčšiu názornosť budeme substitúciu reprezentovať ako množinou priradení. Ak máme nejakú substitúciu σ zobrazujúcu premennú x_1 do termu s_1, premennú x_2 do termu s_2, ..., premennú x_n do termu s_n, tak jej explicitné vyjadrenie budeme písať: $\sigma=\{x_1\rightarrow s_1,x_2\rightarrow s_2,\ldots x_n\rightarrow s_n\}$.

Definícia 1.3. Aplikácia substitúcie σ na term t (ozn. $t\sigma$) je definovaná nasledovne:

$$t\sigma = \begin{cases} s & \text{ak } t\in Var \text{ a } \{t\rightarrow s\}\subseteq\sigma \\ f(t_1\sigma,\ldots,t_n\sigma) & \text{ak } t=f(t_1,\ldots,t_n) \end{cases}$$

napr. ak $\sigma=\{x\rightarrow f(a),y\rightarrow x\}$ tak $y\sigma=x$, $g(x,b)\sigma=g(x\sigma,b\sigma)=g(f(a),b)$, $z\sigma=z$

Je vidieť, že aplikácia substitúcie na konštantu nespôsobuje zmenu konštanty.

Definícia 1.4 *Dom(s)* := {x; x Var & xσ≠x}
Ran(s) := $\bigcup_{x \in Dom(\sigma)}$ {xσ}
Vran(s) := Vars(Ran(σ))

napr. ak σ={y→y, x→f(a), z→f(x)}, kde x,y,z∈Var, a∈Const, potom
Dom(σ)={x,z}
Ran(σ)={f(a),f(x)}
Vran(σ)={x}

Definícia 1.5. Skladanie 2 substitúcií (ozn. sq) je definované vzťahom: tσθ=(tσ)θ, kde t je ľubovoľný term.

Definícia 1.6. Zúženie substitúcie σ na množinu premených X (ozn. $s|_X$) je substitúcia, ktorá je indentita všade až na X Dom(σ), kde je zhodná so substitúciou σ.

napr. ak σ={x→y,z→a,u→f(a)} a X={u,y,z} tak $\sigma/_X$={z→a,u→f(a)}

V jednotlivých unifikačných algoritmoch sa často stretávame so skladaním substitúcií. Na túto operáciu existuje nasledovný jednoduchý algoritmus.

Algoritmus 1.1. na konštrukciu kompozície σθ kde σ aj θ sú zadané explicitne.

1. aplikuj θ na každý term z Ran(σ)→získame σ_1
2. odstráň z θ všetky zobrazenia x→t kde x Dom(σ)→získame θ_1
3. odstráň z σ_1 všetky zobrazenia x→x → získame σ_2
4. vezmi za výsledok zjednotenie σ_2 θ_1

napr. majme σ={x→y,z→f(u)} a θ={y→x,u→y,x→a}, potom
σ_1={x→x,z→f(y)}
θ_1={y→x,u→y}
σ_2={z→f(y)}
σθ=σ_2 θ_1={y→x,u→y,z→f(y)}
Nech t=f(x,g(y,z),u)
t{y→x,u→y,z→f(y)}=f(x,g(x,f(y)),y)
tσθ=(tσ)θ=f(y,g(y,f(u)),u)θ=f(x,g(x,f(y)),y)

Je prirodzenou požiadavkou aby substitúcia, ktorú nám vráti unifikačný algoritmus mala nejaké "rozumné" vlastnosti. Jednou z nich je aby substitúcia bola idempotentná.

Definícia 1.7. Substitúcia je *idempotentná* ak σσ=σ (teda Dom(σ) Vran(σ)=∅). Substitúcia, ktorá nie je idempotentná, je *neidempotentná.*

napr. substitúcia {x→f(y),z→g(x)} je neidempotentná, {x→f(y),z→g(f(y))} už idempotentná je.

Idempotentnosť nám zabezpečí, že premenná, ktorá sa vyskytuje na ľavej strane substitúcie, sa nevyskytne na jej pravej strane. Keby substitúcia túto vlastnosť nemala, mohli by sme sa pri jej aplikácii dostať do nekonečného cyklu.

Definícia 1.8. *Substitúcia premenovania premenných* je taká, že Dom(σ)=Ran(σ).

napr. {x→y, y→z, z→x} je subst. premenovania premenných ale {x→y, y→z} nie je.

Definícia 1.9. ak p={x_1→y_1,...,x_n→y_n} premenúva premenné tak p^{-1}={y_1→x_1,...,y_n→x_n} je ku nej *inverzná*.

Definícia 1.10. Dve substitúcie σ a θ sú *zhodné* (ozn. $s=q$) ak pre každý term x platí, že xσ=xθ.

Konečne sa dostávame ku pojmu unifikátor, čiže k substitúcii, ku ktorej smerujeme v každom unifikačnom algoritme.

Definícia 1.11. Substitúcia σ je *unifikátor temov s a t* ak sσ=tσ.

Definícia 1.12. Hovoríme, že substitúcia σ je *všeobecnejšia* ako substitúcia θ (ozn. $s£q$) ak existuje substitúcia δ taká, že θ=σδ. Takto definovaná relácia je kvázi usporiadaním.

kde ≤ je inštancia kvázi zoraďovania.

Definícia 1.13. Substitúcia σ je *najvšeobecnejší unifikátor* (*mgu*=most general unifier) termov s a t, ak pre každý unifikátor θ termov s a t platí σ≤θ.

Najvšeobecnejší unifikátor je ďalšou prirodzenou požiadavkou na unifikačný algoritmus. Takto dostaneme substitúciu, ktorá ma najmenší počet dvojíc zobrazení a aj najmenší počet premenných na ľavej strane, teda jej aplikácia je rýchla. Vyplýva to z toho, že na ľavej strane substitúcie sa vyskytnú iba premenné, ktoré je potrebné zmeniť v zúčastnených termoch, aby bola splnená podmienka unifikátora týchto dvoch termov. Keby sa v najvšeobecnejšom unifikátore vyskytovali aj dvojice, ktoré na ľavej strane majú premenné, ktoré nie je potrebné meniť, tak by existoval všeobecnejší unifikátor. S problémom najvšeobecnejšieho unifikátora sa budeme zaoberať neskôr pri unifikačných algoritmoch.

1.2. Prvé unifikačné algoritmy

Teraz, keď sme už zvládli základné pojmy môžeme pristúpiť k prvému a najjednoduchšiemu unifikačnému algoritmu. Vstupom do tohto algoritmu sú, podobne ako u všetkých ostatných, dva termy, o ktorých chceme rozhodnúť či sú unifikovateľné. Ak algoritmus zistí, že vstupné termy nie sú unifikovateľné, tak skončí chybou, inak vráti najvšeobecnejší unifikátor vstupných termov.

V tomto algoritme sa za symbol považujú aj symboly "(", ")", ",", používaných v zápise funkčných symbolov.

Algoritmus 1.2. Naivný unifikačný algoritmus

0. Nastav smerníky na začiatok oboch termov. Substitúciú σ inicializujeme na prázdnu
1. Posúvaj naraz smerníky po jednom symbole pokiaľ nedôjdeš na koniec ($\rightarrow$OK) alebo kým nenarazíš na rôzne symboly.
2. ak ani jeden symbol nie je premenná ($\rightarrow$CHYBA, koniec pre nezhodu symbolov), inak ak x je premenná (1. term) a t je začiatok podtermu (2. term) tak
   a) ak x $\in$ Vars(t) $\rightarrow$CHYBA, koniec
   b) inak substitúciu σ nahraď kompozíciou $\sigma\{x \rightarrow t\}$, zmeň každý výskyt x na t v oboch termoch a SKOK NA 1.

Otázky, ktoré sa nám môžu teraz vynoriť, sú asi tieto:

Implementácia

-Aké štruktúry môžu byť použité pre termy a substitúcie?
-Ako implementovať aplikácie substitúcií?
-V akom poradí môžu byť operácie vykonané?

Korektnosť

-Skončí algoritmus stále?
-Nájde stále najvšeobecnejší unifikátor?
-Odmietne všetky neunifikovateľné termy?
-Závisia odpovede na posledné dve otázky na poradí operácií?

Zložitosť

-priestorová aj časová

Teraz si uvedieme ďalší unifikačný algoritmus, ktorý na prvý pohľad nie je taký jednoduchý ako naivný. Využíva to, že term je v podstate stromová štruktúra, v ktorej sa venuje každému uzlu zvlášť ako samostatnému termu. Čísla na pravej strane si zatiaľ nevšímajte, vrátime sa k nim neskôr.

Algoritmus 1.3. Unifikácia rekurzívnym vnáraním

```
global σ:substitúcia; //inicializovaná na prázdnu

unifikuj ( s:term, t:term )
begin
  if s je premenná then s:=sσ;                                    {6.1}
  if t je premenná then t:=tσ;                                    {6.2}
  if (s je premenná) and (s=t) then                               {1}
    //nič
  else
    if (s=f(s1,...,sn)) and (t=g(t1,...,tm)) and (n,m≥0) then     {2.1}
      begin
       if f=g and n=m then                                        {2.2}
           for i:=1 to n do
                 unifikuj(si,ti);                                 {2.3}
       else
           ERROR; //nezhoda symbolov - unifikátor neexistuje      {3}
      end;
    else
      if s nie je premenná then
         unifikuj(t,s);                                          {4}
      else
        if t Vars(s) then
           ERROR; //obsahujú sa - unifikátor neexistuje          {5}
        else
           σ:=σ{s→t};                                            {6.3}
end;
```

Je potrebné poznamenať, že ak sa už raz dvojica x→t zapíše do substitúcie, tak x sa už nikdy nevyskytne v žiadnom terme, ktorý prejde algoritmom, teda x sa vo výslednej substitúcii vyskytuje len raz.

1.3. Unifikácia pomocou U-systému

V tejto časti budeme unifikovať termy pomocou nižšie definovaného U-systému. Využijeme jeho formálny základ na to, aby sme dokázali jeho korektnosť i úplnosť a s jeho pomocou aj korektnosť a úplnosť už spomínaného algoritmu rekurzívnym vnáraním. Opäť prejdeme niekoľkými definíciami.

Definícia 1.14. *Multimnožina* je nezoradená skupina prvkov s možným opakovaním prvkov.

Označenie 1.3. Počet výskytov prvku x v multimnožine M označujeme *M(x)*.

Definícia 1.15. Multimnožina Q je zjednotením multimnožín M a N (ozn. $Q=M \cup N$), ak pre každé x platí : Q(x)=M(x)+N(x).

To, že chceme zistiť unifikovateľnosť termov s a t, budeme v ďalšom označovať ako unifikačný problém termov s a t, a zapíšeme $s=^{?}t$.

Unifikačný problém si môžeme predstaviť aj ako otázku pre unifikačný algoritmus, teda otázku, či sú termy s a t unifikovateľné. Riešením unifikačného problému termov s a t je unifikátor týchto termov ak existuje, inak riešenie tohto unifikačného problému neexistuje.

Definícia 1.16. *U-systém* je usporiadaná dvojica $\langle P;\sigma\rangle$, kde P je multimnožina unifikačných problémov a σ je substitúcia. Definovaný je nasledujúcimi transformáciami:

1) ak $s \in Var : \langle\{s =^? s\}\ P;\sigma\rangle \rightarrow \langle P;\sigma\rangle$

2) $\langle\{f(s_1,\ldots,s_n) =^? f(t_1,\ldots,t_n)\}\ P;\sigma\rangle \rightarrow \langle\{s_1 =^? t_1,\ldots,\ s_n =^? t_n\}\ P;\sigma\rangle$

3) ak $f \neq g$ alebo $m \neq n$ pre $m,n \geq 0 : \langle\{f(s_1,\ldots,s_n) =^? g(t_1,\ldots,t_m)\}\ P;\sigma\rangle \rightarrow \perp$

4) ak $t \notin Var : \langle\{t =^? x\}\ P;\sigma\rangle \rightarrow \langle\{x =^? t\}\ P;\sigma\rangle$

5) ak $x \in Vars(t)$ a $x \neq t : \langle\{x =^? t\}\ P;\sigma\rangle \rightarrow \perp$

6) ak $x \in Var$ a $x \notin Vars(t) : \langle\{\{x =^? t\}\ P;\sigma\rangle \rightarrow \langle P\{x \rightarrow t\};\sigma\{x \rightarrow t\}\rangle$

kde, $\langle P;\sigma\rangle \rightarrow \perp$ znamená že neexistuje unifikátor, ktorý by riešil multimnožinu unifikačných problémov P, a $P\{x \rightarrow t\}$ znamená, že substitúcia $\{x \rightarrow t\}$ sa aplikuje na obe strany všetkých unifikačných problémov v P.

Ak chceme unifikovať s a t tak začíname U-systémom $\langle\{s =^? t\};\emptyset\rangle$. Ukážeme, že unifikácia skočí buď v $\perp$ alebo v U-systéme $\langle\emptyset;\sigma\rangle$, kde σ najvšeobecnejší unifikátor (mgu) termov s a t. Ukážeme, že tieto transformácie môžu simulovať algoritmus rekurzívneho vnárania.

Každý krok v algoritme rekurzívnym vnáraním sa dá simulovať pomocou transformácií na U-systéme.

rekurzívny alg.:		transformácie:	
$s_1\ t_1\ \emptyset$			$\{s_1 =^? t_1\};\emptyset$
$s_2\ t_2\ \sigma_2$			$\{s_2 =^? t_2\}\ P_2;\sigma_2$
$s_3\ t_3\ \sigma_3$			$\{s_3 =^? t_3\}\ P_3;\sigma_3$

V algoritme rekurzívneho vnárania v riadkoch {6.1} a {6.2} sa aplikuje substitúcia na vstupné termy s a t. Tento problém je v transformáciách U-systému vyriešený v šiestom pravidle, vykonaním aplikácie pridávanej dvojice do substitúcie σ na obe strany všetkých unifikačných problémov v multimnožine P: $P\{x \rightarrow t\}$. Riadok {1} v algoritme je v transformáciách riešený prvým pravidlom. Riadky {2.1},{2.2},{2.3} sú riešené druhým pravidlom. Podmienka na ľavej strane druhého pravidla je testovaná v riadkoch {2.1} a {2.2}. Vnáranie v riadku {2.3} je v druhom pravidle riešené pridaním tých istých unifikačných problémov, ktoré majú riešiť vyvolávané rekurzívne volania. Ľahko je vidieť, že algoritmus sa dostane k riadkom {3},{4},{5} iba vtedy keď sú splnené podmienky v treťom, štvrtom, respektívne piatom pravidle transformácií U-systému. Ak algoritmus skončí v ERROR tak transformácia končí v $\perp$. Podmienka šiesteho pravidla je splnená iba v riadku {6.3} a do substitúcie σ sa dostane v oboch prípadoch tá istá dvojica $\{s \rightarrow t\}$. Ak v U-systéme budeme vyhodnocovať stále pravidlo s unifikačným problémom, ktorý je na ľavej strane multimnožiny P, tak transformácie na U-systéme budú presne simulovať algoritmus rekurzívneho vnárania. Teda transformácie dávajú rovnaké výsledky ako rekurzívne vnáranie a môžu byť použité na dôkaz korektnosti algoritmu.

Lema 1.1. Pre každú konečnú multimnožinu rovností P každá postupnosť transformácií v U-systéme typu: $<P;\varnothing>\rightarrow<P_1,\sigma_1>\rightarrow\ldots$ končí buď v $\perp$ alebo v U-systéme $<\varnothing;\sigma>$, kde σ je substitúcia, v ktorej premenné na ľavej strane majú jediný výskyt v tejto substitúcii.

Dôkaz:
Každé pravidlo redukuje multimnožinu unifikačných problémov P. Každý unifikačný problém je jedného z typov na ľavej strane pravidiel U-systému. Teda jediné, čo nemá svoj typ pravidla, je $\perp$ alebo $<\varnothing;S>$. Ak sa nejaká dvojica $x\rightarrow t$ pridá do σ, tak premenná x na ľavej strane sa už nikdy nevystytne, lebo jej aplikáciou na unifikačné problémy multimnožiny P sa každý jej výskyt nahradí termom t. Jej kompozíciou so substitúciou σ sa premenná x odstráni aj z pravých strán v substitúcii. Z toho vyplýva, že aj pre každú substiúciu $\sigma_1,\sigma_2,\ldots,\sigma$ platí, že premenné na ľavých stranách majú jedinečné výskyty.

ù

Dôsledok 1.1. Ak existuje postupnosť transformácií na U-systéme v tvare $<P;\varnothing>\rightarrow\ldots\rightarrow<\varnothing;\sigma>$ tak σ je neidempotentná substitúcia.

Základom pre korektnosť algoritmu je nasledujúca ekvivalencia.

Lema 1.2. Pre ľubovoľnú transformáciu $<P_1;\sigma_1>\rightarrow<P_2;\sigma_2>$ platí, že substitúcia θ je unifikátor všetkých unifikačných problémov v multimnožine P_1 práve vtedy, keď θ je unifikátor všetkých unifikačných problémov v multimnožine P_2.

Dôkaz:
Musíme preveriť všetky prípady transformácií
1. Keďže $s\theta=s\theta$ platí pre ľubovoľnú substitúciu, tak to triviálne platí.
2. Platí: $f(s_1,\ldots,s_n)\theta=f(t_1,\ldots,t_n)\theta \Leftrightarrow f(s_1\theta,\ldots,s_n\theta)=f(t_1\theta,\ldots,t_n\theta) \Leftrightarrow s_1\theta=t_1\theta, \ldots, s_n\theta=t_n\theta$ čo chceme aby platilo na pravej strane.
3. Platí triviálne, keďže neexistuje substitúcia, ktorá by vedela premenovať funkčné symboly
4. Ak $t\theta=x\theta$ tak aj $x\theta=t\theta$.
5. Ak $x\in Vars(t)$ a $x\neq t$ tak určite neexistuje unifikátor θ taký, že $x\theta=t\theta$, keďže x obsahuje menej symbolov ako t. Substitúcia, ktorá by spôsobila, že $x\theta$ má o n symbolov viac ako x, by spôsobila, že $t\theta$ by mal aspoň o n symbolov viac ako t.
6. vieme, že $x\theta=t\theta$, potom pre ľubovoľný term u platí $u\theta=(u\{x\rightarrow t\})\theta$ a teda platí aj rovnosť: $P\theta=P\{x\rightarrow t\}\theta$.

ù

Prvým dôležitým výledkom je, že odvodzovaný U-systém naozaj vytvára unifikátor.

Veta 1.1. (korektnosť) Ak existuje postupnosť transformácií na U-systéme v tvare $<P;\varnothing>\rightarrow\ldots\rightarrow<\varnothing;\sigma>$, tak σ unifikuje všetky unifikačné problémy v multimnožine P.

Dôkaz:
Indukciou podľa lemy 1.2. je σ unifikátor všetkých unifikačných problémov v multimnožine P.

ù

Ďalší dôležitý výsledok je, že U-systém poskytuje najvšeobecnejší unifikátor.

Veta 1.2. (úplnosť) Ak θ unifikuje všetky unifikačné problémy v multimnožine P, potom ľubovoľná maximálna postupnosť transformácií $P;\emptyset\rightarrow\ldots$ musí končiť v nejakom systéme $\emptyset;\sigma$ takom, že $\sigma\leq\theta$.

Dôkaz:
Podľa lemy 1.1. a lemy 1.2. takáto postupnosť musí končiť v nejakom konečnom U-systéme $\emptyset;\sigma$. Keby táto postupnosť skončila $\ldots\rightarrow<P_n,\sigma>\rightarrow\perp$, tak θ nemôže unifikovať unifikačné problémy v multimnožine P_n. Ale podľa lemy 1.2. nemôže unifikovať ani problémy vo všetkých predchádzajúcich $P_{n-1},P_{n-2},\ldots,P$ čo je spor.
Ešte musíme dokázať, že σ je najvšeobecnejší unifikátor všetkých problémov v multimnožine P. Pre každé priradenie $x\rightarrow t$ v substitúcii σ podľa lemy 1.2. platí, $x\sigma\theta=t\theta=x\theta$. Pre každé $x\notin Dom(\sigma)$ platí, $x\sigma_S\theta=x\theta$ teda $\theta=\sigma_S\theta$.

ù

Priamym dôsledkom je predchádzajúcich viet je:

Dôsledok 1.2. Ak P nemá unifikátor potom ľubovoľná maximálna postupnosť transformácií z $P;\emptyset$ musí končiť v $\perp$.

Teda každý unifikačný algoritmus, ktorý vykonáva jednotlivé akcie U-systému v ľubovoľnom poradí je korektný a úplný a generuje idempotentné najvšeobecnejšie unifikátory pre unifikovateľné termy. Niektoré môžu počítať dlhšie, niektoré kratšie, môžu generovať väčšie termy a nekončia vždy v pevnom najvšeobecnejšom unifikátore. Najvšeobecnejšie unifikátory sú jednoznačné až na premenovanie premenných.

<u>Zložitosť algoritmu rekurzívneho vnárania:</u>

Pre tento algoritmus je časová aj priestorová zložitosť exponenciálna. napr. pri unifikácii týchto termov:

$h(x_1, \ldots,x_n, f(y_0,y_0),\ldots,f(y_{n-1},y_{n-1}),y_n)$
$h(f(x_0,x_0),\ldots,f(x_{n-1},x_{n-1}),y_1, \ldots,y_n, x_n)$

Unifikovaním týchto dvoch termov vzniká najvšeobecnejší unifikátor, v ktorom je ku každému x_i a y_i priradený term s 2^{i+1}-1 symbolmi. Najvšeobecnejší unifikátor obsahuje veľké množstvo kópií rovnakých podtermov.

<u>napr.</u> Pre n=3 je najvšeobecnejší unifikátor σ={x0→y0; x1→f(y0,y0);
x2→f(f(y0,y0),f(y0,y0)); x3→f(f(f(y0,y0),f(y0,y0)),f(f(y0,y0),f(y0,y0)));
y1→f(y0,y0); y2→f(f(y0,y0),f(y0,y0));
y3→f(f(f(y0,y0),f(y0,y0)),f(f(y0,y0),f(y0,y0)));}

Pri volaní funkcie unifikuj pre posledné 2 argumenty x_n a y_n, ktoré sú týmto viazané k termom s 2^{n+1}-1 symbolmi, dôjde k exponenciálnemu počtu rekurzívnych volaní.

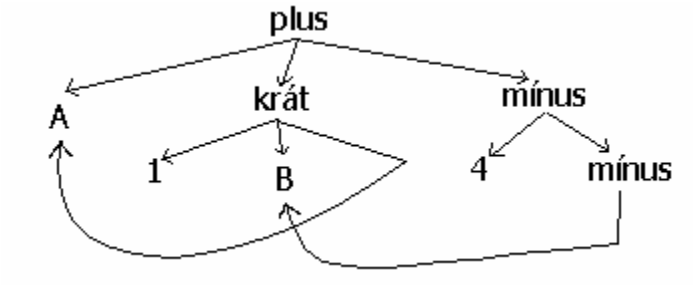
Riešením tohto problému je vytvorenie šikovnejších dátových štruktúr pre termy a inou metódou na aplikovanie substitúcií.

1.4. Unifikácia grafov termov

Pekným príkladom šikovnejších štruktúr na uchovávanie termov je použitie grafov termov.

Definícia 1.19. *Graf termu* (*dag*) je orientovaný acyklický graf, ktorého vrcholy sú funkčné symboly alebo premenné. Hrany vychádzajúce z ľubovoľného vrchola do jeho podtermov sú zoradené v rovnakom poradí, ako jeho podtermy a s rovnakým počtom ako jeho árnosť.

napr. toto je príklad na reprezentáciu termu plus(A,krát(1,B,A),mínus(4,mínus(B))) grafom termu (dagom) so zdieľaným výskytom premenných



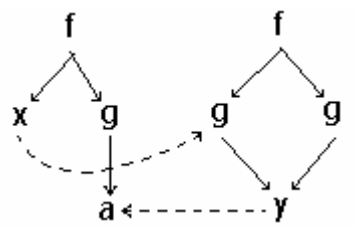
V ďalšom predpokladáme, že vstupom pre unifikačný algoritmus je graf 2 termov, ktoré treba unifikovať, s jediným zdieľaným výskytom každej premennej.

1.4.1. Rekurzia na grafoch termov

Tento unifikačný algoritmus používam v praktickej časti mojej diplomovej práce.

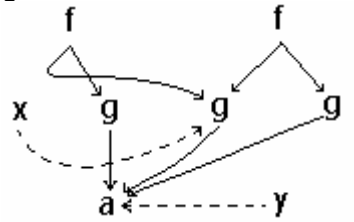
Označenie *rodičia(t)* bude predstavovať množinu funkčných symbolov z ktorých ide hrana grafu termu do termu t. Substitúcia bude reprezentovaná priamo reláciou na vrcholoch grafu ako zoznam dvojíc vrcholov spojených orientovanou hranou.

napr. termy f(x,g(a)) a f(g(y),g(y)) a substitúcia {x→g(y),y→a}



V tomto prípade rodičia(a)={g}.

napr. toto reprezentuje typický stav po skončení algoritmu rekurzie na grafoch termov. Môžeme vidieť najvšeobecnejší unifikátor {x→g(a),y→a}.



V tomto prípade rodičia(a)={g,g,g}, no z obrázku je jasné, že každý z funkčných symbolov g je samostatný vrchol, takže správnejší zápis by mal byť rodičia(a)={$g^{(0)}$,$g^{(1)}$,$g^{(2)}$}, kde v hornom indexe je akési identifikačné číslo vrchola v grafe. Prejdime teraz k samotnému algoritmu.

Algoritmus 1.4. Rekurzia na grafoch termov

```
global Δ:termDag      {vstup}
global σ:zoznam hrán  {inicializované na ∅}

nahraď(Δ,s,t)
 begin
   ak rodičia(s)={p1,...,pn} potom
    1, ∀pi nahradiť hranu pi→s za pi→t;
    2, rodičia(t) := rodičia(s)   rodičia(t);
    3, rodičia(s) := ∅;
 end;

UnifyDag( s:vrchol, t:vrchol )
 begin
  if s=t then
    //nič
  else
    if s=f(s1,...,sn) & t=g(t1,...tm) pre n,m≥0 then
      begin
        if f=g & n=m then
          for i:=1 to n to
              UnifyDag(si,ti);
        else
          Error("nezhoda symbolov");
      end;
    else
      if s∉Var then
        UnifyDag(t,s);
      else
          if s Vars(t) then
            Error("obsahujú sa");
          else
            begin
             Add(s,t) na koniec σ;
             Δ := Nahraď(Δ,s,t);      {už sú unifikované}
            end;
 end;
```

Výskyt premennej s v terme t je implementovaný ako štandardný prechod grafom na vyhľadanie daného vrchola s v podterme t. Funkcia add(s,t) pridá novú orientovanú hranu do zoznamu dvojíc substitúcie σ. Zjavne platí nasledujúca lema.

Lema 1.3. Nech Δ je graf termov obsahujúci vrcholy x,t také, že neexistuje cesta z t do x (teda $x \notin Vars(t)$). Potom
1. Nahraď(Δ,x,t) je acyklický graf obsahujúci tie isté vrcholy ako Δ.
2. Uvažujme pevný vrchol v Δ zodpovedajúci termu s, nech s' je term zodpovedajúci tomu istému vrcholu v Nahraď(Δ,x,t). Potom
   a. ak s=x tak s'=x
   b. inak $s'=s\{x \rightarrow t\}$

Na dôkaz *korektnosti* a *úplnosti* môžeme opäť ukázať, že tento algoritmus dokáže simulovať U-systém. Z logického hľadiska sa nič nestalo, iba sa zmenili dátové štruktúry v pozadí (termy, substitúcie).

Zložitosť

Naznačíme dôkaz zložitosti pre tento algoritmus. Algoritmus izoluje vrchol po každom volaní funkcie čo je maximálne O(n) krát. (n=počet vrcholov v pôvodných termoch). Každé volanie robí rovnaké množstvo práce až na výskyt v podstrome (čo prejde max. n vrcholov). Udržiavanie rodičov stojí O(n) v každom volaní, konštrukcia grafu stojí O(n) teda časová zložitosť je $O(n^2)$ a priestorová je O(n).

1.4.2. Takmer lineárny algoritmus

Takmer lineárny algoritmus je ďalším veľmi rýchlym unifikačným algoritmom. Namiesto rekurzívnych volaní pre dvojice podtermov, ktoré majú byť unifikované, prepracujeme problém na konštrukciu relácie unifikácie, ktorej triedy sú tvorené termami, ktoré majú byť unifikované. Substitúcia bude nahradená spojením tried ekvivalncie. Opakované volania pre hľadanie výskytov v podtermoch budú nahradené prechodom triedami na overenie acyklickosti.

Definícia 1.20. Relácia @ na termoch je *relácia unifikácie*, ak
1. je to relácia ekvivalencie
2. je acyklická, teda ak $t \cong s$ tak s nie je podtermom t
3. je homogénna, teda ak $f(t_1,..,t_n) \cong g(s_1,..,s_m)$; $m,n \geq 0$, tak f=g (a teda n=m)
4. spĺňa unifikačnú axiómu: Pre každý n-árny funkčný symbol f a pre všetky termy s_i,t_i, pre i=1..n, platí:
   $f(s_1,...,s_n) \cong f(t_1,...,t_n) \rightarrow s_1 \cong t_1 \& ... \& s_n \cong t_n$.

Definícia 1.21. *Unifikačný uzáver termov s a t* je najmenšia relácia unifikácie, v ktorej sú termy s a t ekvivalentné.

Definícia 1.22. Majme nejakú reláciu unifikácie ≅. Pre ľubovoľný term t, *[t]* označuje triedu relácie unifikácie ≅ obsahujúcu t, teda $[t]=\{s \in Term: t \cong s\}$.

napr. [x]={f(a,y),z,f(a,b),x} môže byť triedou relácie unifikácie.

Na dôkaz základnej vety ešte budeme potrebovať dve definície.

Definícia 1.23. Majme nejakú reláciu unifikácie $\cong$.
Definujme výberovú funkciu z tried relácie unifikácie $\cong$ do termov takú, že pre ľubovoľný term t platí:
1. $([t]) \in [t]$
2. $([t]) \in Var$ iba vtedy, keď $[t] \subseteq Var$.

Term $([t])$ nazveme schéma termu pre triedu [t].

napr. Ak [x]={f(a,y),z,f(a,b),x} je triedou relácie unifikácie, tak $([t])$ môže byť buď f(a,y) alebo f(a,b). Premenné x a z neprichádzajú do úvahy, pretože sa v triede nachádzajú funkčné symboly.

Definícia 1.24. Nech $\cong$ je ľubovoľný unifikačný uzáver a nech je lubovoľná výberová funkcia z tried unifikácie $\cong$. Substitúcia $\sigma_{\cong}$ je definovaná nasledovne:
Pre každý term x, platí

$$x s_{@} = \begin{cases} y & \text{ak } ([x])=y, y \in Var \\ f(s_1\sigma_{\cong},\ldots s_n\sigma_{\cong}) & \text{ak } ([x])=f(s_1,\ldots,s_n), n \geq 0 \end{cases}$$

Veta 1.3. Termy s a t sú unifikovateľné $\Leftrightarrow \exists$ unifikačný uzáver termov s a t. Naviac ak ekvivalencia platí, tak $\sigma_{\cong}$ je najvšeobecnejší unifikátor termov s a t.

Dôkaz:
Ü**:**
Ukážeme, že ak $\cong$ je unifikačný uzáver termov s a t, tak $\sigma_{\cong}$ unifikuje termy s a t, teda $s\sigma_{\cong}=t\sigma_{\cong}$. Zrejme stačí ukázať, že pre ľubovoľný term u platí, že $u\sigma_{\cong}= ([u])\sigma_{\cong}$. Inými slovami, že $\sigma_{\cong}$ unifikuje každú dvojicu prislúchajúcich podtermov v konkrétnych termoch s a t. Ideme indukciou podľa štruktúry termu u:
1^{o} Ak u je konštanta, tak zrejme musí byť aj schémou termu pre triedu [u], teda rovnosť platí triviálne. Ak u je premenná, aj $([u])$ je premenná tak to platí triviálne. Ak u je premenná a $([u])=f(t_1,\ldots,t_n)$, tak $u\sigma_{\cong} = f(t_1\sigma_{\cong},\ldots,t_n\sigma_{\cong}) = f(t_1,\ldots,t_n)\sigma_{\cong} = ([u])\sigma_{\cong}$.
2^{o} Predpokladajme, že $u=f(s_1,\ldots,s_n)$, $([u])=f(t_1,\ldots,t_n)$. Keďže $\cong$ musí spĺňať unifikačnú axiómu tak pre každé $i \in \{1\ldots n\}$ platí: $s_i \cong t_i$. Potom podľa indukčného predpokladu platí $s_i\sigma_{\cong}=t_i\sigma_{\cong}$. Teda môžeme písať : $u\sigma_{\cong} = f(s_1,\ldots,s_n)\sigma_{\cong} = f(s_1\sigma_{\cong},\ldots,s_n\sigma_{\cong}) = f(t_1\sigma_{\cong},\ldots,t_n\sigma_{\cong}) = f(t_1,\ldots,t_n)\sigma_{\cong} = ([u])\sigma_{\cong}$

Þ**:**
Pre ľubovoľný unifikátor θ termov s a t definujeme reláciu $\cong_{\theta}$ nasledovne: ak u a v sú ľubovoľné termy, tak $u \cong_{\theta} v$ práve vtedy, keď $u\theta=v\theta$. Zjavne to je relácia unifikácie. Keďže existuje nejaká relácia unifikácie, tak určite existuje najmenšia relácia unifikácie $\cong$. Teda ak s a t sú unifikovateľné tak existuje unifikačný uzáver $\cong$ termov s a t.

mgu:
Nech θ je ľubovoľný unifikátor termov s a t. Nech $\cong$ je ľubovoľný unifikačný uzáver obsiahnutý v relácii $\cong_\theta$. Vieme, že pre lubovoľné termy u a v platí, že ak $u \cong v$ tak $u\theta = v\theta$. Ďalej, nech je ľubovoľná pevná výberová funkcia pre substitúciu $\sigma_{\cong}$. Ukážeme, že pre ľubovoľný term u platí, že $u\sigma_{\cong}\theta = u\theta$. Dokážeme to indukciou podľa veľkosti u.
1^o ak trieda [u] obsahuje len konštanty a premenné tak platí, že $u\sigma_{\cong} = ([u]) \cong u$, z čoho vyplýva, že $u\sigma_{\cong}\theta = u\theta$.
2^o V prípade, že ([u]) sa rovná nejakému $f(s_1,...,s_n)$ a u je term typu $f(t_1,...,t_n)$, tak $u\sigma_{\cong} = f(s_1\sigma_{\cong},...,s_n\sigma_{\cong})$ a $u\theta = f(s_1,...,s_n)\theta$. Môžeme teda napísať:

$$u\theta = f(s_1\theta,...,s_n\theta) = f(s_1\sigma_{\cong}\theta,...,s_n\sigma_{\cong}\theta) = f(s_1\sigma_{\cong},...,s_n\sigma_{\cong})\theta = u\sigma_{\cong}\theta,$$

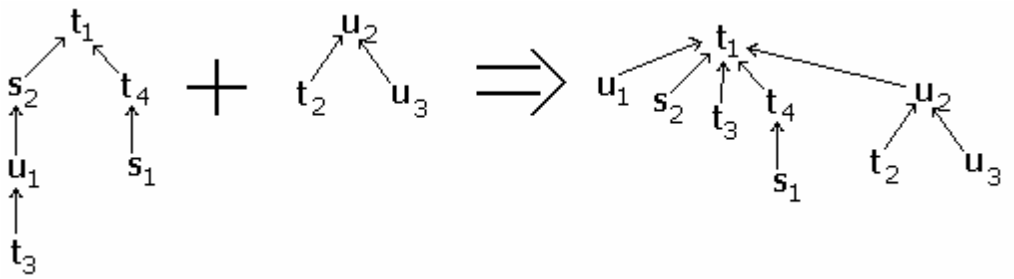
kde druhá rovnosť vyplýva z indukčného predpokladu.

ù

V ďalšom je nevyhnutné mať prostriedok na uchovávanie tried relácií unifikácie a na použitie unifikačnej axiómy na triedach. Asi najschopnejšou štruktúrou reprezentácie tried sú stromy triedových vzťahov, kde koreň je reprezentant triedy, čiže vrchol stromu zodpovedajúci schéme termu pre danú triedu. Na určenie toho, či sú dva termy v relácii unifikácie, stačí nájsť korene a preveriť to na nich. Pri spájaní dvoch tried sa jedna trieda stane podstromom druhej. Kôli zmenšeniu výšky stromov použijeme jemné zlepšenia:

(i) v koreni budeme uchovávať veľkosti triedy a pri spájaní napájať menší podstrom pod väčší. (vykonávané v procedúre Spojenie nižšie uvádzaného algoritmu)
(ii) ak pri preverovaní toho, či sú dva termy v relácii unifikácie, prejdeme nejakou vetvou do koreňa, tak všetky vrcholy na tejto ceste napojíme priamo na koreň. (vykonávané vo funkcii Nájdi nižšie uvádzaného algoritmu)

napr.



Takže v ďalšom budeme potrebovať:
- triedové stromy (odkazy vrámci triedy)
- počítadlo veľkosti triedy v koreni (reprezentatovi) triedového stromu (premenná veľkost(t) pre vrchol t)
- odkaz z každého koreňa (reprezentanta) triedy do schémy termu v triede
  (premenná ([t]) pre každý koreň (reprezentanta)
- boolean premenné "navštívený()" a "acyklický()" v každom vrchole, inicializované na false
- koreň je vrchol, ktorého triedový odkaz ukazuje na seba samého

Algoritmus 1.5. Takmer lineárny algoritmus unifikácie

Inicializácia:
Každá trieda obsahuje iba 1 term, pre každý term odkazy schémy a triedy ukazujú na seba samého, veľkosť je 1. Zoznam premenných je buď prázdny alebo ak vrchol je premenná tak je tam iba táto premenná. Pre každý vrchol premenné navštívený a acyklický sú false.

```
global Δ: TermDag;
global σ: zoznam spojení :=nil; {riešenie}

Unifikuj ( s:vrchol; t:vrchol ) //vracia unifikátor σ
 begin
   UnifUzáver(s,t);
   Riešenie(s);
 end;

UnifUzáver( s:vrchol; t:vrchol )
 begin
  s:=Nájdi(s);  {nájde reprezentantov}
  t:=Nájdi(t);
  if s=t then
     //nič
  else
    begin
      if  ([s])=f(s1,...,sm) &  ([t])=g(t1,...,tn) pre m,n≥0 then
        begin
          if f=g then
            begin
              Spojenie(s,t);
              for i:=1 to n do
                 UnifUzáver(si,ti);
            end;
          else
             Error("nezhoda symbolov");
        end;
      else
        Spojenie(s,t);
    end;
 end;

Nájdi(s:vrchol) {vráti reprezentanta a redukuje cesty}
 t:vrchol;
 begin
  if trieda(s)=s then {s je reprezentant}
     return s;
  else
   begin
     t:=nájdi(trieda(s));
     trieda(s):=t;
     return t;
   end;
 end;
```

```
Spojenie(s:vrchol; t:vrchol) {s a t sú reprezentanti}
 begin
  if veľkosť(s)≥veľkosť(t) then
   begin
    veľkosť(s):=veľkosť(s)+veľkosť(t);
    premenné(s):=premenné(s) premenné(t);
    if  ([s]) Var then
       ([s]):= ([t]);
    trieda(t):=s;
   end;
  else
   begin
    veľkosť(t):=veľkosť(t)+veľkosť(s);
    premenné(t):=premenné(t) premenné(s);
    if  ([t]) Var then
       ([t]):= ([s]);
    trieda(s):=t;
   end;
 end;

Riešenie(s:vrchol) {padne ak ∃ cyklus obsahujúci s}
 begin
  s:= (Nájdi(s));
  if acyklický(s) then
     return;
  if prejdený(s) then
     Error("cyklus");
  if s=f(s1,...,sn) pre nejaké n>0 then
     begin
       prejdený(y):=true;
       for i:=1..n do
          Riešenie(si);
       prejdený(s):=true;
     end;
  acyklický(s):=true;
  Pre každé x premenné(Nájdi(s))
    if x≠s then
       σ:=σ{x→s};
 end;
```

Ak funkcia Unifikuj(s,t) nepadne, potom σ obsahuje navšeobecnejší unifikátor termov s a t. Preverenie korektnosti spočíva v overení toho, že správne implementuje konštrukciu uzáveru unifikácie. Pri tomto preverovaní si treba všímať tieto body:

- ekvivalencia je čisto homogénna
- triedy ekvivalencie sú spojené práve vtedy, keď sú vyžadované unifikačnou axiómou, čiže relácia je minimálna
- funkcia Riešenie() padne práve vtedy, keď je v grafe cyklus

Zložitosť

Každá funkcia až na Nájdi() je volaná maximálne n krát, pre termy s n symbolmi a každá vykoná konštantne dlhý výpočet. Spájanie polí premenných môže trvať O(n), ak sú realizované dynamicky. Dominantnou cenou je volanie Nájdi(), ktoré potrebuje čas $O(n.\alpha(n))$.

V tejto časti sme sa venovali problému výberu správneho algoritmu na unifikáciu dvoch termov. Pozreli sme sa aj na otázky korektnosti a úplnosti týchto algoritmov. Niečo sme si povedali aj o ich zložitosti. V ďalšej kapitole sa budeme venovať problému efektívneho ukladania množiny termov a jej indexácii. Bude nás teda zaujímať problém unifikácie a iných návratových podmienok pri vstupe viac ako dvoch termov.

2.Indexovanie temov

2.1. Formulácia problému

Definícia 2.1. (Problém indexovania termov)
Majme *množinu indexovaných termov* a binárnu reláciu R nad množinou termov (*návratová podmienka*) a term t (*dopytovací (query) term*). Hľadáme množinu M, $M \subseteq L$ množinu všetkých termov s spĺňajúcich R(s,t).

V niektorých aplikáciách stačí hľadať nadmnožinu M ktorá obsahuje aj termy s ktoré nespĺňajú R(s,t), ale môžme prirodzene minimalizovať počet nájdených termov s cieľom zvýšiť efektivitu indexovania. V niektorých aplikáciách je prípustné vrátiť iba nejaké R-kompatibilné termy teda podmnožinu M. Ak je množina nájdených termov zaručene zhodná s množinou R-kompatibilných termov tak indexovacia technika sa nazýva *perfektné filtrovanie* inak je to *nedokonalé filtrovanie*.

V súvislosti s indexovaním termov je bežný prípad, že návratová podmienka R je splnená ak $\exists$ substitúcie σ a θ také že $s\sigma=t\theta$ a okrem toho tieto substitúcie spĺňajú určité ďalšie podmienky. Napr. ak σ a θ majú podmienku, že sú to premenovávacie substitcie, tak R(s,t) jednoducho vracia nejakú obmenu otázky. Podobne ak θ má byť prázdna substitúcia tak R(s,t) vracia inštanciu otázky. Okrem identifikovania R-kompatibilných termov, niekedy potrebujeme vypočítať aj substitúcie σ a θ.

Základné parametre prislúchajúce termom sú
- *návratová podmienka* - vyjadrená reláciou R ktorá určuje podmnožinu indexovaných termov M. Najčastejší prípad návratovej podmienky je unifikácia, spájanie,...
- *spôsob návratu* - určuje či je vrátená celá množina M alebo či sú to prvky tejto množiny vrátené postupne. V niektorých prípadoch sa zaujímame len o neprázdnosť kandidátskej množiny.

Základné typy indexovania sú
§ *Indexovanie založené na atribútoch*

V tomto indexovaní zachytávame niektoré črty termu T do celočíselného atribútu a_t. Indexovanie je potom založené na identifikácii relácie R_A na atribútoch termov t a s: $R(s,t) \rightarrow R_A(a_s,a_t)$ [$R_A(s,t) \rightarrow R(a_s,a_t)$]. Napr. ak návratová relácia je inštancia tak atribút môže byť počet funkčných symbolov v terme. Ak t je inštancia s tak počet funkčných symbolov v t je väčší alebo rovný počtu v s. Budeme uvažovať niekoľko nasledujúcich príkladov indexovania založeného na atribútoch:
- predskúšanie spojenia (matching pretest) - využíva fakt, že na to aby term t bol inštanciou s, je nutné aby počet symbolov v t bol $\geq$ ako počet symbolov v s.
- indexovanie čŕt (outline indexing) - využíva fakt, že t a s sú unifikovateľné iba ak súhlasia na všetkých miestach kde nie sú premenné. Používa bitový vektor na kódovanie miest bez premenných a zodpovedajúcich symbolov
- superuložené kľúčové slová (superimposed codewords) - atribút sa získava pomocou logických operácií or na bitových reprezentáciách funkčných symbolov na určitých pozíciách vnútri termu.

Indexovanie založené na atribútoch je založené na predpoklade, že relácia na číselných atribútoch sa ľahšie počíta ako vykonávanie spájania alebo unifikácie. Môže byť použité na hrubé filtrovanie, ale aj tak má niekoľko nevýhod. Správnosť tohoto indexovania je typicky nízka. Ak množina indexov je veľká, tak hrubý filter môže byť stále rovnako neefektívny ako testovanie relácie R_A na každý term v množine.

§ Indexovanie založené na funkčných symboloch

Návratová podmienka je typicky založená na identifikácii unifikačnej substitúcie medzi dopytovacím termom a indexovanými termami s rôznymi podmienkami pre substitúcie. Teda otázka či návratová podmienka medzi dopytovacím termom a indexovaným termom je určená funkčnými symbolmi oboch termov. Na pozorovaní, že oba termy sa musia zhodovať vo funkčných symboloch je založených najviac indexovacích techník.

2.2. Základné definície

Symboly v terme pochádzajú z neprázdnej abecedy Σ a spočítateľnej množiny premenných V.

Term sa nazýva *základný* (*ground*) ak neobsahuje žiaden výskyt premennej.

Každému symbolu s Σ prináleží nezáporné číslo nazvané *árnosť(s)*. Predpokladáme že všetky termy sú dobre vytvorené teda že každý term má rovnaký počet atribútov ako daný funkčný symbol.

Na označenie symbolov používame a,b,c,f,..., na označenie premenných x,y,z. Budeme používať aj divoký symbol * na označenie premenných, a ? na označenie ľubovoľných funkčných symbolov iných ako určitá množina funkčných symbolov.

Definícia 2.2. *Pozícia* je alebo prázdny reťazec L alebo *p.i* kde p je pozícia a i je číslo. Predstava pozície v terme a podterme t a pozície p (ozn. *t/p*) je def. nasledovne:

- Λ je pozícia v t & $t/\Lambda=t$
- Ak $t/p=f(t_1,...,t_n)$, $n>0$ tak p.1,...,p.n sú pozície v t, $t/p.i=t_i$ $\forall i \in \{1,...,n\}$

Namiesto Λ.i píšeme i. Označíme *P* množinu všetkých pozícií. *P(t)* označuje všetky pozície v terme t. $P_v(t)$ a $P_f(t)$ označuje podmnožiny tých pozícií, v ktorých má t premenné resp. funkčné syboly. Množina $P_v(t)$ sa nazáva *obal (fringe)* termu t. $t[s]_p$ označuje term získaný z t premenovaním t/p termom s.

napr. $t=f(a(x),b(a(y),c))$ $t/\Lambda=t$ $t/2=b(a(y),c)$ $t/2.1=a(y)$ $t/2.2=c$ $t[c]_2=f(a(x),c)$ $obal(t)=\{1.1,2.1.1\}$

Substitúcia je zobrazenie z premenných do termov. Majme substitúciu θ, tq je term ktorý vznikol z t premenovaním $x \rightarrow \theta(x)$, $x \in Vat(t)$.

Definícia 2.3. Hovoríme, že *t je inštancia u*, ak $u\theta=t$ pre nejakú substitúciu θ. Ak t je inštancia u tak píšeme $u \pounds t$ a povieme *u je prefix t*.

Definícia 2.4. Termy t a s nazývame *unifikovateľné* ak ∃ substitúcia taká, že sθ=tθ.

Definícia 2.5. *{x_1 ® t_1,...,x_n ® t_n}* je taká substitúcia θ, že:

$$\theta(x) = \begin{cases} t_i & \text{ak } x=x_i \\ x & \text{inak.} \end{cases}$$

napr. t=f(a(x),b(y,z)), θ={x→b(x',x''), y→c} potom tθ=f(a(b(x',x'')),b(c,z))

2.3. Operácie indexovania termov

Návrat kandidátskych termov

Majme dopytovací term t a indexovanú množinu L, *návratová operácia* je proces identifikácie podmnožiny M tých termov z L, ktoré sú v relácii R s t. Návratová relácia R identifikuje tie termy l L, ktoré treba vybrať. Toto sú niektoré relácie, ktoré sa tým zaoberajú:

unif(l,t) ⇔ ∃σ: lσ=tσ
(logické programovanie, deduktívne databázy,...)

inst(l,t) ⇔ ∃σ: l=tσ
(tabelované log. programovanie, deduktívne DB, dokazovače viet,...)

gen(l,t) ⇔ ∃σ: lσ=t
(funkcionálne programovanie, optimalizácia v constrainovom programovaní, spätná kontrola dôkazov viet)

var(l,t) ⇔ ∃σ: lσ=t & σ premenuváva premenné
(tabelované log. programovanie, deduktívne DB,...)

Konštrukcia indexu

Konštrukcia indexu sa zaoberá počiatočnou konštrukciou tejto dátovej štruktúry pre danú operáciu R a indexovanú množinu L. Po počiatočnej konštrukcii môžme chcieť zmeniť indexovanú množinu pomocou vkladania alebo mazania termov.

Udržiavanie indexu

Udrižiavanie indexu začína s indexom pre množinu L a vytvára index pre ďalšiu množinu L' ktorá vznikla vložením alebo vymazaním termu z L.

Rôzne typy indexovacích techník typicky vyjadrujú rôzne "dohody" medzi cenou vykonávania každej z 3 úloh (návrat, konštrukcia indexu, udržiavania indexu). Napr. vo funkcionálnom a logickom programovaní je indexovaná množina v podstate nemenná a tak netreba udržiavanie indexu, naviac index je konštruovaný počas kompilácie. Teda indexovacie techniky sa zoberajú optimalizáciou návratového času, prípadne aj na úkor zvýšenej ceny konštrukcie a udržiavania indexu. V iných aplikáciách napr. v tabelovanom log. programovaní je vkladanie indexu časté a mazanie sa nevyskytuje, inde napr. pri dokazovaní viet sa indexovaná množina mení stále.

2.3.1. Variácie operácií indexovania termov

Nelineárnosť

Vo všeobecnosti dopytovací term a indexované termy môžu byť nelineárne, teda môžu obsahovať opakované výskyty premenných. Viacnásobné výskyty sa môžu vyskytovať vnútri jedného zo zúčastnených termov alebo jedna premenná sa môže vyskytovať v oboch termoch. V tomto prípade je potrebné preverovať konzistentosť medzi substitúciami získanými viacnásobným výskytom tej istej premennej. Zisťovanie konzistetnosti je typicky drahá operácia a pokiaľ sa s ňou nezaobchádza opatrne, nastáva nelineárnosť počas indexácie, čo spôsobí zníženie výkonu. Teda, veľa techník ignoruje nelineárnosť pri indexovaní a ráta s posledným krokom ktorý odstráni nekonzistetnosť. Iné metódy odkladajú odstraňovanie nekonzistetnosti za najmenej drahé operácie a proste konštruujú štruktúru termu. V týchto prípadoch predpokladáme, že žiadna premenná z indexovanej množiny sa nezhoduje s premennou v dopytovacom terme.

Teórie rovností (Equational theories)

Návratová podmienka môže byť založená na teórii rovností E, v kt. dopytovací term t je inštanciou termu z množiny L s rešpektovaním E, čiže ak existuje term l L a substitúcia θ taká, že E├ lθ=t. Napr. v prípade automatizovanej dedukcie (automated reasoning) sa zaujímame o spájanie a unifikáciu za prítomnosti asociatívno-komutatívnych operátorov. V lazy funkcionálnom programovaní sa zaujímame o spájanie v súvislosti s teóriou rovnosti danou programom.

Priority

V mnohých prípadoch sú termy v indexovanej množine prepojené s prioritami, ktoré je potrebné rešpektovať. V niektorých prípadoch napr. vo funkcionálnom programovaní sa môžme zaujímať iba o návrat prvkov s najväčšou prioritou. napr. *R(l,t) & "l' L (priorita(l') > priorita(l) ® ØR(l',t)* V iných prípadoch (log. programovanie, deduktívne DB) môžmeme požadovať návrat prvkov v neklesajúcom poradí priority. V automatizovanej dedukcii môžme požadovať generovanie najprv "jednoduchšej", "všeobecnejšej", "použiteľnejšej" teórie.

Počítanie substitúcií

Niekedy môžme požadovať identifikáciu substitúcií ktorou dopytovací term a kandidátsky term vyhovujú návratovaej podmienke. Teoreticky výpočet substitúcií môže byť vykonávaný po indexovaní ale tento prístup zvyšuje poprocesnú cenu po nájdení kandidátskych termov. Preto veľa indexovacích techník počíta substitúcie už ako časť indexovacej operácie a vracia ich. Tento po častiach vracajúci spôsob je obzvlášť vhodný pre vyhľadávacie techniky.

Operácie many-to-many

Tieto indexovacie problémy vznikajú v súvislosti s operáciami ktoré sú vykonávané spoločne na skupinách termov. Najčastejšie príklady sú subsumption, hyperresolution a unit-resulting resolution.

Návratové podmienky pre podtermy

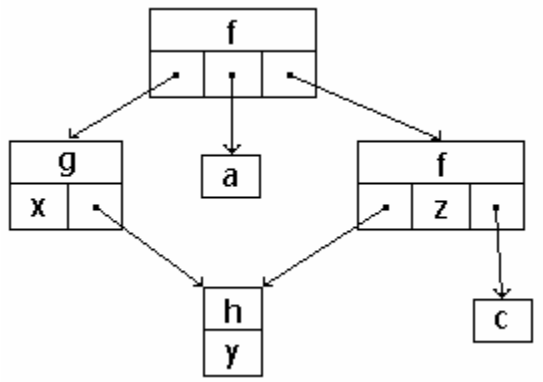
Niekedy sa môžeme zaujímať o všetky podtermy dopytovacieho termu ako o samostatné dopytovacie termy. Napr. pri prepisovaní termov chceme určiť indexovaný term l (zodpovedajúci prepisovaciemu pravidlu) a podterm t/p daného termu tak, že t/p je inštanciou l. podobne môžme chcieť oindexovať všetky podtermy danej množiny indexovaných termov.

2.4. Dátové štruktúry na reprezentáciu termov a indexov

2.4.1. Reprezentácia termov

Najprehľadnejšia reprezentácia termov je použitím stromov alebo **orientovaných acyklických grafov (DAG).** Hlavný vrchol stromovej reprezentácie termu t obsahuje hlavný funkčný symbol t a smerníky do vrcholov ktoré ukazujú na priame podtermy t. Tieto smerníky môžu byť uložené v poli alebo ako dynamický zoznam. Obvykle je reprezentácia univerzálna a umožňuje bežné operácie ako prechádzanie termom rôznymi cestami, prechádzanie podtermami,...

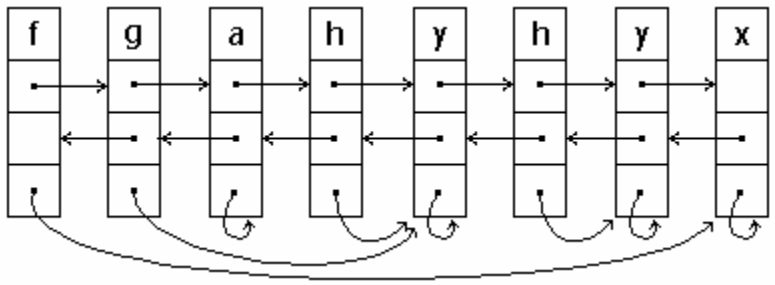
napr. DAG reprezentácia termu f(g(x,h(y)),a,f(h(y),z,c))



Na porovnanie so stromovou reprezentáciou, DAG umožňuje zdieľanie podtermov. Takéto zdieľanie môže prispieť až k exponenciálnemu zníženiu veľkosti termu. Úžitky zdieľania sú významné v praxi. Napr. pri prepisovaní termu a funkcionálnom programovaní pravidlá typu $f(x) \rightarrow g(x,x)$ sa vyskytujú bežne, čo pri stromoch môže vyžadovať zdvojenie termu. Naproti tomu použitie DAG reprezentácie umožňuje dosiahnuť ten istý efekt zdvojením smerníka na substitúciu bez potreby zdvojiť substitúciu. Niektoré systémy používajú DAG s agresívnym zdieľaním, tiež zvaným perfektné zdieľanie, kde zaistíme, že existuje len jediný výskyt termu bez ohľadu na počet súvislostí v kt. sa nachádza. Agresívne zdieľanie sa používa v niektorých dokazovačoch viet. Agresívne zdieľanie taktiež zjednodušuje nelineárne spájanie, pokiaľ úloha preverovania súdržnosti v hromadných substitúciách, je lepšia, ako zisťovanie, či substitúcie sú štruktúrovo identické. Čiastočne účinné reprezentácie boli vyvinuté pre takéto agresívne zdieľanie v súvislosti s problémom kongruenčného uzáveru (congruence closure problem). Agresívne zdieľanie je typické v aplikáciách ako LP a FP aj keď v niektorých prípadoch sa javí ako užitočnejšia iná optimalizovaná reprezentácia známa ako hashed cons.

Priame termy (flat terms) sú lineárna štruktúra ktorá zodpovedá reprezentácii dynamického zoznamu (linked-list) vrcholov navštívených pri priamom (preorder) prechádzaní termom. Pod priamym prechádzaním budeme rozumieť prehľadávanie do hĺbky z ľava do prava. Na uľahčenie prechádzania podtermammi má vrchol n šípky do vrcholov, ktoré nasledujú bezprostredne za všetkými deťmi vrchola n.

napr. reprezentácia flattermu f(g(a,h(h(y)),h(y),x)

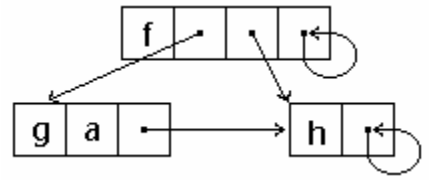


- symbol je funkčný symbol v danom vrchole
- ďalší je ďalší vrchol pri priamom prechádzaní termom
- predch je predchádzajúci prchol pri priamom prechádzaní termom
- koniec je posledný vrchol (pri priamom prechádzaní) v podstrome, začínajúci v danom vrchole

Väčšina operácií na termoch vyžaduje nejakú formu prechádzania termom. Ak sa obmedzíme na priame prechádzanie tak flatterm ponúka oveľa efektívnejšiu cestu na prechádzanie ako cez bežné štruktúry. Naviac každý vrchol má pevnú štruktúru s rovnakým počtom políčok. Z toho vyplýva, že veľkosti vrcholov sú rovnaké čo uľahčuje správu pamäte narozdiel od štruktúr v ktorých veľkosť vrchola závisí od počtu jeho detí. Ak sa umiestnenie urobí do poľa je to naozaj malá štruktúra z ktorej vypadnú políčka predch a ďalší. Prechádzanie je potom rýchlejšie. Flattermy sú obzvlášť užitočné ak sú použité spolu so stromami diskriminácie z ľava do prava. Ale nie sú výhodné v prípadoch kedy priame prechádzanie je rôzne od stromov diskriminácie z ľava do prava. Ďalším problémom je, že táto štruktúra nepodporuje zdieľanie. Flattermy sa používajú v niektorých dokazovačoch viet na reprezentáciu dopytovacích termov v kt. nezáleží na štruktúre.

Termy v Prologu Prolog používa optimalizovanú verziu obvyklej reprezentácie termov, kde každý vrchol je označený ako konštanta, funkcia alebo referencia s árnosťou 0, >0, resp. so smerníkmi na iné termy. Premenné sú reprezentované ako odkazy na ten istý vrchol, čo umožňuje obzvlášť efektná implementácia spájania premenných cez nastavenie smerníka na term ktorý sa má pripojiť. Treba poznamenať, že ani premenné ani konštanty nemajú deti. Preto sa prologovské implementácie uchovávajú ako hodnoty ktorým zodpovedajú rodičovské vrcholy. Funkčné symboly sú reprezentované použitím vrcholov so všetkými funkčnými symbolmi a zodpovedajúcimi deťmi ktoré sú reprezentované použitím iných polí.

napr. Prologovská reprezentácia termu f(g(a,h(y)),h(y),x)



2.4.2. Uchovávanie premenných

Sú situácie keď rovnaký term môže byť použitý s rôznymi množinami premenných počas tej istej návratovej operácie. Napr term f(y,g(y)) sa môže použiť n krát v klauzule $Øf(x_0,x_1)$ *v* $Øf(x_1,x_2)$ *v* $Øf(x_{n-1},x_n)$ *v* $h(x_0,x_n)$. Na túto klauzulu potrebujeme n kópií f(y,g(y)) s y inicializovanými termami x, $g(x_0)$,...,$g^{n-1}(x_0)$. Typická realizácia pre takéto situácia je použitie **bánk premenných**. Predpokladajme, že máme indexovaný term l s premennými y_1,...,y_n. Vytvoríme niekoľko kópií y_1,...,y_n:

$$y_{11} \ldots y_{1n}$$

$$y_{m1} \ldots y_{mn}$$

Každá kópia sa nazýva banka premenných. Pri k-tej operácii sa použije y_{k1},...,y_{kn} namiesto y_1,...,y_n. Na zvýšenie efektivity sa banky premenných uchovávajú len raz ako pole.

2.4.3. Reprezentácia indexov

Reprezentácia grafmi

Reprezentácia grafmi sa zaoberá otázkami podobnými tým, čo nastávajú v automatoch spájania reťazcov. Asi najdôležitejšia stránka je reprezentácia vonkajších prechodov zo stavu. Rôzne dátové štruktúry sú

- *pole* - rýchle ale s veľkou pamäťovou náročnosťou ak počet vonkajších prechodov je oveľa menší ako veľkosť abecedy.
- *dynamický zoznam* - zaberá menej pamäte ale je pomalý pri vykonávaní prechodov
- *hashovacia tabuľka* - požiadavky na pamäť sú trochu vyššie ako dynamický zoznam ale je významne rýchlejšia, aj keď kolízie môžu spôsobiť problémy
- *jump table* - kombinácia reprezentácie poľa s malými pamäťovými požiadavkami. V jump tabuľkách sa hodnota symbolu používa priamo ako index v prípade reprezentácie v poli. Na zmenšenie pamäťových požiadaviek sú tabluľky pre rôzne automaty "prekrývané".

Problém oprimalizácie pamäťových požiadaviek je NP-úplný ale existuje efektívna heuristika ktorá v praxi celkom dobre funguje.

Kódové stromy

Je obvyklé v automatizovanej dedukcii kompilovať dopytovací term do kódu ktorý je vykonávaný na indexe. Ak je čas kompilácie porovnateľný s návratovým časom tak sa kompilácia vypláca. Je menej časté kompilovať index samotný do kódu ktorý sa vykoná na dopytovacom terme.

2.5. Bežná prácia pri indexovaní

Indexovanie podľa symbolov môže byť opísané nasledovne: Vybrané kandidátske termy sú tie ktoré majú identické funkčné symboly na príslušných miestach dopytovacieho a indexovaných termov. Počas vybratia kandidátskych termov potrebujeme preskúmať podmnožinu pozícií v dopytovacom terme v nejakom poradí. V podstate, celý proces sa dá vidieť ako konštrukcia reťazca symbolov z dopytovacieho termu a odhalenie, či sa tento reťazec dá spojiť s reťazcami z množiny indexovaných termov. Väčšina techník indexovania podľa symbolov spadá pod širokú tému indexovania založeného na zhode reťazcov (string matching).

2.5.1. Reťazce pozícií

Prvé, čo treba urobiť, je vytvoriť textovú reprezentáciu termu. Jedou cestou je napísať symboly vyskytujúce sa v terme v istej sekvencii a tak dôjsť k reťazcu. Tým sa môžu stratiť niektoré informácie zo štrukúry termu. Na zachovanie tejto informácie môžeme spojiť každý symbol termu s jeho pozíciou.

napr. term f(a,g(b,c)) môžeme reprezentovať ako reťazec:

$<\Lambda;f><1;a><2;g><2.1;b><2.2;c>$ (2.1)

Alebo ak nechceme generovať jeden reťazec môžeme zvoliť generovanie viacerých reťazcov:

$<\Lambda;f><1;a>$, $<\Lambda;f><2;g><2.1;b>$, $<\Lambda;f><2;g><2.2;c>$ (2.2)

Reťazce (2.1), (2.2) nazývame reťazce pozícií alebo skrátene p-stringy.

Definícia 2.6. (p-sting) Reťazec pozícií S nad abecedou je neprázdny reťazec tvaru $<p_1,s_1><p_2,s_2>...<p_n,s_n>$ kde p_i P, s_i V taký že:

1. $\forall$ $1 \leq i,j \leq n$ ak p_i je prefix p_j taj $i<j$
2. $\exists$ term t, zvaný charakteristický term pre S taký že
   a. $\forall$ i [1..n], $root(t/p_i)=s_i$
   b. $p_1,...,p_n$ je presne množina pozícií v t.

Intuitívne môžme vidieť pozície $p_1,...,p_n$ v p-stringu ako vyjadrenie cesty na prechod termom so symbolmi $s_1,..,s_n$ po ktorých na tejto ceste prechádzame. Ak je poradie prechádzania dané pevne (prehľadávať môžme do hĺbky alebo do šírky), tak informácia o pozícii je nepotrebná. Ak sa nechceme obmedziť žiadnou pevnou cestou prehľadávania tak informácia o pozícii je dôležitá. V štruktúre termu je možné a-alebo nezmyselné navštíviť vrchol pred navštívením všetkých jeho rodičov. Naviac ak chceme mať p-string na reprezentáciu charakteristického termu, ktorý je jedinečný až na 1 alebo viac premenných. Napr. p-string $<\Lambda;f><1;*><2;g><2.1;*><2.2;c>$ reťazcom $<\Lambda;f><2;g><2.2;c>$.

Na označenie charakteristického termu použijeme ct(S).

napr. $ct(<\Lambda;f><2;g><2.2;c>)=f(*,g(*,c))$.

2.5.2. Zlúčiteľnosť p-stringov a indexovanie

Pristupujeme k popísaniu toho, ako môžu byť p-stringy, vytvorené z indexovaných termov, použité ako základ identifikácie tých termov, ktoré sú zlúčiteľné s daným dopytovacím termom. Na tento účel potrebujeme rozšíriť predstavu R-zlúčiteľnosti na posobenie medzi p-stringami a termami.

Definícia 2.7. *(zlúčiteľnosť p-stringov)*
Majme term t a p-string S, definujme S-prefix termu t (ozn. t\S) ako term t' získaný nahradením každej pozície v t neobsiahnutej v S nejakou novou premennou. S nazveme R-zlúčiteľná (R-compatible) s termom t ak R(ct(S),t').

Rozšírime označenie R(S,t) na označenie R-zlúčiteľnosti medzi reťazcom S a t. Napr. p-string <Λ;f><2;g><2.2;c> ktorého charakteristický term je f(*,g(*,c)) je zlúčiteľný s dopytovacím termom f(a,g(b,x)) s rešpektovaním návrarovej podmienky "unif". Na druhej strane p-string <Λ;f><1;c><2;g><2.2;c> nie je unif-zlúčiteľný s týmto termom. Naviac <Λ;f><2;g><2.2;c> nie je zlúčiteľná s týmto termom s rešpektovaním relácie "gen". Jednoduchý spôsob na určenie či je R(I,t) splnená je vygenerovanie 1 alebo viac p-stringov z I a potom zistenie či je každý z týchto p-stringov R-zlúčiteľný s t. Na zaistenie toho, aby indexovanie založené na tomto prístupe bolo korektné (nájde všetky termy, ktoré môžu byť R-kompatibilné s t) potrebujeme aby tieto p-stringy boli charakteristické reťazce I:

Definícia 2.8. *(charakteristická množina termov)*
Množina $\{S_1,...,S_k\}$ p-stringov sa nazýva charakteristická množina termu I ak $\forall$ term t : $R(I,t) \rightarrow \Lambda_{1\le i\le k} R(S_i,t)$

Táto podmienka zaručuje, že každý potenciálny term je zechytený filtrom. Pre daný term t označíme túto množinu S_t. Ak je táto množina jednoprvková tak S_t je označenie charakteristického reťazca. Označíme S_L zjednotenie charakteristických množín všetkých termov t množiny L : $\bigcup_{I\,L} S_I$.

Indexovania založené na symboloch sú založené na vytváraní charakteristickej množiny S_L reťazcov z indexovanej množiny termov L a na vytváraní automatu (alebo stromu) všetkých týchto reťazcov. Automat môže byť použitý na rýchle nájdenie zlúčiteľných p-stringov s dopytovacím termom. Táto informácia musí byť rozšírená o nájdenie potenciálnych termov. Uvažujme každý indexovaný term I a zistíme, či je dopytovací term zlúčiteľný s jeho charakteristickou množinou, V najhoršom prípade prezretie všetkých kombinácií zaberie čas $O(\bigcup_{I\,L} |S_I|)$. V praxi sa za prechod kôli vytvoreniu charakteristických reťazcov pridajú nejaké ďalšie obmedzenia, čo zrýchli indexovanie.

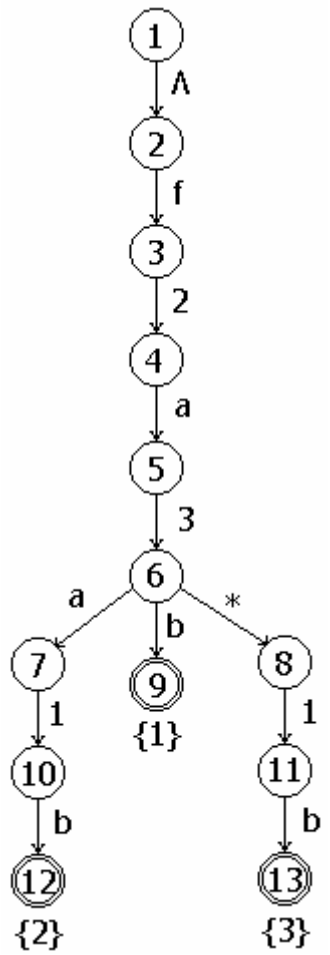
Na ilustráciu uvažujem indexovanú množinu

L={f(*,a,b),f(b,a,a),f(b,a,*)}

s návratovou podmienkou "gen". Dajme tomu, že vytvárame 1 charakteristický reťazec z každého termu, potom

S_L={ <Λ;f><2,a><3;b>,
<Λ;f><2;a><3;a><1;b>,
<Λ;f><2;a><3;*><1;b>}

Automat pre tieto 3 reťazce vyzerá takto:



Na koncových vrcholoch sú čísla termov, ktorým p-stringy zodpovedajú. Majme dopytovací term t=f(b,a,c) a návratovú podmienku "gen", potom môžeme použiť tento automat na indexovanie nasledovne: Najprv nájdeme Λ pozíciu v t. Keďže symbol pri prechode 2→3 je identický prejdeme do 3. Potom nájdeme pozíciu 2 v terme t. Zhodou so symbolom a sa dostaneme do stavu 5. Potom preveríme 3. pozíciu v terme a so zistením, že je zlúčiteľná s * sa dostaneme do 8. Nakoniec preverením 1. pozície v koncovom vrchole zistíme {3}. Pokiaľ je dopytovací term zlúčiteľný len s indexovaným termom f(b,a,*), môžeme priamo prehlásiť že dopytovací term je asi zlúčiteľný s týmto jediným termom. Počas vykonávania návratovej operácie môže byť potrebný backtrack ak je možné prejsť viacerými vetvami.

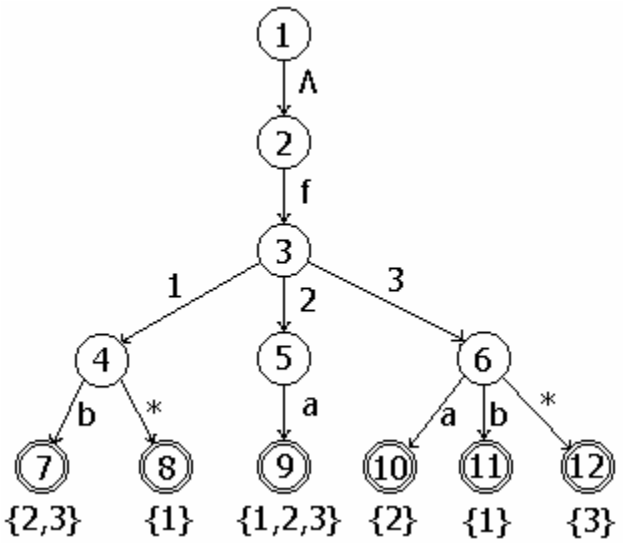
Teraz uvažujme tú istú množinu L ale s rôznymi množinami p-stringov a návratovou podmienkou "unif". Nech p-stringy sú:

$S_{f(*,a,b)}$={<Λ;f><1;*> , <Λ;f><2;a> , <Λ;f><3;b>}

$S_{f(b,a,a)}$={<Λ;f><1;b> , <Λ;f><2;a> , <Λ;f><3;a>}

$S_{f(b,a,*)}$={<Λ;f><1;b> , <Λ;f><2;a> , <Λ;f><3;*>}

V tejto štruktúre je vždy 1 p-string pre každú cestu z koreňa do listu. Automat vyzerá takto:



Teraz vyskúšame vrátenie termov pre dopytovací term t=f(b,*,b) s návratovou podmienkou "unif". Potrebujeme vrátiť tie indexované termy, z ktorých všetky p-stringy sú zlúčiteľné s t. Keďže úspešné vetvy končia vo vrcholoch 7,8,9,11,12 tak prehlásime že potenciálne zlúčiteľné termy sú 1 a 3. Tento príklad zodpovedá technike ktorú využíva indexovanie ciest (path indexing).

2.6. Indexovanie ciest (path indexing)

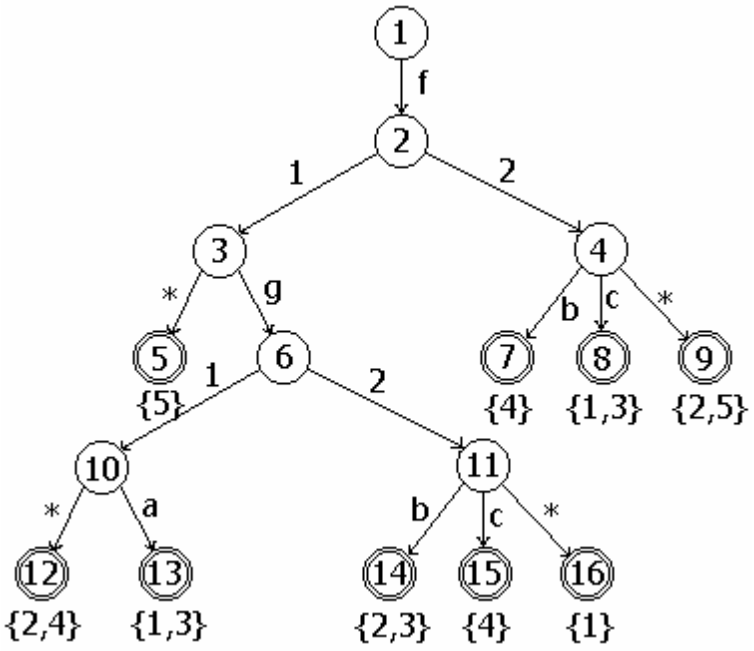
Pri indexovaní ciest vytvárame niekoľko p-stringov z každého indexovaného termu, z ktorých každý zodpovedá prechodu z koreňa do listu termu. Potom vytvoríme strom týchto p-stringov ktorý použijeme na vykonanie návratovej operácie.Keďže sa uchováva stále iba 1 cesta nie je potrebné uchovávať v indexe pre každý term celú cestu. Napr. pri terme f(b,g(f(x),a)) použijeme miesto p-stringu <Λ;f><2;g><2.1;f><2.1.1,*> jednoduchšiu reprezentáciu f.2.g.1.f.1.*.

Druhou optimalizáciou je kombinovanie výsledkov spájaním p-stringov. Napríklad pre množinu indexovaných termov:
1, f(g(a,*),c) 2, f(g(*,b),*) 3, f(g(a,b),c) 4, f(g(*,c),b) 5, f(*,*)
získame tieto p-stringy:

f.1.*	{5}
f.1.g.1.*	{2,4}
f.1.g.1.a	{1,3}
f.1.g.2.*	{1}
f.1.g.2.b	{2,3}
f.1.g.2.c	{4}

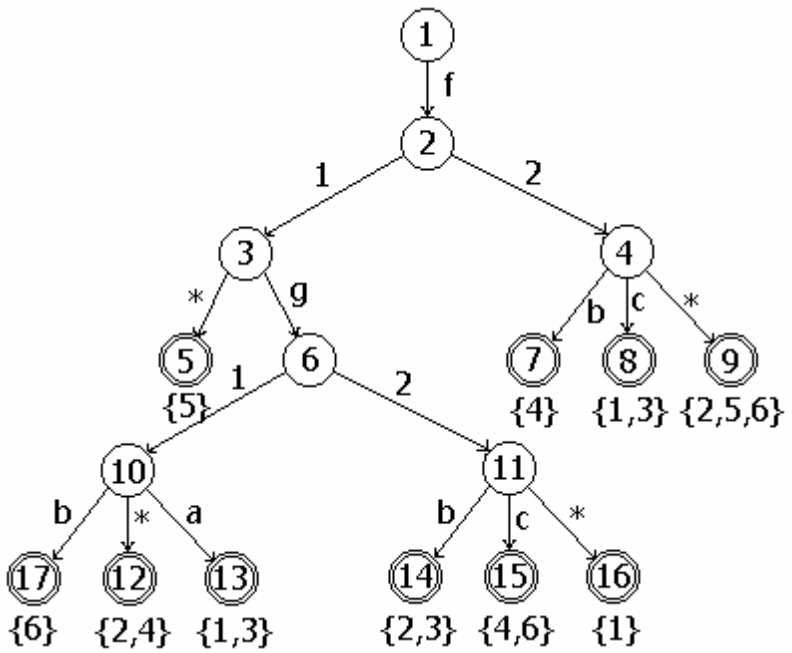
f.2.*	{2,5}
f.2.c	{1,3}
f.2.b	{4}

strom pre tieto p-stringy vyzerá takto:



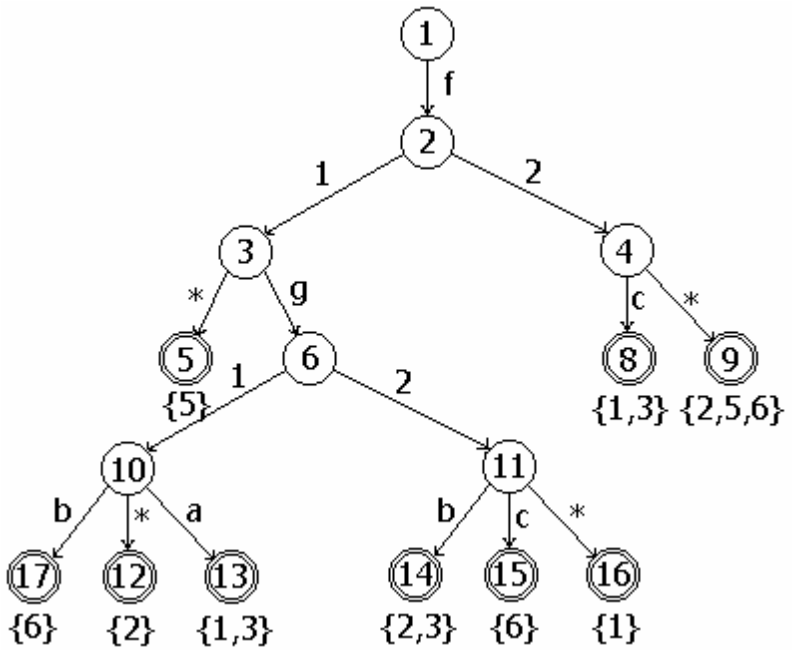
Konštrukcia indexu

Konštrukcia indexu sa deje postupným vkladaním p-stringov z každého indexovaného termu do stromu. Ak už daný p-string bol vložený, jednoducho sa zväčší množina asociovaná s jeho koncovým stavom o tento term. Ak je to nový p-string, tak sa napojí nová vetva s jednoprvkovou asociovanou množinou s jej koncovým stavom. Obrázok ukazuje strom po vložení f(g(b,c),*) (ozn. 6).



Mazanie termov

Najprv vytvoríme p-stringy z mazaného termu. Každý p-string nájdeme v strome a odstránime term z asociovanej množiny s jeho koncovým stavom. Ak po vymazaní termu je množine prázdna zmažeme aj koncový stav. Zmažeme aj všetkých predchodcov tohoto stavu pokiaľ nemajú deti. Obrázok ukazuje vymazanie f(g(*,c),b) (ozn. 4)z predchádzajúceho obrázka.



Návrat generalizácií

```
function retrievegen(stav_v_strome s, term u) //vracia množinu termov M
 begin
  M:=∅;
  if u∉Var & ∃ hrana zo stavu s do stavu s' označená root(u) then
    if s' je koncový stav then
      M:=Ms';
    else
      nech I je množina čísiel i takých že ∃ hrana z s' do si označená i;
      M:= ∪i∈I retrievegen(si,u/i)
  if ∃ hrana zo stavu s do stavu s* označená * then
   M:=M Ms*;
  return M;
 end;
```

V tomto algoritme je s stav v strome p-stringov a u je podterm dopytovaného termu t. Na začiatku sa spýtame retrievegen(s_0,t), kde s_0 je koreň stromu. Pužívame označenie M_s pre množinu kandidátskych termov asociovaných s koncovým stavom s. Môžeme spraviť jedno zlepšenie na vyhnutie sa zjednocovaniu množín nasledovne: Nech s je stav v strome ktorý má hranu označenú * do stavu s'. Vytvoríme kópiu indexovaných termov v Ms_* do kandidátskych množín asociovaných so všetkými potomkami stavu s. To spôsobí menšiu cenu návratov ale môže zvýšiť cenu vkladania a mazania. Keď sa vkladá (maže) nový p-string končiaci , nemôžeme iba obnoviť množinu kandidátskych termov asociovaných so

stavom s_*, ale aj kandidátske množiny asociované so ∀ potomkami s. Ešte horšie je mazanie t zo stromu. Musíme zisťovať, či iné termy, ktoré majú * na tých pozíciách, kde term t nemá premenné, môžu byť vymazané z vetiev ktoré patrili termu t.

Návrat inštancií

```
function retrieveinst (stav_v_strome s, term u) //vracia množinu termov M
 begin
  if u=* then
    nech F je množina ∀ koncových stavov potomkov s;
    M:= s' F Ms';
  else
   if ∃ hrana z s do s' označená root(u) then
     if s' je koncový stav then
       M:=Ms';
     else
       nech I je množina čísiel i takých že ∃ hrana z s' do si ozn. i;
       M:= i I retrieveinst(si,u/i)
  return M;
 end;
```

Všimnime si, že premenné v indexovanom terme nehrajú rolu, lebo ignorujeme nelineárnosť počas indexovania. Napr.ak v dopytovacom terme máme p-string $s_1.p_2.s_2.p_3...p_{k-1}.s_{k-1}.p_k.*$ tak tento p-string je zlúčiteľný so V p-stringami indexu tvaru $s_1.p_2.s_2.p_3...p_{k-1}.s_{k-1}...$. Takže vezmeme zjednotenie všetkých kandidátskych množín zodpovedajúcim všetkým takým p-stringom. Treba pripomenúť, že množina F môže byť dosť veľká a zjednotenie množín časovo náročné. Túto cenu môžeme znížiť predpočítaním zjednotení a ich uchovávaním v každom stave. Narozdiel od generalizácie, tento proces nezvýši cenu vkladania alebo mazania, môže zvýšiť akurát pamäťovú náročnosť.

Návrat unifikovateľných termov

Návratový algoritmus je v tomto prípade získaný kombináciou generalizácie a inštancie. Kombinácia je v istom zmysle ako zjednotenie, kde získané kandidátske množiny sú väčšie.

```
function retrieveunif(stav_v_strome s, term t) //vráti množinu termov M
 begin
  if u=* then
    nech F je množina ∀ koncových stavov potomkov s;
    M:= s' F Ms';
  else
    M:=∅;
    if ∃ hrana z s do s' označená root(u) then
     if s' je koncový stav then
       M:=Ms';
     else
       nech I je množina čísiel i takých že ∃ hrana z s' do si ozn. i;
       M:= i I retrieveunif(si,u/i);
  if ∃ hrana zo stavu s do stavu s* označená * then
   M:=M Ms*;
  return M;
 end;
```

Aj tu môžeme urobiť rovnakú optimalizáciu s rovnakým účinkom ako pri gen a inst.

Návrat variantov(premenovanie premenných)

Aj táto operácia je symetrická vzhľadom na dopytovací a indexované termy. Na rozdiel od unifikácie sa zaoberáme s premennými tak ako s funkčnými symbolmi.

```
function retrievevar (stav_v_strome s, term u) //vráti množinu termov M
 begin
  if ∃ hrana z s do s' označená root(u) then
     if s' je koncový stav then
       M:=Ms';
     else
       nech I je množina čísiel i takých že ∃ hrana z s' do si ozn. i;
       M:= ⋃i∈I retrievevar(si,u/i);
  return M;
 end;
```

Variácie indexovania ciest

Operácie zjednotenia a prieniku množín môžu byť efektívnejšie pri použití vektorov bitov (bitvector reprezentation) na množiny. Môžu vzniknúť problému ak množiny obsahujú veľa termov (>20 000). Použitie vektorov ciest je účinné pri malom počte termov (niekoľko 100). Tak isto sa dá testovať iba prvých k p-stringov v dopytovacom terme a pri k+1 p-stringubrať ohľad aj na už nájdené kandidátske termy v množine D a testovať či sa už hľadaná vetva v nej nenachádza a tak neprechádzať nepotrebné vetvy. Variáciou indexovania ciest môže byť aj obmedzenie dĺžky p-stringov, dlhšie sa odseknú. Takto môžeme kontrolovať maximálnu veľkosť stromu. Tieto obmedzenia nespôsobia stratu možných kandidátskych termov. Ďalšie zlepšenie môže vracať nájdené termy po jednom. Na to treba vytvoriť tzv. query-tree (strom dopytu) ktorý uchováva zjednotenia a prieniky čiže je to strom operácií ktoré sa majú vykonať.

Výhody a nevýhody indexovania ciest

Výhodou je malý nárok na pamäť ak sa kandidátske termy uchovávajú len pri koncových stavoch, takisto sa pamäť dá zmenšiť obmedzeniami na veľkosti p-stringov. Druhou výhodou je, že sa tu nevyskytuje žiaden backtrack. Vkladanie a mazanie indexov je taktiež veľmi účinné.

Nevýhodou je cena kombinácie medzivýsledkov, čo spôsobuje zníženie výkovu. Oproti ostatným technikám môže byť indexovanie ciest účinné pre návrat inštancií, nevýhodná je pri unifikácii a generalizácii.

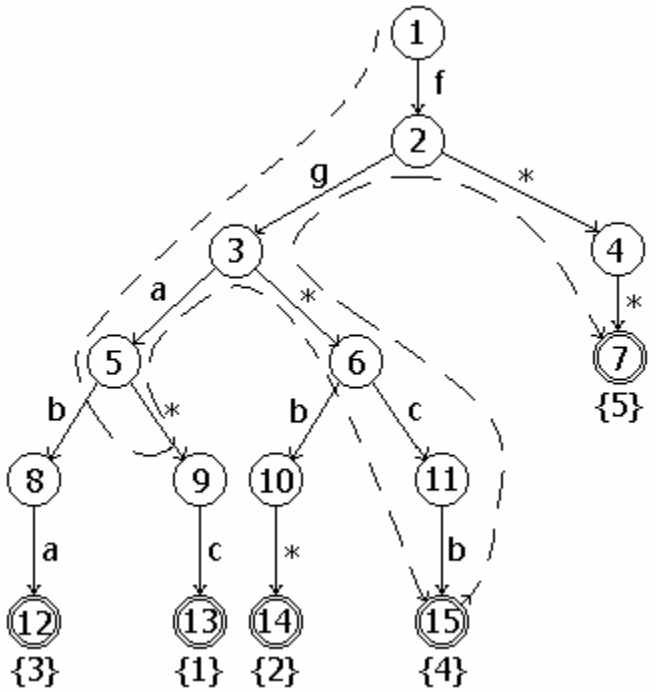
2.7. Rozlišovacie stromy (discrimination trees)

Pri indexovaní rozlišovacími stromami vytvárame z každého indexovaného termu jediný p-string. P-string sa získava prechodom v terme do hĺbky (preorder traversal). Vytvárame strom práve z týchto p-stringov. Lepšiu reprezentáciu p-stringov môžeme získať využitím vlastností prehľadávania do hĺbky, keď si uvedomíme, že je jednoznačná

korešpondencia medzi reťazcom vzniknutým prehľadávaním do hĺbky a zápisom termu, teda informácia o pozícii nie je dôležitá. Majme takúto množinu termov a k nim prislúchajúcich p-stringov:

(1)	f(g(a,*),c)	f.g.a.*.c	{1}
(2)	f(g(*,b),*)	f.g.*.b.*	{2}
(3)	f(g(a,b),a)	f.g.a.b.a	{3}
(4)	f(g(*,c),b)	f.g.*.c.b	{4}
(5)	f(*,*)	f.*.*	{5}

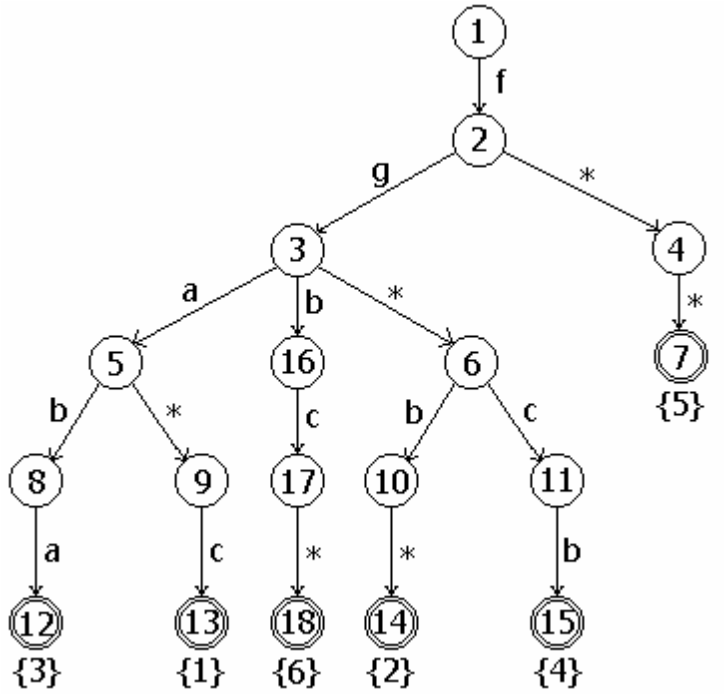
Index a prechod ním pre návratovú podmienku gen a dopytovací term f(g(a,c)b) vyzerá takto:



P-string pre dopytovací term je f.g.a.c.b. Nám treba porovnávať symboly tohto p-stringu s vrcholmi na ceste z koreňa po vrchol 5. Na tomto mieste nemôžeme pokračovať ľavou vetvou lebo b≠c. Aj keď je * prijateľný symbol, zo stavu 9 sa pokračovať nedá a musíme sa rekurzívne vrátiť do stavu 3. Odtiaľ môžeme postupovať až do stavu 15 a označiť 4. kandidátsky term. Ak chceme nájsť všetky generalizácie pokračujeme aj cez vrcholy 2,4,7. Ak by sme chceli za návratovú podmienku inštanciu alebo unifikáciu, musíme sa zaoberať prípadmi, keď dopytovací term má premenné na mieste, kde má indexovaný term funkčný symbol. V tomto prípade potrebujeme mechanizmus na efektívne skákanie na príslušné podtermy v indexovaných termoch. Na to budeme používať "jump listy".

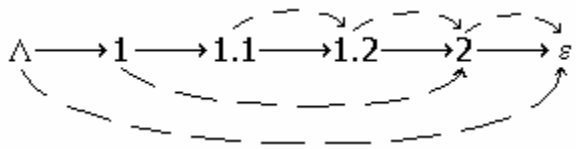
Konštrukcia indexu

Konštrukcia rozlišovacieho stromu je priama. Začíname s prázdnym stromom a postupne vkladáme jednotlivé indexované termy. Teda vytvoríme p-string a ten vložíme podobne ako pri indexovaní ciest. Na obrázku je vloženie termu *f(g(b,c),*)* do predchádzajúceho príkladu:



Operácie prechodu termom

Na technické účely rozšírime P(t) (všetky pozície v terme t) o 1 špeciálny znak ε, ktorý nazveme posledná pozícia v t. Množinu P(t) ∪ {ε} označíme $P^+(t)$. Budeme používať aj lexikografické zoradenie pozícií. $\forall u\ P(t): u<\varepsilon$. Na prechod termom použijeme 2 operácie $next_t$ a $after_t$ definované nasledovne: Nech $\Lambda=p_1<p_2<...<p_n<p_{n+1}=\varepsilon$ sú všetky pozície v t. Potom $next_t(p_i)=p_{i+1}$ pre $\forall i \le n$, $after_t(p_i)=p_j$ kde j je najmenšie také číslo, že j>i a $\forall k$: i<k<j je pozícia p_i prefixom p_k. Na ďalšom obrázku sú $next_t$ znázornené plnou a $after_t$ perušovanou čiarou na pozíciách v t=f(f(a,a),a).



Návrat generalizácií

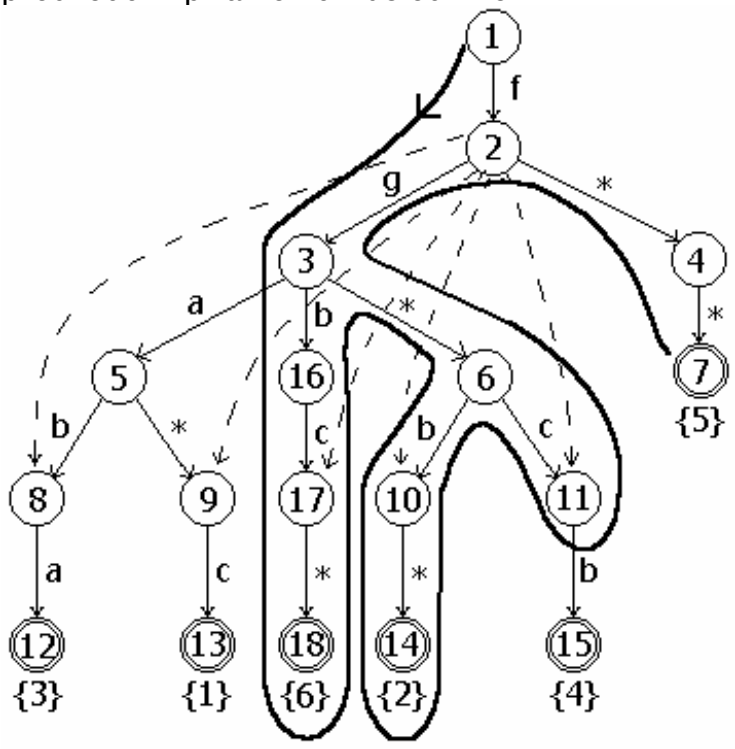
```
function retrievegen(stav indexu V, term t, pozícia p)
                                                    //vracia množinu termov C
 begin
  C:=∅;
  if ∃ hrana z V do V' označená znakom root(t/p) then
    if V' je koncový stav then
      C:=C_V'
    else
      C:=retrievegen(V',t,next_t(p));
  if ∃ hrana z V do V* označená * then
    C:=C   retrienegen(V*,t,after_t(p))
 end;
```

Môžeme si všimnúť, že algoritmus sa dosť podobá aglgorimu používaného pri indexovaní ciest. Iné tu je to, že nie je potrebné počítať prienik a nasledujúca pozícia je implicitne daná súčasnou pozíciou. V* bol pri indexovaní ciest konečný stav, zatiaľ čo v rozlišovacom strome treba vo všeobecnosti ďalšie vnáranie. Tento algoritmus umožňuje nevracať naraz celú množinu výsledkov, ale jeden po druhom. Teda miesto zjednotenia na konci funkcie sa môže generovať výsledok a prípadne vložiť otázku o ďalšom pokračovaní.

Návrat unifikovateľných termov

```
function retrieveunif(stav indexu V, term t, pozícia p)
begin
  if t/p='*' then
    C:= V' JumpList(V)retrieveunif(V',t,next_t(p))
  else
   begin
    C:=∅;
    if ∃ hrana z V do V' označená znakom root(t/p) then
      if V' je koncový stav then
        C:=C_V'
      else
        C:=retrieveunif(V',t,next_t(p));
    if ∃ hrana z V do V* označená * then
      C:=C   retrieneunif(V*,t,after_t(p))
   end;
 end;
```

Aj tento algoritmus je podobný tomu z indexovania ciest. Funkcia JumpList(V) v podstate zodpovedá operácii $after_t$, ktorá zabezpečí pri nájdení premennej v dopytovacom terme preskočenie zodpovedajúcej časti v indexovanom terme. V nasledujúcom obrázku je znázornený výpočet pre dopytovací term f(g(b,*),a). Čiarkovane je naznačený JumpList rôzny od prechodu k priamemu následníkovi.



Výhody a nevýhody rozlišovacích stromov

Rozlišovacie stromy vylepšujú indexovanie ciest tak, že odstraňujú drahé množinové operácie. Jednou nevýhodou je, že vyžadujú viac miesta, pretože nepodporujú zdieľanie stavov pre jednotlivé pozície pokiaľ nemali na všetkých predchádzajúcich pozíciách zhodné symboly. Výskyt jump listov spôsobuje drahšie vkladanie a mazanie. Pre návratové operácie je potrebný backtrack vyžadujúci preskúšavanie symbolov. Tieto spomalenia sú ale obyčajne menšie ako množinové operácie v indexovaní ciest. Rozlišovacie stromy sú vhodné pre aplikácie pri ktorých je dôležitá rýchlosť návratovej operácie a nie operácie správy indexu.

Okrem posledných dvoch indexovacích techník je vymyslených ešte veľa ďalších , využiteľných pri rôznych iných programoch. V praktickej časti tejto práce je využitá práve technika rozlišovacích stromov. Po preštudovaní aj rôznych iných indexovacích techník ktoré sa nachádzajú v použitej literatúre, sa mi zdala najvhodnejšia pre vybranú štruktúru uloženia termov a pre požadované návratové podmienky.

3. Praktická časť

3.1. Vlastnosti programu

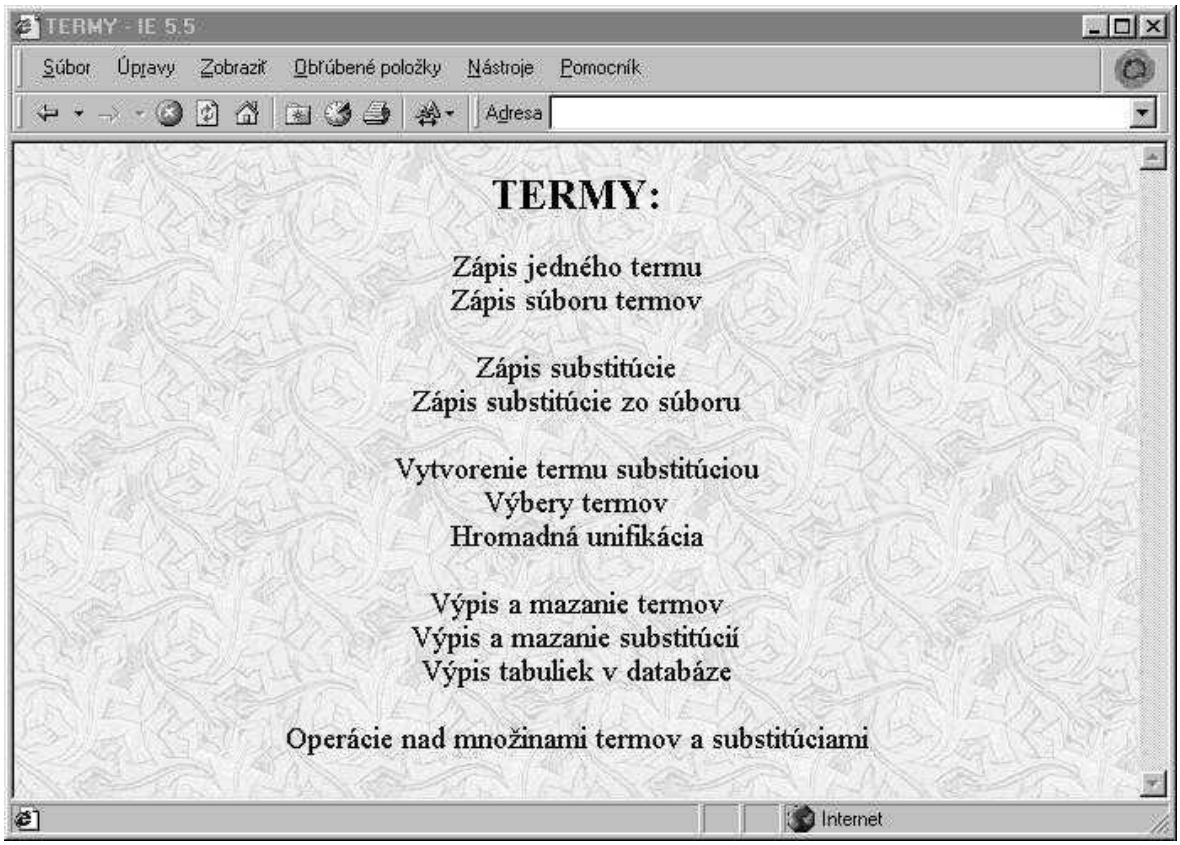
Databázový systém, ktorý som vytvoril v softwarovej časti tejto diplomovej práce, umožnuje bežnú prácu s množinami termov. Niektoré algoritmy, ako napríklad unifikácia dvoch termov alebo indexovanie množiny termov, som prebral z literatúry. Mojou prínosom je hlavne uloženie množín termov v databáze a následná práca s nimi. Program som písal v jazyku PHP. Za databázový server som zvolil MySQL. Tento databázový systém umožnuje nasledovné operácie:

1. Úschova termov a množín termov. Sem patria takéto operácie:
   - vkladanie termu alebo množiny termov do databázy priamym zadaním textovej reprezentácie termov alebo zo zdojového súboru
   - vymazanie termov alebo celých množín z databázy
   - operácie zjednotenia, prieniku a rozrielu množín termov
2. Substitúcie.
   - vkladanie substitúcií do databázy priamym zadaním textovej reprezentácie dvojíc v substitúcii alebo zo zdojového súboru
   - pri vkladaní je možné zadať požiadavku zachovania idempotentnosti v subtitúcii
   - vytvorenie nových termov aplikovaním substitúcie na iné termy
3. Výbery termov.
   - nájdenie identických termov s dopytovacím termom v databáze
   - nájdenie k dopytovaciemu termu termy ktoré sú s ním unifikovateľné, alebo sú jeho generalizáciou, inštanciou resp. totožné až na premenovanie premenných
   - uloženie takejto skupiny termov ako ďalšej množiny termov v databáze
4. Hromadná unifikácia.
   - Nájdenie unifikátora pre zadanú množinu termov

Všetky tieto vyhľadávacie operácie využívajú indexovaciu techniku rozlišovacích stromov (discrimanation trees), ktorej štrukúra je taktiež uložená v databáze.

3.2. Opis programu z hľadiska užívateľa.

Keďže program je robený v jazyku PHP, je prirodzené, že sa spúšťa pomocou internetového prehliadača zadaním URL adresy. Keďže neviem kde bude tento program zavesený, neuvádzam tu žiadnu určitú adresu. Po napojení sa nám zobrazí hlavná stránka:



Po zobrazení stránky môžeme vidieť hlavné menu. Po kliknutí na "Zápis jedného termu" sa nám zobrazí formulár do ktorého treba vpísať vkladaný term v štandartnom prefixovom tvare (napr. f(g(c,"b",1),ke), kde **f** je binárny term, **g** je 3-árny term, **c** a **ke** sú premenné a **"b"** a **1** sú konštanty. Vo všeobecnosti sú čísla a ľubuvoľné znaky v úvodzovkách považované za konštanty, postupnosť znakov je považovaná za premennú alebo za názov n-árneho termu ak za ním nasleduje otvárajúca zátvorka) a vybrať množinu, do ktorej má byť tento term vložený. Máme aj možnosť zadať názov novej množiny. Po odoslaní formulára sa nám zobrazí stránka potvrdzujúca zapísanie termu do databázy.

Ak nechceme vkladať iba jeden ale veľa termov, je jednoduchšie tieto termy vkladať cez kliknutie na "Zápis súboru termov". Zobrazí sa nám podobná obrazovka ako pri vkladaní jedného termu akurát že miesto zadávania termu sa zadá názov súboru, v ktorom sú termy napísané pod sebou. Po odoslaní formulára sa znovu zobrazí stránka ktorá potvrdzuje vloženie každého termu.

Program pravdaže v obidvoch prípadoch pred vložením termu do databázy overí jeho syntaktickú správnosť. Ak zistí nejakú chybu (napr. zlé uzátvorkovanie) tak vypíše že tento term sa uložiť nepodarilo.

Prehľad termov vložených do množín je možný po kliknutí na "Výpis a mazanie termov". Ako už názov napovedá, v tejto sekcii je možné mazať termy z množín. Stačí len označiť termy, ktoré chceme vymazať a stlačiť tlačidlo "označené vymaž".

Po kliknutí na "Zápis substitúcie" sa nám zobrazí formulár do ktorého môžeme vpisovať dvojice "premenná -> term" ktoré budú zaradené do substitúcie. Samozrejem určíme aj to, do ktorej substitúcie sa bude

vkladať. Okrem toho máme možnosť zvoliť, aby sa zachovala idempotentnosť, pri ktorej sú premenné na ľavej strane jedinečné a nevyskytujú sa na pravej strane. Pokiaľ to vkladané dvojice nespĺňajú a nedajú sa ani modifikovať tak aby to spĺňali tak sa nevloží tá dvojica v poradí, ktorá to nespĺňa. Ak nemáme odškrtnuté zachovávanie idempotentnosti tak sa nevložia len tie dvojice ktoré zopakovali premennú na ľavej strane.

Ako pri vkladaní termov aj substitúcia sa dá vložiť zo súboru. V súbore sú dvojice písané pod sebou a dvojica má premennú a term na ľavej strane oddelené medzerou.

Prehľad substitúcií a mazanie dvojíc z nich je možné po kliknutí na "výpis a mazanie substitúcií".

Klik na "Vytvorenie termu substitúciou" umožní vytvorenie nových termov aplikovaním vybranej substitúcie na vybrané termy z databázy a ich vloženie termov do ľubovoľnej množiny.

Klik na "Operácie nad množinami termov a substitúciami" umožňuje zjednotenie, prienik a rozdiel vložených množín ktorý môže byť vložený do vybranej aj novej množiny. Ďalej je možné tu vymazať ľubovoľnú z množín, rovnako ako je možné vymazať ľubovoľnú substitúciu.

Po kliknutí na "Výbery termov" si vyberáme množinu z ktorej má byť vybratý dopytovací (query) term. Tento výber nie je podmienkou. Po odoslaní požiadavky sa nám zobrazí formulár, kde si vyberieme návratovú podmienku (unifikácia, generalizácia, inštancia, premenovanie prememnných, identita). V časti dopytovací term vyberieme dopytovací term z jednej z množín, alebo sa zadá ručne textovou reprezentáciou v štandartnom prefixovom tvare. Potom si vyberieme množinu (množiny) v ktorej sa majú hľadať termy, spĺňajúce návratovú podmienku vzhľadom na dopytovací term. Nakoniec vyberieme množinu do ktorej sa majú zapísať termy spĺňajúce návratovú podmienku. Pokiaľ nevyberieme žiadnu tak sa výsledok iba vypíše na obrazovku. Ak si za návratovú podmienku vyberieme identitu tak sa výsledok hľadania nezapisuje nikde, iba sa vypíše, že v ktorej z prehľadávaných množín sa našiel term identický s dopytovacím termom. Pokiaľ vyberieme za návratovú podmienku napr. generalizáciu tak návratom budú termu ktoré sú generalizáciou dopytovacieho termu. Po vyplnení požiadaviek formulár odošleme a zobrazí sa nám stránka informujúca o tom ktoré termy boli vybraté za kandidátov, ktorí by mohli spĺňať návratovú podmienku. Títo kandidáti boli vybratí pomocou indexovacej techniky pre termy zvanej rozlišovacie stromy. Vedľa každého kandidáta je buď uvedená informácia o tom že term nevyhovel návratovej podmienke, alebo substitúcia v idempotentnom tvare. Pokiaľ je políčko vedľa termu prázdne tak term je totožný s dopytovacím a teda substitúcia je prázdna čiže je to identita.

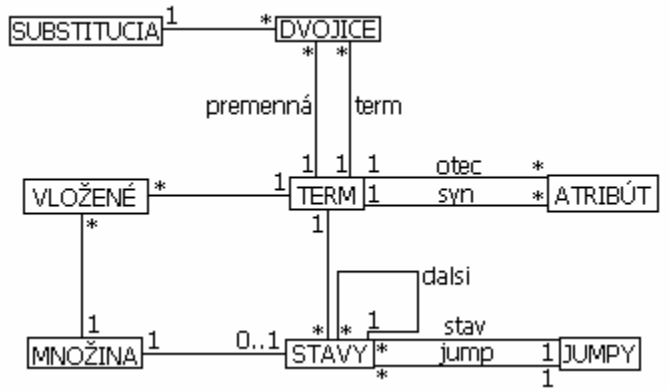
Pri "Hromadnej unifikácii" si vyberieme množiny pre ktoré sa má testovať, či sú v nich len termy ktoré majú spoločný unifikátor, teda substitúciu, ktorú keď aplikujeme na všetky termy v množine vznikne nám ten istý term. Ak takýto unifikátor existuje tak sa vypíše. V opačnom prípade sa vypíše informácia o tom, že daná množina sa nedá zunifikovať.

Po kliknutí na "Výpis tabuliek v databáze" sa nám zobrazí obsah všetkých tabuliek v databáze.

Týmto máme zabezpečené všetky požiadavky zo zadania. Celá štruktúra termov aj s indexmi a substitúciami je uložená v databáze a program umožňuje všetky zákaladné operácie nad množinami termov a substitúciami. Po niekoľkých úpravách by sa dal využiť aj v praxi tak, aby mohol slúžiť ako databáza termov pre niektoré ďalšie programy pracujúce s termami ako je napr. PROLOG. To však už nie je obsahom môjho zadania.

3.3. Realizácia programu

Rozpis tabuliek v databáze a ich vzájomné prepojenie je uvedené na nasledujúcom obrázku:



Jednotlivé tabuľky majú nasledujúce atribúty:

1. Tabuľka **VLOŽENÉ** bude v sebe uchovávať množinu všetkých vložených termov a zároveň aj ich príslušnosť k nejakej teórii. Má nasledovné atribúty:
   a. **ID** index vloženého termu
   b. **ID_TERM** odkaz na index hlavného termu v tabuľke TERM
   c. **MNOŽINA** číslo množiny do ktorej vložený term patrí
   d. **CAS** čas vloženia tohto termu
2. Tabuľka **TERM** bude uchovávať názov termu a jeho árnosť. Má tieto atribúty
   a. **ID** index hlavného termu resp. subtermov
   b. **NÁZOV** názov tohoto termu
   c. **ÁRNOSŤ** árnosť tohoto termu (v prípade nulovej árnosti to je konštanta (árnosť= 0) alebo premenná (árnosť= -1)
3. Tabuľka **ATRIBÚT** bude spravovať samotnú štruktúru termu teda pozíciu každého atribútu v hlavnom i vnorených termoch. Jej atribúty sú
   a. **ID_OTEC** index termu ktorého podtermy sú v stĺpci ID_SYN
   b. **ID_SYN** index podtermu termu s indexom ID_OTEC
   c. **PORADIE** umiestnenie podtermu s indexom ID_SYN vrámci árností v terme ID_OTEC.

4. Tabuľka **MNOŽINA** uchováva názvy jednotlivých množín
   a. **ID** index množiny
   b. **MENO** názov skupiny
   c. **STAV** číslo začiatočného stavu indexu termov patriaceho tejto množine
5. Tabuľka **STAVY** obsahuje čísla stavov indexu termov pre každú množinu
   a. **STAV** číslo stavu
   b. **NAZOV** názov termu na ktorý sa má prejsť do stavu uvedeného v atribúte DALSI. Pokiaľ je hodnota NULL tak je názov považovaný za premennú, ak je hodnota "" (prázdny reťazec) tak hodnota atribútu DALSI je id termu prislúchajúce k tomuto listu v indexovacom strome.
   c. **DALSI** číslo ďalšieho stavu, do ktorého sa má prejsť pri zhode s atribútom NAZOV, resp. id termu prislúchajúceho k listu
6. Tabuľka **JUMPY** obsahuje dojice čísiel stavov v tabuľke stavy v súlade s indexovaním rozlišovacími stromami a jeho štruktúrou jump_table.
   a. **STAV** číslo stavu z ktorého sa skáče pri narazení na premennú
   b. **JUMP** číslo stavu do ktorého sa skáče pri narazení na premennú
7. Tabuľka **DVOJICE** bude v sebe uchovávať dvojice premenná -> term ktoré tvoria jednotlivé substitúcie
   a. **ID** index dvojice
   b. **ID_SUB** odkaz na index substitúcie, do ktorej táto dvojica patrí
   c. **PREMENNA** odkaz na index do tabulky term ktorý zodpovedá premennej na ľavej dvojice
   d. **TERM** odkaz na index do tabulky term ktorý zodpovedá termu na pravej dvojice
   e. **CAS** čas vloženia dvojice
8. Tabuľka **SUBSTITUCIA** bude v sebe uchovávať dvojice premenná -> term ktoré tvoria jednotlivé substitúcie
   a. **ID** index substitúcie
   b. **MENO** názov substitúcie

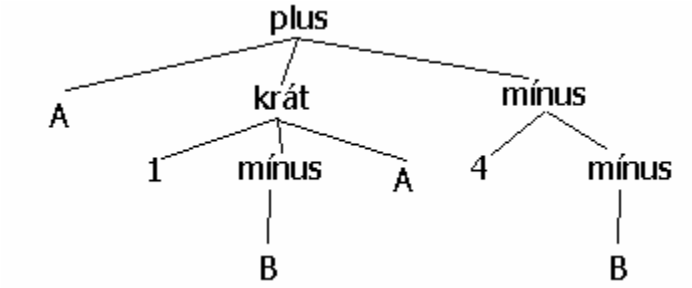
3.4. Vkladanie do databázy

V tejto časti si opíšeme spôsob ukladania termu a ďalších štruktúr do databázy a funkcie ktoré sa pri tom používajú.

Všetky funkcie potrebné na uloženie tremu do databázy, ako aj všetky ostatné obslužné funkcie sú obsiahnuté v súbore term_kniznica.php. Po jeho importovaní tak získame knižnicu pomocou ktorej, by bolo možné ukladať množinu termov do databázy aj pomocou iného obslužného programu, ako toho, čo som vytvoril v tejto diplomovej práci. Dovoľujem si pripomenúť, že aj spomínanú knižnicu som vytváral vrámci tejto diplomovej práce. Všetky funkcie som vyvýjal sám, okrem tých, ktoré boli spomínané v teoretickej časti mojej diplomovej práce. Tie som musel iba upraviť tak, aby fungovali nad mojou dátovou štruktúrou.

Vstupom pri vkladaní termu do databázy je textová reprezentácia termu a číslo množiny, do ktorej sa má tento term zapísať. Term je definovaný ako konštanta, premenná alebo n-árny funkčný symbol, ktorého atribútmi sú termy. Term sa tým pádom dá interpretovať ako akási

stromová štruktúra, kde koreňom je názov najvrchnejšieho (koreňového) termu a listami sú konštanty alebo premenné. Napríklad term s textovou reprezentáciou *plus(A,krát(1,mínus(B),A),mínus(4,mínus(B)))* by vyzeral takto (predpokladajme že množina, do ktorej sa má term vložiť má číslo 1.):



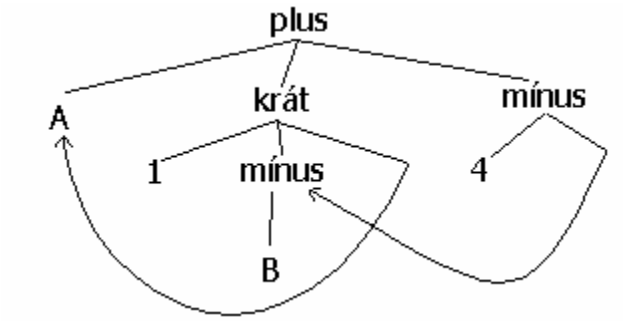
Tento term by sa uložil nasledovne:

Vložené		
id	**id_term**	**množina**
1	1	1

Term		
id	**názov**	**árnosť**
1	plus	3
2	A	-1
3	krát	3
4	mínus	2
5	1	0
6	mínus	1
7	B	-1
8	4	0

Atribút		
id_otec	**id_syn**	**poradie**
1	2	1.
1	3	2.
1	4	3.
3	5	1.
3	6	2.
3	2	3.
4	8	1.
4	6	2.
6	7	1.

Toto však už nie je stromová štruktúra, pretože každý term sa tu vyskytuje len raz. Situácia teda vyzerá nasledovne:



Podobne ako termy A a mínus(B), by sa mohli opakovať aj väčšie podtermy a v databáze by sa ich implementácia vyskytovala iba raz. Zdôrazňujem, že v databáze, pretože to neplatí iba vrámci jedného termu, ale keby iný term obsahoval nejakú totožnú časť (prípadne aj celý obsah termu) s už existujúcou v databáze, tak sa na ňu jednoducho odvolá. Takéto riešenie tým, že zaberie menej miesta, spôsobí rýchlejšie vyhľadanie termu v databáze a neskôr nám toto zdieľanie podtermov pomôže pri unifikácii a iných návratových podmienkach.

Takže teraz, keď vieme, čo by sme chceli dosiahnuť, pozrime sa na to, ako to spraviť. Všetky funkcie, ktoré tu budem uvádzať sa nachádzajú v súbore term_kniznica.php.

Pokiaľ nemáme vytvorenú množinu, do ktorej chceme term vložiť, využijeme funkciu:

int **vytvor_množinu (**string **text)**
- Vytvorí množinu s názvom **text** a vráti číslo tejto množiny v databáze. Ak množina s týmto názvom už existuje tak len vráti jej id.

Vďalšom majme textovú reprezentáciu termu v premennej $vstup a číslo množiny v premennej $mnozina. Zavoláme funkciu nastav_term ($vstup).

array **nastav_term (**string **text)**
- Vykoná overenie správnosti zápisu textovej reprezentácie termu. Ak **text** nie je term vráti false. Inak vráti pole $pole = array ($term,$atribut), kde $term = array([1] => $id, [2] => $nazov, [3] => $arnost) a $atribut = array([1] => $id_otec, [2] => $id_syn, [3] => $poradie). $id, $nazov a $arnost sú polia predstavujúce stĺpce tabuľky term. $id_otec,$id_syn a $poradie sú polia predstavujúce stĺpce tabuľky atribut. V poli $pole je tak uchovaná celá štruktúra termu ale ešte s opakovaním podtermov z textovej reprezentacie termu. Nie sú ešte nastavené árnosti premenných na -1.
- napr. zavoláme funkciu nastav_term("f(g(a,b),b)"). Výstupom bude pole $pole, ktoré by sa dalo znázorniť tabuľkou:

$pole					
$term			$atribut		
$id	$nazov	$arnost	$id_otec	$id_syn	$poradie
1	f	2	1	2	1
2	g	2	2	3	1
3	a	0	2	4	2
4	b	0	1	5	2
5	b	0			

reálne ide o pole:

```
Array(   [0] => Array (    [1] => Array(     [0] => 1
                                             [1] => 2
                                             [2] => 3
                                             [3] => 4
                                             [4] => 5)
                           [2] => Array(     [0] => f
                                             [1] => g
                                             [2] => a
                                             [3] => b
                                             [4] => b)
                           [3] => Array(     [0] => 2
                                             [1] => 2
                                             [2] => 0
                                             [3] => 0
                                             [4] => 0))
```

```
[1] => Array(     [1] => Array(     [0] => 1
                                    [1] => 2
                                    [2] => 2
                                    [3] => 1)
                  [2] => Array(     [0] => 2
                                    [1] => 3
                                    [2] => 4
                                    [3] => 5)
                  [3] => Array(     [0] => 1
                                    [1] => 1
                                    [2] => 2
                                    [3] => 2)))
```

Pokiaľ nebolo vrátené false, tak výsledné pole uložíme do pomocnej premennej. Nech je to premenná $p1. V tomto poli, ako bolo uvedené, ešte nie sú podtermy jedinečné a nie je ani vyznačené, ktoré termy sú premenné. Teraz zavoláme funkciu rovnake_prec($p1).

array **rovnake_prec (**array **pole)**

- vstupom aj výstupom tejto funkcie je $pole = array ($term,$atribut). Jeho bližší popis je uvedený pri funkcii nastav_term($vstup). Táto funkcia odstráni zo štruktúry termu, danej poľom na vstupe, všetky duplicitné podtermy a nastaví príslušné odvolávky v poli atribút. Ak niečo zlyhá, funkcia vráti false.
- napr. zavoláme funkciu rovnake_prec($p1). $p1 je pole ktoré vrátila funkcia nastav_term("f(g(a,b),b)") z predchádzajúceho príkladu.Výstupom bude pole $pole, ktoré by sa dalo znázorniť tabuľkou:

$pole					
$term			$atribut		
$id	$nazov	$arnost	$id_otec	$id_syn	$poradie
1	f	2	1	2	1
2	g	2	2	3	1
3	a	0	2	4	2
4	b	0	1	4	2

Ak je výsledok pole, tak ho uložíme do pomocného poľa. Nech je to premenná $p2. V tomto poli je už každý term jedinečný. Stačí nám už len nastaviť pre premenné árnosť=-1. Zavoláme funkciu nastav_premenne($p2).

array **nastav_premenne (**array **pole)**

- vstupom aj výstupom tejto funkcie je $pole = array ($term,$atribut). Jeho bližší popis je uvedený pri funkcii nastav_term($vstup). Táto funkcia určí, ktorý podterm s doterajšou arnostou rovnou nule, bude premennou a teda mu nastaví árnosť -1. Za premennú je považované každé slovo ktoré nie je v úvodzovkách a nieje to číslo.
- napr. zavoláme funkciu nastav_premenne($p2). $p2=rovnake_prec(nastav_term("f(g(a,b),b)")) z predchádzajúceho príkladu.Výstupom bude pole $pole, ktoré by sa dalo znázorniť tabuľkou:

$pole					
$term			$atribut		
$id	$nazov	$arnost	$id_otec	$id_syn	$poradie
1	f	2	1	2	1
2	g	2	2	3	1
3	a	-1	2	4	2
4	b	-1	1	4	2

Výsledok uložíme do pomocnej premennej $p3. Teraz už máme štruktúru, ktorá by mohla byť vložená priamo do databázy. My však chceme, aby v celej databáze bol každý podterm jedinečný, bez dvojníka. O vkladanie do databázy s rešpektovaním tejto podmienky sa stará funkcia vloz_term_do_DB(). My však zavoláme funkciu vloz_term_do_mnoziny($t3,$mnozina), ktorá ukrem toho, že zavolá funkciu vloz_term_do_DB($t3), priradí tento term k množine s číslom $mnozina a zavolá funkciu na vloženie tohoto termu do indexu množiny, ktorý sa naplno využije pri výberoch termov.

vloz_term_do_mnoziny (array **pole,** int **mnozina)**

- **pole** je $pole = array ($term,$atribut). Jeho bližší popis je uvedený pri funkcii nastav_term($vstup). **Mnozina** je číslo množiny do ktorej sa má tento term vložiť. Funkcia vloží term daný poľom **pole** do databázy, zaradí ho k množine s číslom **mnozina** a rozšíri index tejto množiny o tento term. Ak sa vkladanie nepodarí, funkcia vráti false.

int **vloz_term_do_DB (**array **pole)**

- **pole** je $pole = array ($term,$atribut). Jeho bližší popis je uvedený pri funkcii nastav_term($vstup). Funkcia vloží term daný poľom **pole** do databázy. Pri tom kontroluje, aby žiaden podterm nebol v databáze duplicitný. Ak sa vkladanie nepodarí, funkcia vráti false. Ak sa to podarí, funkcia vráti id vloženého termu v tabuľke TERM.

vloz_index (int **id_vlozene)**

- **id_vlozene** je číslo záznamu v tabuľke VLOZENE. Funkcia vloží novú vetvu do indexu príslušnej množiny (s riadkom v tabuľke VLOZENE dostaneme aj číslo množiny aj číslo termu, ktorý treba vložiť do indexu).

Indexovanie je v mojej diplomovej práci realizované cez metódu rozlišovacích stromov (discrimination trees). Spôsob práce tejto metódy je bližšie opísaný v teoretickej časti mojej diplomovej práce. Jej základom je vytvorenie p-stringov a jump listov z dopytovacieho aj indexovaných (vložených v niektorej množine) termov. O ich tvorbu sa starajú funkcie pstring a pstring_DB.

array **pstring (**int **id,** array **vstup)**

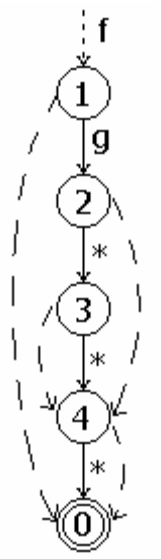
- vstupom tejto funkcie je $vstup = array ($term,$atribut). Jej bližší popis je uvedený pri funkcii nastav_term($vstup). **id** je číslo termu z poľa vstup, z ktorého sa má vytvoriť p-string. Funkcia vráti pole $pole=>($pole[0]=$next,$pole[1]=$jump) obsahujúce názvy termov (v $next), ktoré tvoria pstring termu s id-ckom $id v poli $vstup, názvy premenných sú nahradené hodnotou "", ktorá zodpovedá ľubovoľnej

premennej (lebo v indexe nezáleží na názvoch premenných), $jump obsahuje čísla políčok v poli $next a predstavuje jump list tohto p-stringu.

– napr. majme na vstupe pole, ktoré je výstupom z príkladu uvedeného pri funkcii $t3=nastav_premenne($t2). Výsledkom bude pole ktoré by sa dalo znázorniť tabuľkou:

$next	$jump
g	0
""	4
""	3
""	4
" "	0

Možno viac napovie reprezentácia automatom:



Z obrázku je zjavné, že vo výsledku nie je zahrnutý hlavný (koreňový) term f, pretože pri ňom je vždy za ďalší považovaný jeho prvý syn a v prípade volania jumptable je vždy odkaz na koniec p-stringu. Prerušovanou čiarou je vyznačený jump list p-stringu, plnou čiarou next.

array **pstring_DB (**int **id)**

– **id** je číslo termu v tabuľke TERM, z ktorého sa má vytvoriť p-string. Funkcia vráti pole s rovnakými vlastnosťami ako funkcia pstring()

Okrem vkladania termov a indexov množín, umožňuje táto knižnica aj vkladanie substitúcií. Substitúcia je chápaná ako množina dvojíc premenná => term. Na vytvorenie novej substitúcie sa používa funkcia vytvor_substituciu($text)

int **vytvor_substituciu (**string **text)**

– Vytvorí substitúciu s názvom **text** a vráti jej číslo v tabuľke SUBSTITUCIA. Ak substitúcia s týmto názvom už existuje tak len vráti jej id.

Vkladanie dvojíc do substitúcie sa robí postupne dvojica za dvojicou. Využijeme funkciu dvojica_do_substitúcie().

dvojica_do_substitucie($prem,$term,$sub,[$idempot])
- Vloží do substitúcie s id-čkom $sub v tabuľke SUBSTITUCIA, dvojicu $prem=>$term zadaných textovo, ak je $idempot=true, tak sa v substitúcii zachová idempotentnosť. Ak sa niečo nepodarilo, teda ak textové reprezentácie na vstupe nie sú korektné, alebo pri požiadavke idempotentnosti nie je možné idempotentnosť zachovať, tak sa vráti false.

3.5. Mazanie z databázy

V tejto časti si opíšeme funkcie, ktoré sa používajú pri mazaní termu a ostatných štruktúr z databázy.

Na vymazanie termu sa využíva funkcia vymaz_term_z_vlozenych()

vymaz_term_z_vlozenych(int **id)**
- Vymaže term s id-čkom **id** v tabuľke VLOZENE. Pokiaľ sa term s takýmto id-čkom nenájde tak vráti false. Okrem toho vymaže vetvu z indexu príslušnej množiny.

vymaz_index (int **id_vlozene)**
- **id_vlozene** je číslo záznamu v tabuľke VLOZENE. Funkcia vymaže vetvu z indexu príslušnej množiny (s riadkom v tabuľke VLOZENE dostaneme aj číslo množiny aj číslo termu, ktorý treba vyhodiť z indexu).

Na mazanie celej množiny je funkcia vymaz_mnozinu()

vymaz_mnozinu(int **id)**
- Vymaže množinu s číslom **id** v tabuľke MNOZINA. Zároveň odstráni aj všetky termy, ktoré v nej boli zapísané. Odstráni aj celý svoj index.

Na mazanie substitúcií je funkcia vymaz_substituciu()

vymaz_substituciu(int **id)**
- Vymaže substitúciu s číslom **id** v tabuľke SUBSTITUCIA. Zároveň odstráni aj všetky dvojice premenná=>term, ktoré v nej boli zapísané.

vymaz_dvojicu(int **id)**
- Vymaže term s id-čkom **id** v tabuľke DVOJICE. Pokiaľ sa dvojica s takýmto id-čkom nenájde tak vráti false.

3.6. Výbery termov

Tu si opíšeme ako dostať z databázy z termy, ktoré splňujú rôzne podmienky.

Najjednoduchšou návratovou podmienkou je identita. Na nájdenie termu zhodujúceho sa s dopytovacím termom použijeme funkciu najdi_term_v_DB()

int **najdi_term_v_DB(**string **text)**
- Vráti index v tabuľke TERM, ktorý zodpovedá textovej reprezentácii termu v premennej **text**. Pokiaľ takýto term v databáze nie je alebo **text** nie je syntakticky správne zadaný term tak vráti false.

Pre návratovú podmienku inštancia budeme potrebovať tieto funkcie:

array **kandidati_instancie(**int **dopyt,**int **mnozina)**
- Vráti pole id-čiek z tabuľky termy, ktoré sú kandidátmi návratovej podmienky inštancie pre dopytovací term zadaný indexom z tabuľky TERM, **mnozina** je index z tabuľky MNOZINA a určuje množinu, v ktorej sa majú hľadať kandidáti. Kandidáti sa hľadajú v indexe tejto množiny metódou opísanou pri indexovacej technike rozlišovacích stromov spomínanou v teoretickej časti mojej diplomovej práce.

array **instancia_v_DB(&**array **pole,**int **id1,**int **id2)**
- ak term s id-ckom **id1** v tabuľke TERM nie je inštanciou termu **id2** v tabulke TERM tak funkcia vráti false. Inak vráti pole $subst = array($prem,$term), kde $prem[i]=>$term[i] je i-ta dvojica výslednej substitúcie, ktorá je idempotentná a po jej aplikovaní na term s id-čkom **id2** vznikne term s id-čkom **id1**. Premenné $prem a $term sú polia id-čiek termov z vráteného poľa **pole** = array ($term_,$atribut_). Bližší popis tohto poľa je uvedený pri funkcii nastav_term($vstup).

array **instancia(&**array **pole,**int **id1,**int **id2)**
- **pole** je pole = array ($term_,$atribut_), ktoré má štruktúru uvedenú pri funkcii nastav_term($vstup), predpokladá sa však, že v ňom sú oba vstupné termy so zdieľanými podtermami. ak term s id-ckom **id1** v poli **pole** nie je inštanciou termu **id2** v poli **pole** tak funkcia vráti false. Inak vráti pole $subst = array($prem,$term), kde $prem[i]=>$term[i] je i-ta dvojica výslednej substitúcie, ktorá je idempotentná a po jej aplikovaní na term s id-čkom **id2** vznikne term s id-čkom **id1**. Premenné $prem a $term sú polia id-čiek termov z vráteného poľa **pole** = array ($term_,$atribut_), ktoré nie je zhodné so vstupným poľom. Pole **pole** bolo modifikované upraveným unifikačným algoritmom "Rekurzia na grafoch termov" uvedenom v teoretickej časti mojej DP. Túto úpravu som spravil sám.

string **textova_r(**int **id,**array **pole)**
- Funkcia vráti textovú reprezentáciu termu s id-čkom **id** v poli **pole** = array ($term,$atribut). Bližší popis tohto poľa je uvedený pri funkcii nastav_term($vstup).

Na vrátenie kandidátov inštancie dopytovacieho termu s id-čkom $dopyt v tabuľke TERM z množiny s indexom $mnozina z tabuľky MNOZINA teda najprv zavoláme funkciu kandidati_instancie($dopyt,$mnozina). Výsledok uložíme napríklad do premennej $kandidati. Vcykle zavoláme funkciu instancia_v_DB($pole,$dopyt,$kandidati[$i]), kde premennou $i prechádzame všetky indexy poľa $kandidati. Výsledok funkcie zakaždým uložíme napr. do premennej $subst. Pokiaľ to je pole tak v ňom máme dvojice výslednej substitúcie. Textovú reprezentáciu týchto dvojíc získame zavolaním nasledovných funkcií:

textova_r($subst[0][$j],$pole)."=>".textova_r($subst[1][$j],$pole),

kde $j je premenná prechádzajúca cez všetky dvojice poľa $subst. Pokiaľ je pole $subst prázdne, tak dopytovací term je identický s kandidátskym termom. Ak $subst nie je pole tak dopytovací term nie je inštanciou kandidátskeho termu.

Pre ostatné návratové podmienky fungujú nasledujúce funkcie:
pre unifikáciu sú to :

array **kandidáti_unifikacie(**int **dopyt,**int **mnozina)**
array **unifikuj_v_DB(&**array **pole,**int **id1,**int **id2)**
array **unifikuj(&**array **pole,**int **id1,**int **id2)**

pre generalizáciu sú to :

array **kandidáti_generalizacie(**int **dopyt,**int **mnozina)**
array **generalizacia_v_DB(&**array **pole,**int **id1,**int **id2)**
array **generalizacia(&**array **pole,**int **id1,**int **id2)**

pre variáciu sú to :

array **kandidáti_variacie(**int **dopyt,**int **mnozina)**
array **variacia_v_DB(&**array **pole,**int **id1,**int **id2)**
array **variacia(&**array **pole,**int **id1,**int **id2)**

Vo funkciách generalizacia() a variácia() som takisto upravil algoritmus "Rekurzia na grafoch termov". Takmer bez zmeny ostal vo funkcii unifikuj().

Pre úplnosť uvediem ešte funkciu na návrat textovej reprezentácie termu z databázy

string **textova_r_DB(**int **id,**array **pole)**

– Funkcia vráti textovú reprezentáciu termu s id-čkom **id** v tabuľke TERM.

3.7. Ostatné funkcie

Tu uvediem funkcie, ktoré podľa mňa už nie sú nevyhnutné pre úplnú prácu s množinami termov a výbermi termov z množín. Sú to však funkcie, ktoré sa prirodzene žiadajú a uľahčujú prácu.

Ako prvé uvediem funkcie, ktoré robia základné množinové operácie s množinami termov.

zjednotenie_mnozin(int **prva**,int **druha**,int **ciel)**

– Funkcia urobí zjednotenie množín s id-čkami **prva** a **druha** v tabuľke MNOZINA a výsledok vloží do množiny s id-čkom **ciel** v tabuľke MNOZINA.

prienik_mnozin(int **prva**,int **druha**,int **ciel)**

– Funkcia urobí prienik množín s id-čkami **prva** a **druha** v tabuľke MNOZINA a výsledok vloží do množiny s id-čkom **ciel** v tabuľke MNOZINA.

odcitanie_mnozin(int **prva**,int **druha**,int **ciel)**

– Funkcia urobí rozdiel množín s id-čkami **prva** a **druha** v tabuľke MNOZINA a výsledok vloží do množiny s id-čkom **ciel** v tabuľke MNOZINA.

Ďalšími funkciami je možné aplikovať substitúcie na termy.

array **substituuj(**int **id,**string **z,**array **do)**

– Funkcia vráti pole $pole = array ($term,$atribut), ktoré má štruktúru uvedenú pri funkcii nastav_term($vstup), ktoré je výsledkom aplikácie substitúcie **z** =>**do** na term s id-čkom **id** v tabuľke TERM. Vo výslednom poli sa ešte opakujú rovnaké podtermy. **z** je premenná zadaná reťazcom a **do** je pole $pole_ = array ($term_,$atribut_)

int **substituuj_v_DB(**int **id,**int **z,**int **do)**

– Funkcia vráti id-čko termu v tabuľke TERM, ktorý je výsledkom aplikácie substitúcie **z** =>**do** na term s id-čkom **id** v tabuľke TERM. **z** a **do** sú tiež id-čka v tabuľke TERM. Starý term sa neodstraňuje.

int **substitucia_termu_v_DB(**int **id,**int **sub)**

– Funkcia vráti id-čko termu v tabuľke TERM, ktorý je výsledkom aplikácie substitúcie s id-čkom **sub** v tabuľke SUBSTITUCIA na term s id-čkom **id** v tabuľke TERM. Starý term sa neodstraňuje.

Poslednou funkciou je funkcia používaná pri hromadnej unifikácii. Po jej zavolaní sú premenné pripravené na test unifikovateľnosti s ďalším termom v množine.

nastav_pre_unifikaciu(array **pole**,int **id**,array **subst**,int **kandidat)**

– Funkcia nastaví argumenty potrebné pre hromadnú unifikáciu. Vstupno-výstupné $pole = array ($term,$atribut) bude obsahovať iba term začínajúci id-čkom **id** a substitúciu **subst** = array($prem,$term_), zároveň sa nastavia nové id-čka, ktoré budú zodpovedať aktuálnej databáze. Nakoniec sa do poľa $pole = array ($term,$atribut) pridá ďalší kandidát z databázy určený premennou **kandidat** určujúcou id-čko v tabuľke TERM.

Záver

V tejto práci som sa zameral na problém uchovania množín termov v databáze.

V časti o teórii unifikácie som sa venoval niekoľkým unifikikačným algoritmom, dôkazom ich korektnosti a úplnosti a odhadom zložitosti. V časti o indexovaní termov som previedol analýzu niekoľkých štruktúr na uchovávanie termov a indexov termov. Podrobne som rozpracoval 2 indexovacie techniky indexovania ciest (path indexing) a rozlišovacích stromov (discrimination trees).

Praktická časť využíva z teoretickej časti algoritmus unifikácie na grafoch termov, ktorý som po menších úpravách použil aj na hromadnú unifikáciu, generalizáciu, inštanciu aj varianty. Na indexovanie som využil indexovaciu techniku rozlišovacích stromov.

Všetky funkcie potrebné na prácu s množinami termov pri uchovávaní a výbere z databázy som zahrnul do samostatnej knižnice. Týmto sa táto knižnica dá využiť aj pre iné aplikácie ako je tá, ktorú som k tejto knižnici vytvoril.

Keďže som program programoval v jazyku PHP je ho možné spúšťať cez internetový prehliadač z ľubovoľného miesta pripojeného na internet. Jazyk PHP spolu s databázovým serverom MySQL a webovým serverom Apache sú pod GNU licenciou a teda nie je žiaden problém si tieto programy stiahnuť z internetu a spustiť spolu s mojim programom na ľubovoľnom počítači.

Použitá literatúra

1) R.Sekar, I.V.Ramakrishnan, Andrei Voronkov – Term indexing in Alan Robinson, Andrei Voronkov - Handbook of automated reasoning, © 2001 Elsevier Science Publishers B.V.
2) Franz Baader, Wayne Snyder – Unification theory in Alan Robinson, Andrei Voronkov - Handbook of automated reasoning, © 2001 Elsevier Science Publishers B.V.
3) Paul Singleton, O. Pearl Brereton - Storage and Retrieval of First-Order Terms using a Relational Database
4) Stig Sæther Bakken, Alexander Aulbach a kol. - PHP manual, © 1997-2002 the PHP Documentation Group
5) MySQL Reference Manual for version 3.23.11-alpha.

Príloha

Abstrakt

Táto práca opisuje problém uchovávania množiny termov v databáze. Je tu predstavených niekoľko unifikačných algoritmov a zložitosť ich výpočtov. Porovnáva sa tu niekoľko štruktúr na ukladanie termov a indexov termov. Spracováva aj 2 indexovacie techniky indexovania ciest a rozlišovacích stromov. Súčasťou tejto práce je aj popis knižnice vytvorenej v jazyku PHP, pomocou ktorej je možné vkladať termy do databázy a vyberať ich v súlade s podmienkou unifikácie, generalizácie, inštancie alebo varianty s využitím indexovania termov.

Abstract

This paper describes the problem of storing a set of terms in a database. Some unification algorithms and the complexity of their calculation is presented. Some structures for keeping terms and for term indices are compared. It also compares two path indexing methods, path indexing and discrimination trees. A component of this work is the description of a library created in PHP which allows the storage and retrieval of terms in the database and with respect to the conditions of unification, generalization, instantation and variation with the use of term indexing.