Induktivní logické programování

Luboš Popelínský
Katedra teoretické informatiky
Fakulta informatiky Masarykovy university
Botanická 68a, CZ - 602 00 Brno
Email: popel@fi.muni.cz

6. ledna 1999

Abstrakt

Induktivní logické programování (ILP) zkoumá oblast, kde se potkává induktivní učení s predikátovou logikou 1. řádu. Systém se induktivně učí, jestliže na základě příkladů z nějaké třídy předmětů nebo z příkladů chování programu dokáže najít obecnou charakteristiku této třídy předmětů resp. tento program v daném logickém kalkulu. Hlavním cílem ILP je potom vývoj teorie, algoritmů a systémů pro induktivní odvozování v predikátovém počtu 1. řádu.

Práce popisuje základní pojmy a techniky ILP, základní úlohu ILP, obecný generický algoritmus a jeho varianty. Soustředí se na top-down ILP, uvádí elementární specializační operátory a pojem specializačního grafu. Následuje popis interaktivního top-down systému MIS. Jsou podrobně popsány jak strategie tvorby nových klauzulí tak tři možné interaktivní režimy. Je uvedena možnost mechanizace orákula. Kapitolu uzavírá příklad syntézy predikátu `reverse/2`. Algoritmus a omezující podmínky (bias) programu $MARKUS$, následníka systému MIS, jsou popsány v následující kapitole. Závěrem práce upozorňuje na některé neřešené (či nedostatečně řešené) obecné problémy ILP a shrnuje nedostatky top-down systémů. Končí přehledem možných směrů vlastního výzkumu v oblasti top-down ILP.

Obsah

1 Úvod

Induktivní logické programování (ILP) [12, 21] zkoumá oblast, kde se potkává induktivní učení s predikátovou logikou 1. řádu. Systém se induktivně učí, jestliže na základě konkrétních příkladů z nějaké třídy předmětů nebo z příkladů chování programu dokáže najít obecnou charakteristiku této třídy resp. tento program v daném logickém kalkulu. Hlavním cílem ILP je potom vývoj teorie, algoritmů a systémů pro induktivní odvozování v predikátovém počtu 1. řádu.

Pro danou množinu faktů hledáme takovou logickou formuli, která tato fakta uspokojivým způsobem vysvětluje. Přesněji řečeno, pro danou množinu pozitivních príkladů E^+ – případně doplněnou o množinu negativních příkladů E^- – a množinu axiomů B se snažíme vytvořit logický program P takový, že v teorii $B \cup P$ jsou dokazatelné všechny pozitivní příklady a žádný negativní. Množina B se označuje jako *doménová(apriorní) znalost.*

Mějme například následující doménovou znalost B, v níž $děd(X,Y)$ platí, pokud X je *děd*em Y, a relaci *děd* popsanou pozitivními (E^+) a negativními (E^-) příklady

$$B \;=\; \left\{ \begin{array}{l} děd(X,Y) \;\leftarrow\; otec(X,Z) \wedge rodič(Z,Y). \\ otec('Ivar','Eva'). \\ matka('Eva','Jan'). \\ matka('Eva','Tom'). \end{array} \right.$$

$$E^+ = \left\{ \begin{array}{l} děd('Ivar','Jan'). \\ děd('Ivar','Tom'). \end{array} \right.$$

$$E^- = \left\{ \begin{array}{l} děd('Jan','Ivar'). \\ děd('Tom','Jan'). \end{array} \right.$$

Obr. 1a.

Potom logická formule P

$$P \;=\; rodič(X,Y) \;\leftarrow\; matka(X,Y).$$

je jedním z možných řešení, protože $B \cup P$ *pokrývá* (covers) všechny pozitivní příklady, tj. v této teorii jsou dokazatelné všechny pozitivní příklady a není dokazatelný žádný z protipříkladů $e \in E^-$, tj.

$$\forall e^+ \in E^+ \; \forall e^- \in E^- \; : \; B \cup P \vdash e^+ \wedge B \cup P \not\vdash e^- \; .$$

Popsaná situace je příkladem revize teorie (theory revision) [22], kdy opravujeme danou neprázdnou teorii, aniž předem víme, který z predikátů je chybný. Nejčastěji však stojíme před úkolem najít definici jednoho predikátu na základě příkladů jeho chování (začínáme s chybnou prázdnou teorií). Například hledáme definici predikátu $rodi\check{c}(Z, Y)$, máme-li k dispozici stejnou doménovou znalost a následující učící množinu

$$E^+ = \begin{cases} rodi\check{c}('Eva','Jan'). \\ rodi\check{c}('Eva','Tom'). \end{cases}$$

$$E^- = \begin{cases} rodi\check{c}('Tom','Eva'). \\ rodi\check{c}('Jan','Eva'). \\ rodi\check{c}('Jan','Tom'). \\ rodi\check{c}('Tom','Jan'). \\ rodi\check{c}(X, X). \end{cases}$$

Obr. 1b.

Výsledkem bude stejná klauzule jako v předchozím příkladu, tj. $rodi\check{c}(X, Y) \leftarrow matka(X, Y)$. ■

Uvedené ukázky charakterizují oblast ILP, kdy hledáme popis nějakého konceptu nebo objevujeme novou znalost. Následující příklad je ukázkou automatického logického programování. Budeme chtít najít program $reverse(X, Y)$, který pro vstupní seznam X najde seznam Y s opačným pořadím prvků. Doménová znalost obsahuje program $concat(X, Y, Z)$, který přepíše do proměnné Z seznam X následovaný prvkem Y [1], např. $concat([p, a, s], t, [p, a, s, t])$. Program $reverse/2$ je popsán pozitivními příklady E^+.

$$B = \begin{cases} concat([], X, [X]). \\ concat([X|Y], Z, [X|U]) \; \leftarrow \; concat(Y, Z, U). \end{cases}$$

$$E^+ = \begin{cases} reverse([], []). \\ reverse([1, 2], [2, 1]). \\ reverse([3, 4, 5, 6, 7], [7, 6, 5, 4, 3]). \end{cases}$$

[1] [ ] značí prázdný seznam. $[X|Y]$ je seznam začínající prvkem X, za nímž následuje seznam Y.

Obr. 1c.

Řešením je například

$$P = \begin{cases} reverse([],[]). \\ reverse([X|Y],Z) \leftarrow reverse(Y,U) \wedge concat(U,X,Z). \end{cases}$$

Právě takovými úlohami, jako je tato, se budeme v dalším zabývat. Učící množina se bude vždy skládat z příkladů chování právě jednoho predikátu a cílem učení bude najít úplnou a bezespornou proceduru logického programu – definici predikátu.

■

V následující části podáváme přehled základních pojmů a technik induktivního logického programování. V kapitole 2.1 nejdříve charakterizujeme základní úlohu ILP. Obecný algoritmus uvádíme v kapitole 2.2 . Možné varianty ILP systémů jsou diskutovány v kapitole 2.3 . Popis jednoho ze základních typů – top-down ILP – obsahuje kapitola 3. Poté popisujeme top-down systémy MIS v kapitole 4 a $Markus$ v kapitole 5. Závěrem shrneme nedostatky těchto systémů, uvedeme některé neřešené problémy základního modelu ILP a možné směry vlastního výzkumu v oblasti top-down ILP.

2 Základní pojmy a techniky ILP

2.1 Základní úloha ILP

Uvedeme nejčastější a nejdůležitější variantu, specifikaci pomocí příkladů. Obecnou úlohu lze najít např. v [12].

Specifikace pomocí příkladů (example settings, specification by examples) znamená, že příklady mohou být vyjádřeny jen jako fakta bez proměnných, stejně jak tomu je v našich dvou příkladech. Jak program P tak doménová znalost B se skládají z logických formulí $A \leftarrow A_1 \wedge ... \wedge A_2$, kde A je pozitivní literál, A_i jsou pozitivní nebo negativní. Základní úlohu ILP pak definujeme takto :

Pro dané množiny pozitivních a negativních příkladů E^+ a E^- a množinu axiomů B takových že

Apriorní bezespornost: $\forall e \in E^- : \; B \not\vdash e$
Apriorní nutná podmínka: $\exists e \in E^+ : \; B \not\vdash e$

hledáme P takové, že

Aposteriorní úplnost: $\forall e \in E^+ : \; B \cup P \vdash e$

Aposteriorní bezespornost: $\forall e \in E^- : \; B \cup P \not\vdash e$

Tyto požadavky však v příkladu na Obr.1a. splňuje i formule $d\check{e}d('Ivar','Jan')$ $\wedge d\check{e}d('Ivar','Tom')$. Od P proto navíc očekáváme, že bude **zobecněním** příkladů, tj. bude platit i pro některé příklady, které se nevyskytly v učící množině.

2.2 Obecný generický algoritmus

Nahoře uvedená definice základní úlohy neposkytuje návod, jak P nalézt. Ten je obsahem této kapitoly, v níž uvádíme obecný algoritmus pro výpočet formule P. Některé neřešené problémy tohoto algoritmu jsou diskutovány v kap. 6. Jednotlivé ILP systémy se liší „jen" implementací jednotlivých funkcí tohoto algoritmu. QH obsahuje nalezená řešení.

Vstup: B, E^+, E^-
$QH :=$ *inicializuj*(B, E^+, E^-) ;
while not(*kriterium_ukončení* (QH)) **do**
 zruš H z QH ;
 zvol odvozovací pravidla $r_1, ..., r_k \in R$;
 aplikací pravidel $r_1, ..., r_k$ na H vytvoř množinu $H1$
$QH := QH \cup H1$;
 zruš některé prvky z QH ;
Výstup: *vyber_hypotézu* $\mathrm{P} \in QH$

Generický algoritmus pro ILP

Množinu přípustných hypotéz označme QH. Algoritmus začíná s nějakou počáteční množinou možných řešení QH (funkce *inicializuj*). V cyklu while se z QH zruší (viz funkce *zruš*) jedna z hypotéz H a na ní se aplikují odvozovací pravidla $r_1, ..., r_k$. Dostáváme nové hypotézy $H1 = \{H_1, ..., H_n\}$ a ty přidáme do QH. Poté se z QH zruší málo nadějné prvky. Algoritmus skončí, jestliže na množině QH platí *kriterium_ ukončení*. Množina QH může obecně obsahovat více než jedno řešení.

Algoritmus má následující parametry:

- Funkce *inicializuj* vytváří počáteční množinu hypotéz;
- R je množina inferenčních pravidel;
- Různé varianty funkce *zruš* znamenají různé strategie prohledávání grafu, např. do hloubky (*zruš*=LIFO), do šířky(*zruš*=FIFO), heuristické prohledávání (heuristická funkce nabývá na učící množině svého maxima [18]) apod. ;
- *zvol* určuje, která odvozovací pravidla aplikovat na H;
- *zruš některé prvky* určuje, které z hypotéz z QH budou zrušeny jako málo nadějné;
- *kriterium ukončení* určuje, zda již bylo nalezeno řešení a/nebo zda je QH prázdná;
- *vyber_ hypotézu* vybere z QH některé z možných řešení. Tato funkce, která nebývá v generickém algoritmu explicitně uváděna, bude diskutována v odstavci 6.1.2.

Některou z variant top-down ILP algoritmu, jemuž je věnována pozornost dále, získáme potom tak, že funkce *inicializuj* vrátí nejobecnější klauzuli (např. pro $reverse/2$ to bude $reverse(X, Y)$) a množina inferenčních pravidel se bude skládat z operací specializace (např. spojení dvou proměnných nebo přidání podcíle do těla klauzule).

2.3 Varianty ILP systémů

Jestliže implementujeme ILP systém, musíme uvážit, které z následujících variant zvolíme.

Interaktivita. ILP systém může být buď interaktivní nebo bez interakce s okolím. *Interaktivní* systém klade během učení otázky *orákulu* (člověku nebo jinému programu, např. databázovému systému). Takovým systémem je například MIS [19] (viz kap. 4), Clint [4] nebo FILP [1] .

Nepřesná data. Vstupní data mohou obsahovat různé typy nepřesné informace. O doménové znalosti B většinou předpokládáme, že neobsahuje chyby. V řadě aplikací nepracujeme s přesnými daty (což v ILP znamená, že příklad označený jako pozitivní je ve skutečnosti negativním a/nebo naopak). Aby byl systém schopen zpracovat taková data, musí být podmínky úplnosti a bezespornosti změkčeny. Výsledný program je pak přijatelný tehdy, když pokrývá téměř všechny pozitivní a téměř žádný negativní příklad (např. systém $FOIL$ [18]).

Nalezení chybějících predikátů. Doménová znalost B může být pro nalezení popisu příkladů nedostatečně bohatá, tj. nejsme pomocí ní schopni nalézt takovou formuli, která je správná a bezesporná. Metody *predicate invention* [20] slouží k automatickému doplnění B o nové predikáty.

Jednopredikátové učení vs. učení více predikátům. Některé systémy – např. MIS – dovolují, aby učící množina obsahovala příklady chování více než jednoho predikátu. Výsledkem je potom definice všech predikátů, jejichž příklady jsou v učící množině. Většina ILP systémů je však jednopredikátových.

Inkrementální vs. neinkrementální systémy. Inkrementální systémy dovedou modifikovat nalezené řešení, jakmile se objeví nový pozitivní nebo negativní příklad, který není ve shodě s dosud navrženým řešením. Neinkrementální systémy musí v takovém případě začít od začátku.

3 Top-down ILP

Popis jedné z dvou základních instancí obecného algoritmu ILP, *top-down algoritmu*, je obsahem této kapitoly [2]. V dalším textu předpokládáme, že

[2] O bottom-up strategii viz např. [12]).

v učící množině jsou příklady chování jen jednoho predikátu a že řešením jsou všechna pravidla obsažená v QH, tj. QH obsahuje definici právě tohoto predikátu. *Top-down algoritmus* začíná s nejobecnější logickou formulí, v našem případě je to např. *rodič(X,Y)* v příkladu na obr.1b anebo *reverse(X,Y)* (obr.1c). V dalším textu předpokládáme, že *Kriterium_ukončení* je splněno, pokud QH (chápáno jako definice predikátu) platí pro všechny pozitivní příklady a pro žádný negativní. Pokud tato formule platí i pro některý negativní příklad, musíme ji „zeslabit", a tak najít formuli méně obecnou.

3.1 Základní pojmy

Definice 1: *Řekneme, že F* **je specializací** *G právě když $G \models F$. Tedy libovolný model G je modelem F.*

Definice 2: Specializační operátor *přiřazuje každé klauzuli množinu jejích specializací.*

Popis základních specializačních operátorů a specializačního grafu je je obsažen v následujících kapitolách.

3.2 Obecné operátory specializace v logice 1. řádu

Většina ILP systémů používá dvě základní operace specializace:

- **ztotožnění 2 proměnných**
  např. $spec(reverse(X, Y)) = reverse(X, X)$;

- **přidání podcíle** do těla formule, přičemž tento cíl obsahuje dosud nepoužité proměné
  např. $spec(reverse(X, Y)) = reverse(X, Y) \leftarrow concat(U, V, W)$;

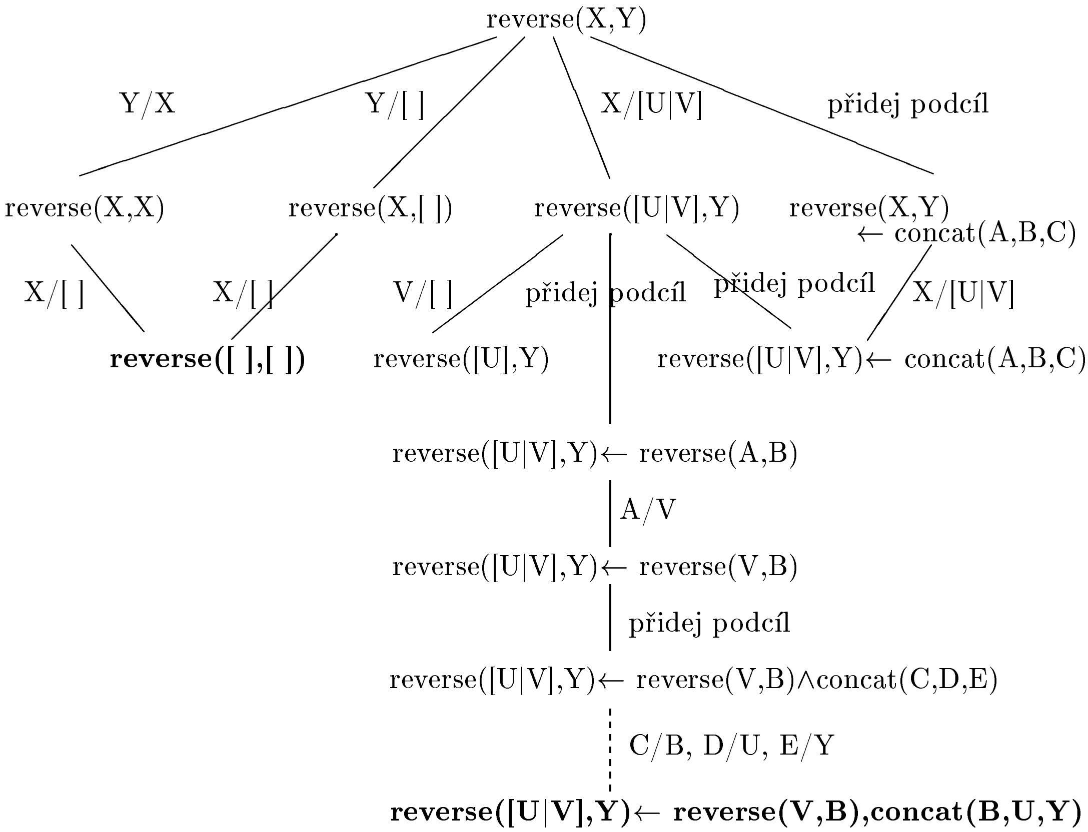
Obvykle pracujeme s programy, v nichž se mohou vyskytnout konstanty a/nebo složené termy. Potom použijeme dalších dvou operátorů

- **nahrazení proměnné konstantou**
  např. $spec(reverse(X, Y)) = reverse([\,], Y)$;

- **nahrazení proměnné složeným termem**
  např. $spec(reverse(X, Y)) = reverse([U|V], Y)$;

3.3 Specializační graf

Ukázka části specializačního grafu pro $reverse(X, Y)$ je na následujícím obrázku. Kořenem grafu je nejobecnější klauzule reverse(X,Y). Z uzlu C_1 vede hrana do uzlu C_2, jestliže klauzule C_2 vznikne z C_1 aplikací jednoho ze specializačních operátorů. Klauzule, které tvoří definici predikátu $reverse/2$, jsou vysázeny tučně. [3]



[3] Z důvodu úspory místa není specializační graf úplný. Např. na první úrovni chybí specializace $X/[]$ a přidání podcíle $reverse(A, B)$.

4 MIS

4.1 Popis

MIS - Model Inference System [19] je jeden z prvních ILP systémů. Pracuje metodou top-down, je interaktivní a příklady zpracovává po jednom, je tedy inkrementální. Vstupem je 1 pozitivní příklad a znalosti o procedurách, které se mohou vyskytnout ve výsledném programu. Znalost obsahuje informaci o tom, které parametry procedur jsou vstupní resp. výstupní(tzv. módy parametrů) a jakého jsou typu (seznam apod.) a dále které procedury tato procedura volá. Výsledkem je teorie, která je úplná a bezesporná vzhledem k dosud známým příkladům. Během učení může být uživatel dotazován na očekávaný výsledek některých podcílů.

MIS začíná s prázdným programem. V každém kroku může pro vytvořený program detekovat některou z následujících chyb:

1. Pro negativní příklad program uspěje - NEKONZISTENCE.
   Po nalezení chybné klauzule (tj. té, která způsobila, že je program nekonzistentní) systém tuto zruší a hledá další možnou (vzhledem k jistému uspořádání), která odpovídá zadaným příkladům chování učeného programu.

2. Pro pozitivní příklad program neuspěje - NEÚPLNOST
   MIS přidá na konec programu novou klauzuli, která pokrývá tento příklad a je nejobecnější možná, tj. nepokrývá žádný negativní příklad a je méně obecná než všechny dosud vyzkoušené(a zavržené) klauzule (viz kap. 4.5).

3. Program neskončí - DIVERGENCE
   Prologovský interpret má na vstupu cíl A a maximální hloubku výpočtu d. Vrací true a instanci A, pokud program uspěl bez překročení maximální hloubky. V opačném případě, když program neuspěje při stanovené hloubce výpočtu, se pokračuje stejně jako v případě nekonzistence.

4.2 Algoritmus

V následujícím algoritmu množina označených (rozuměj zavržených) klauzulí obsahuje ty testované klauzule, které pokrývaly alespoň jeden negativní příklad.

```
Vstup: B, E+, E−, omezující podmínky
    QH := {} ;
    Množina označených klauzulí je prázdná ;
    repeat
       read příklad;
       repeat
          if program QH neuspěje na pozitivním příkladu
             najdi novou klauzuli C, aby QH ∪ {C} pokrylo tento příklad;
             QH := QH ∪ {C};
          if program QH uspěje na negativním příkladu
             najdi chybnou klauzuli C programu QH;
             QH := QH − {C};
             označ klauzuli C ;
       until QH je úplná a bezesporná
       write QH
    until všechny příklady z E+, E− byly načteny
Výstup: posloupnost programů QH1, QH2, ... úplných a bezesporných
vzhledem k známým pozitivním a negativním příkladům
```

MIS: schéma algoritmu

Nejzajímavější je jistě způsob nalezení nových klauzulí, a proto si ho všimneme podrobněji.

4.3 Tvorba nových klauzulí

je založena na strategii top-down (začíná se od nejobecnější klauzule, která se postupně specializuje) a je inkrementální, tj. klauzule jednou odstraněná z programu nebude nikdy vrácena zpět. Je silně závislá na informaci o *módech argumentů*. Každý argument může být buď *vstupní* – musí mít v okamžku vyhodnocování cíle s tímto argumentem instalovanou hodnotu – nebo *výstupní* – bude instalován na konstantu po úspěšném vykonání cíle.

Kořenem stromu možných klauzulí - nejobecnějším možným kandidátem - je predikát se všemi argumenty jako jednoduchými proměnnými a bez podcílů, v našem příkladu `reverse(X,Y)`.

Pro jeho specializaci je možno použít jednak **instalace proměnných**, a to pouze u klauzulí bez podcílů. Proměnné se instalují buď na konstantu, funkci nových proměnných (v příkladu např. na `reverse([U|V],Y)` nebo sloučením dvou vstupních proměnných téhož typu. Výstupní proměnné mouhou být unifikovány pouze se vstupními proměnnými tzv. operací pro uzavření klauzule.

Druhým specializačním pravidlem je **přidání podcíle**. Je-li klauzule rozšířena o podcíl, všechny jeho vstupní argumenty jsou tvořeny již existujícími vstupními proměnnými. Nové proměnné - tj. výstupní proměnné tohoto podcíle - jsou přidány do seznamu vstupních proměnných (mohou být použity jako vstupní v dalších podcílech) a do seznamu volných proměnných. Je-li posléze volná proměnná použita jako vstupní, je ze seznamu volných proměnných odstraněna. Zbývající volné proměnné mohou být odstraněny jen aplikací následujícího operátoru pro uzavření klauzule.

Operátor pro **uzavření klauzule** můžeme aplikovat i na klauzule s podcíli. Pokud výstupní proměnná je unifikována s volnou proměnnou, potom tato volná proměnná je odstraněna ze seznamu volných. Ve skutečnosti je tento operátor aplikován jen tehdy, způsobí-li vyprázdnění seznamu volných proměnných.

Ukažme si celý proces opět na příkladu syntézy predikátu `reverse/2` deklarovaném jako `reverse(+[x],-[x])` (oba argumenty jsou seznamy, první je vstupní, druhý výstupní) s doménovou znalostí obsahující predikát `concat/3`. Předpokládejme, že bázová klauzule `reverse([],[])` už byla nalezena a že se učící množina skládá z následujících příkladů

```
reverse([1,2],[2,1]), true
reverse([1,2],[1]),   false
```

Postup syntézy rekurzivní klauzule potom vypadá takto:

```
reverse(X,Y).
vstupní proměnné  = <X, list(integer)>
výstupní proměnné = <Y, list(integer)>
                  |
                  |   X /[H|T]
                  V
reverse([H|T],Y).
vstupní proměnné  = <H, integer>, <T, list(integer)>
výstupní proměnné = <Y, list(integer)>
                  |
                  |   přidej_podcíl reverse(T,S)
                  V
reverse([H|T],Y) :- reverse(T,S).
vstupní proměnné  = <H, integer>, <T, list(integer)>,
                   <S, list(integer)>
výstupní proměnné = <Y, list(integer)>
volné proměnné   = <S, list(integer)>
                  |
                  |  přidej_podcíl concat(H,S,I)
                  V
reverse([H|T],Y) :- reverse(T,S), concat(H,S,I).
vstupní proměnné  = <H, integer>, <T, list(integer)>,
                   <S, list(integer)>, <I,list(integer)>
výstupní proměnné = <Y, list(integer)>
volné proměnné   = <I, list(integer)>
                  |
                  |  Y/I (uzavři klauzuli)
                  V
reverse([H|T],I) :- reverse(T,S), concat(H,S,I).
```

Protože bázová klauzule `reverse([],[])` byla již nalezena, klauzule v 3. kroku uspěje, i když nemáme k dispozici příklad `reverse([2],[2])`. Závěrečná klauzule je úplná a konzistentní a neobsahuje žádnou volnou proměnnou, je tedy uzavřená. ■

Pomocí konečné množiny příkladů nemůžeme jednoznačně specifikovat nekonečnou relaci, aniž bychom nepřijali další podmínky. Buď omezíme, syntakticky a/nebo sémanticky, prohledávaný prostor – tyto podmínky se označují jako **bias** – nebo můžeme předpokládat něco o **kvalitě příkladů**. Třetí možností je **interakce s okolím**, např. žádost o další příklad. Právě interaktivní strategie systému MIS jsou náplní další kapitoly.

4.4 Interaktivní strategie MIS

Hladová (eager search) strategie je nejsilnější z tří implementovaných strategií, ale též nejvíce zaměstnává orákulum. Hlavní výhodou této strategie je její nezávislost na pořadí příkladů. Např. pro `member(a,[b,a]),true` a pro vygenerovanou klauzuli

```
member(X,[Y|Z]) :- member(X,Z),
```

je orákulu položen dotaz `member(a,[a])`, tzv. dotaz na příslušnost, *membership query*. Je-li zodpovězen kladně, bude tato klauzule akceptována. Další dotaz `member(b,[a])` umožňuje zamítnout klauzuli

```
member(X,[Y|Z]) :- member(Y,Z),
```

je-li odpověď negativní. Při použití hladové strategie MIS najde model množiny příkladů v limitě [4] nezávisle na pořadí příkladů [19].

Líná (lazy search) strategie Protože dotazy se vyhodnocují na množině známých faktů (extensionální pokrytí), uživatel/orákulum není obtěžován dotazy. Strategie však závisí na pořadí příkladů. Např.klauzule `member(X,[Y|Z]) :- member(X,Z)` nebude nalezena, pokud množina příkladů neobsahuje instanci této klauzule, např. `member(a,[b,a]), member(a,[a])`. Pokud příklady přicházejí například v následujícím pořadí

```
member(a,[a]), member(b,[a,b]), member(c,[a,b,c]), ...,
```

tato klauzule nebude nikdy objevena. MIS najde model v limitě jen za předpokladu, že klauzule je h-snadná [5].

[4] *Konvergence v limitě* znamená, že po konečném počtu kroků, tj. příkladů, se už výsledek učení nezmění

[5] Klauzule je *h-snadná*, jestliže pro žádný cíl nebude hloubka výpočtu větší než h. Přesně: Nechť h je funkce z množiny cílů do množiny celých čísel $h : Goals \rightarrow Integer$ a nechť pro každý cíl $A \in Goals$ hloubka libovolného výpočtu na A není větší než $h(A)$. Řekneme, že program P je $h-snadný$, jestliže pro každý cíl A z Herbrandova universa $H(P)$ je hloubka libovolného výpočtu nejvýše $h(A)$. [19], str.92 .

Adaptivní (adaptive search) strategie Adaptivní strategie využívá k nalezení odpovědí na dotazy již vytvořených klauzulí (intenzionální pokrytí). Je méně závislá na pořadí příkladů než líná strategie. Jedinou podmínkou je to, aby nejprve byla nalezena jednodušší klauzule – např. pro `reverse/2` klauzule `reverse([ ],[ ])`. Podobně jako líná ani adaptivní strategie tedy neklade dotazy orákulu. Nechť např. množina příkladů obsahuje dva pozitivní příklady `member(a,[a])`, `member(d,[e,d])` a dva negativní `member(b,[c])`, `member(f,[g,h])` a nechť bázová klauzule `member(X,[X|Y])` již byla nalezena. Potom dotaz `member(b,[b])` nebude položen, protože je možno jej zodpovědět pomocí této bázové klauzule. Za stejného předpokladu jako u líné strategie (h-snadnost), adaptivní strategie identifikuje model v limitě.

4.5 Příklad

V tomto odstavci ukážeme hlavní kroky při syntéze programu $reverse(X,Y)$.

Za znakem '?' následuje vstup uživatele vytištěný kurzívou. True resp. false za každým faktem určuje, má-li cíl uspět nebo neuspět.

`Next fact?` $reverse([],[]), true.$
```
Checking fact(s)...
Error. missing solution reverse([ ],[ ]).diagnosing...
Error diagnosed: reverse([ ],[ ]) is uncovered.
```

`Declare reverse(X,Y)?` $declare(reverse(+[x],-[x]),[total, determinate]).$
`Procedures called by reverse(X,Y)?` $concat(\_,\_,\_), reverse(\_,\_).$

Systém vyžaduje informace o typu, vstup/výstupní módy argumentů a případně charakteristiku procedury. Oba dva argumenty jsou typu seznam (`[x]`), první je vstupní (+), druhý je výstupní (-). Procedura je charakterisovaná jako *total*, tj. pro každou hodnotu vstupního argumentu X existuje alespoň jedna hodnota výstupních argumentů, a *determinate*, tj. taková hodnota je maximálně jedna. Z procedur využívá jen programu `concat/3` a protože v seznamu volaných procedur je uvedeno `reverse/2`, může být rekurzívní.

```
Found clause: reverse(X,X):-true
   after searching 5 clauses.
Listing of reverse(X,Y):
```

```
    reverse(X,X):-true.
Checking fact(s)...no error found.
```

Algoritmus našel nejobecnější řešení, které pokrývá zadaný příklad. Je však korektní jen pro nejvýše jednoprvkové seznamy. Další 2 příklady vedou k nápravě.

Next fact? $reverse([a,b],[a,b]), false.$

```
Checking fact(s)...
Found clause: reverse(X,[ ]):-true
   after searching 4 clauses.
Listing of reverse(X,Y):
    reverse(X,[ ]):-true.
Checking fact(s)...no error found.
```

Next fact? $reverse([a],[]), false.$

```
Found clause: reverse([ ],[ ]):-true
   after searching 5 clauses.
Listing of reverse(X,Y):
    reverse([ ],[ ]):-true.
Checking fact(s)...no error found.
```

Tento program je málo obecný, musíme nabídnout takový příklad, který popisuje chování `reverse/2` pro delší seznamy.

Next fact? $reverse([a,b],[b,a]), true.$

MIS hledá ve specializačním grafu dosud netestovanou klauzuli, která by pokryla tento příklad. Tou je `reverse([X|Y],Z) :- reverse(Y,V)` [6]. Pokud není množina příkladů dostatečně bohatá, MIS klade uživateli otázky. Pro zjištění správnosti této klauzule potřebuje znát odpověď na následující dotaz.

`Query: reverse([b],[b,a])?` *no.*

Klauzule je tedy zamítnuta. Další je `reverse([X|Y],Z) :- concat(Y,X,W)`. Pro zjištění správnosti této klauzule potřebuje znát odpověď na následující dotaz.

[6] Byl použit operátor přidání podcíle, kde vstupní proměnná `reverse/2` je vybrána ze seznamu existujících vstupních proměnných $\{X, Y\}$. Protože musí být typu seznam, byla vybrána proměnná Y, viz kap. 4.3

`Query: concat([b],a,[b,a])` *yes.* [7]

```
Found clause: reverse([X|Y],Z):-concat(Y,X,Z)
after searching 21 clauses.
```

Všimněme si, že tento program je vzhledem k zadaným příkladům správný. Zadali jsme totiž jen příklady s nejvýše dvouprvkovými seznamy. Alespoň jeden příklad s alespoň tříprvkovým seznamem musíme tedy systému poskytnout, například

```
reverse([t,a,b,a,k],[k,a,b,a,t]),
```

aby našel správné řešení

```
reverse([ ],[ ] ):-true.
reverse([X|Y],Z):-reverse(Y,U),concat(U,X,Z).
```

4.6 Mechanizace orákula

Omezením počtu dotazů na výsledky podcílů se zabývají práce [10] a [13]. V druhé zmíněné práci je popsána mechanizace orákula. Uživatel zadává parciální specifikace programu anebo omezující podmínky (assertions). Systém se nejprve snaží získat odpověď s pomocí těchto specifikací. Pokud k zodpovězení dotazu nestačí, teprve tehdy je položen dotaz uživateli. Ten může odpovědět přímo (YES/NO a případně specifikace hodnot proměnných) nebo zadat další omezující podmínku. Typy omezujících podmínek mohou být dvojího druhu (`p(x)` je libovolný predikát):

Universální orákulum (pro všechny instalace proměnných)

`true(predicate(X))`	Odpověď YES pro všechna X
`false(predicate(X))`	Fail

Existenční orákulum (existuje taková instalace proměnných)

`posex(predicate(X))`	splnitelný cíl predicate(X) (ex. pozitivní příklad)
`negex(predicate(X))`	existuje negativní příklad

[7]Tento dotaz by nemusel být položen, kdyby *MIS* věděl, že definice predikátu `concat/3` je správná. Tuto znalost však nemá.

Např. pro predikát `member/2` mají smysl tyto aserce

```
true(member(X,[X]))      posex(member(X,[Y]))
false(member(X,[]))      negex(member(X,[Y])) .
```

5 Markus

5.1 Popis

Markus [7, 8, 9] obohacuje a zdokonaluje Shapirův Model Inference System. Je neinteraktivní a používá tzv. *pokrývací metodu* (covering paradigm): hledá rozdělení množiny pozitivních příkladů na navzájem dizjunktní části, z nichž každá je popsána jedinou klauzulí. Učící algoritmus najde první klauzuli, která pokrývá alespoň jeden pozitivní příklad a žádný negativní. Pokryté pozitivní příklady jsou poté z učící množiny zrušeny a algoritmus pokračuje v hledání pokrytí této menší množiny příkladů. Výpočet končí, jsou-li všechny pozitivní příklady pokryty a není pokryt žádný negativní.

Vlastní syntéza logického programu probíhá ve 3 krocích. Nejprve se ověří, že vygenerovaná klauzule je *slibná*, tj. pokrývá alespoň jeden z dosud nepokrytých pozitivních příkladů. Po uzavření klauzule (viz 4.3) se hotová klauzule testuje, zda je stále slibná a navíc *dobrá*, tj. nepokrývá žádný negativní příklad. Taková klauzule se potom přidá k ostatním a výsledná definice se testuje na *úplnost a správnost*.

Refinement v $MARKUSu$ vychází z MIS. Navíc však umožňuje přidat do těla klauzule negativní literál. Hlavní rysy $MARKUSu$ jsou následující [8].

- optimální generování specializačního grafu bez duplicitních uzlů (zdokonalení postupu z [10]);
- iterativní prohledávání do hloubky (iterative deepening search) specializačního grafu;
- omezení prohledávaného prostoru pomocí parametrů (např. *maximální délka klauzule, max. složitost funkčních termů* viz níže) ;
- pokrývací strategie;
- neinteraktivní učení.

Výpočetní složitost algoritmů ILP je jedním z velkých problémů. Proto si zaslouží pozornost způsob, jak ji lze snížit v systému $MARKUS$. Obecná charakteristika omezujících podmínek(bias) a jejich reprezentace v $MARKUSu$ jsou popsány v následující časti.

5.2 Omezující podmínky(bias)

se obvykle dělí na dvě hlavní skupiny, na jazykový bias – omezení jazyka – a na omezující podmínky pro prohledávání prostoru hypotéz.

Omezení jazyka Tyto podmínky omezují složitost a tvar logických formulí. Řada systémů dovoluje definovat maximální počet klauzulí a maximální počet podcílů v klauzuli. Existují však i jazyky, které dovolují popsat syntaktickou strukturu formulí do libovolných podrobností.

Velmi silným nástrojem jsou definice typů argumentů (slučovat se potom mohou jen proměnné stejných typů) a definice modů, jak jsme viděli v příkladu.

Omezení pro prohledávání prostoru hypotéz určují, jakým způsobem procházet graf řešení a kolik uzlů v grafu má smysl procházet. Pracujeme-li s nepřesnými daty, mohou také určovat míru nepřesnosti, kterou ještě akceptujeme.

Parametry $MARKUSUu$ můžeme rozdělit do tří skupin, kde první dvě odpovídají nahoře uvedené klasifikaci. Třetí skupina obsahuje především parametry pro vstup a výstup.

5.2.1 Omezení jazyka

- typ specializačního (refinement) operátoru [8];
- maximální počet klauzulí ve výsledku;
- maximální počet volných proměnných (nově zavedené proměnné, které ještě nebyly použity v těle klauzule) ;
- maximální počet literálů v těle klauzule;

[8]To potom implikuje hledání buď ve třídě obecných definitních logických programů nebo ve třídě definitních gramatik

- maximální hloubka funkčních termů.

5.2.2 Parametry pro prohledávání prostoru hypotéz

- maximální počet literálů přidaných v jednom kroku;
- maximální hloubka pro iterative deepening search;
- podmínka pro specializaci klauzule (slibná klauzule);
- maximální hloubka rekurze.

5.2.3 Ostatní

- úroveň zobrazované informace při procesu učení;
- ne/testovat vstupní data;
- jméno souboru pro uložení výsledku.

6 ILP: obecné problémy a nedostatky

ILP řeší především dva nedostatky dříve používaných metod induktivního učení: omezenost jejich vyjadřovacího jazyka(nejčastěji výrokový počet) a obtížné využívání doménové znalosti (další znalosti o řešeném problému). Obvykle za to však platíme velkou výpočetní složitostí systémů ILP. V následující části uvádíme další důležité problémy ILP, kterým se podle našeho mínění věnuje v literatuře ILP malá pozornost, a zabýváme se nedostatky současných top-down systémů.

6.1 Generický algoritmus: obecné problémy

6.1.1 Vstupní informace

Velké učící množiny Postupné zvětšování množiny příkladů se částečně řeší v kontextu interaktivních ILP systémů. Tváří v tvář velkým objemům dat však tyto metody selhávají. Výpočetní složitost ILP roste přinejmenším lineárně k počtu příkladů.

Redukce počtu příkladů Známe-li naučenou teorii, umíme téměř vždy pro daný ILP systém najít takovou podmnožinu učící množiny, která stačí pro naučení této teorie. Kdybychom dokázali tuto podmnožinu efektivně vybrat ještě před učením, jistě by se výpočetní složitost učení snížila.

Doménová znalost Zatímco množiny příkladů E^+, E^- se mohou během učení alespoň zvětšovat, doménová znalost, tj. seznam predikátů použitelných pro učení, se vždy předpokládá dán a neměnný. Přitom v netriviálních aplikacích (např. lingvistických [3]) často ani uživatel sám neví, které predikáty se budou ve výsledné teorii vyskytovat. Jen částečně se dá tento problém řešit postupným rozšiřováním omezujicích podmínek (shift of bias). Často je doménová znalost hierarchická. Částečně mohou pomoci jazyky pro popis bias [2].

6.1.2 Implementace funkcí generického algoritmu

Funkce zvol_odvozovací_pravidlo Vázání proměnných je většinou dobře popsáno. Magie systémů tkví v *přidání podcíle do těla klauzule*. Vlastní implmentace se buď značně liší od teoretického popisu (viz MIS) nebo je jen obtížně rozkódovatelná ($Markus$).

Funkce vyber_hypotézu Funkce *vyber_hypotézu*, která vybere jednu z naučených hypotéz, nebývá v generickém algoritmu explicitně uváděna. Proces učení obvykle končí s naučením první hypotézy, která splňuje kriterium ukončení.

6.2 Deduktivní odvozování

Často máme k dispozici, kromě doménové znalosti jako seznamu predikátů, i znalosti o požadovaných vlastnostech cílové teorie či omezující podmínky (viz např. aserce v kap. 4.6). Např. pro `reverse(X,Y)` víme, že argumenty musí být stejné délky a že všechny prvky, které se vyskytují v seznamu `X` se musí vyskytovat i v seznamu `Y`. Potom můžeme příklad `reverse([1,2],[0]), false` generovat automaticky na základě znalosti o stejné délce obou seznamů

$$\forall X, Y \exists N \ : \ reverse(X, Y) \wedge length(X, N) \rightarrow length(Y, N)$$

a příklad `reverse([1],[0]), false` můžeme odvodit ze znalosti

$$\forall X, Y, Z \ : \ reverse(X, Y) \wedge member(Z, X) \rightarrow member(Z, Y)$$

Z těchto znalostí též mj. plyne, že nemá například smysl testovat hypotézu `reverse([X|Y],Y)`, což ve svém důsledku vede k redukci specializačního grafu a tedy snižuje výpočetní složitost učení. Je nutno zdřaznit, že současné ILP systémy žádných takových deduktivních mechanismů nevyužívají.

6.3 Nedostatky top-down ILP systémů

Uvažujeme-li (interaktivní) top-down systémy v kontextu automatického logického programování [5], objevíme především tyto čtyři hlavní nedostatky:

1. Neexistuje strategie výběru učících příkladů;
2. Potřebujeme příliš mnoho (jak pozitivních tak negativních) příkladů;
3. Užitečnost negativních příkladů velmi závisí na konkrétní učící strategii;
4. Při větším počtu příkladů se top-down strategie stává neefektivní;
5. V interaktivních systémech musí uživatel odpovědět příliš mnoho dotazů.

Hlavním problémem systému MIS je výběr správných příkladů a (viz strategie) volba jejich vhodného pořadí. Uživatel musí více méně znát výsledný program a rozumět dobře zvolené strategii interakce.

$Markus$ se umí naučit běžné predikáty pro práci se seznamy stejně jako predikáty Peanovy aritmetiky z max. 4 dobře vybraných příkladů, je-li nastavení parametrů dobré. Právě tento výběr příkladů a též vhodné nastavení bias je hlavním problémem tohoto systému.

7 Dosažené výsledky a směry další práce

Předběžné vlastní výsledky jsou obsaženy v následujících článcích.

MIS se umí naučit jen programy v čistém Prologu. Některá rozšíření jsou popsána v [15]. Otázky týkající se induktivní syntézy logických programů jsou diskutovány v [5]. První výsledky získané modifikací systému $Markus$

jsou popsány v [6, 16, 17].

Vzhledem k dosaženým výsledkům a nahoře uvedeným problémům se možné směry další práce dají shrnout takto:

- Navrhnout a implementovat učící algoritmus vycházející ze systému *Markus*, který bude vyžadovat menší počet příkladů a bude méně závislý na kvalitě učící množiny;
- Demonstrovat použití navržených metod v různých aplikačních oblastech;
- Vytvořit pro některou oblast znovupoužitelnou doménovou znalost;
- Uvážit induktivní učení v některé jiné logice.

Odkazy

[1] Bergadano F. and Gunetti D.: An interactive system to learn functional logic programs. *Proc. of IJCAI'93*, Chambéry, pp. 1044–1049.

[2] F. Bergadano: Towards an Inductive Logic Programming Language (manuscript)

[3] Cussens J.: Part-of-Speech Tagging using Progol. In Proc. of ILP'97, LNAI 1297, Springer-Verlag 1997.

[4] De Raedt, L.: Interactive Concept-Learning. PhD Thesis, Catholic University Leuven, Belgium 1991.

[5] Flener P., Popelínský L.: On the use of inductive reasoning in program synthesis: Prejudice and prospects. Proc. of the 4th Int'l Workshop on Logic Program Synthesis and Transformation (LOPSTR'94), Pisa, Italy, 1994.

[6] Flener P., Popelínský L. Štěpánková O.: ILP nad Automatic Programming: Towards three approaches. Proc. of 4th Workshop on Inductive Logic Programming (ILP'94), Bad Honeff, Germany, 1994.

[7] Grobelnik M.: MARKUS: An optimized Model Inference System In Proceedings of the ECAI-92 Workshop on Logical Approaches to Machine Learning, Vienna 1992.

[8] Grobelnik M.: Induction of Prolog programs with Markus. In Deville Y.(ed.) Proceedings of LOPSTR'93. Workshops in Computing Series, pages 57-63,Springer-Verlag, 1994.

[9] Grobelnik M.: Declarative Bias in Markus ILP system. *Working notes of the ECML'94 Workshop on Declarative Bias*, Catania, 1994. (chairperson Rouveirol C.)

[10] Huntbach, M.: An improved version of Shapiro's Model Inference System. In Shapiro E.(Ed.), Proceedings of Third International Conference On Logic Programming ICLP'86, London, pp.180-187, LNCS 225, Springer-Verlag 1986.

[11] Muggleton S. (ed): Inductive Logic Programming. Volume APIC-38, Academic Press, 1992.

[12] Muggleton S., De Raedt L.: Inductive Logic Programming: Theory And Methods. J. Logic Programming 1994:19,20:629-679.

[13] Nadjm-Tehrani,S: Contributions to the Declarative Approach to Debugging PrologPrograms. Linköping Studies in Science and Technology, Thesis No. 187, Dept. of Computer and Information Science, Linköping University

[14] Nienhuys-Cheng S.-H., de Wolf R.: Foundations of Inductive Logic Programming. Lect. Notes in AI 1228, Springer Verlag Berlin Heidelberg 1997.

[15] Popelínský L.: Towards Synthesis of Nearly Pure Logic Programs. In: Proceedings of LOPSTR'91, Workshops Series Springer Verlag 1992, ISBN 3-540-19742-7

[16] Popelínský L.: Towards Program Synthesis From A Small Example Set. Proceedings of 21st Czech-Slovak conference on Computer Science SOFSEM'94, pp.91-96 Czech Society for Comp. Sci. Brno 1993. (See also Proceedings of 10th WLP'94, Zuerich 1994, Switzerland.)

[17] Popelínský L., Štěpánková O.: WiM: A Study on the Top-Down ILP Program . Technical report FIMU-RS-95-03, August 1995.

[18] Quinlan J.R.: FOIL: A midterm report. Proceedings of ECML'93, LNAI 667, Springer-Verlag 1993.

[19] Shapiro Y.: Algorithmic Program Debugging. MIT Press, 1983.

[20] Stahl I.: Predicate Invention in Inductive Logic Programming. In: L. De Raedt(Ed.), Advances of Inductive Logic Programming, IOS Press, 1996.

[21] Štěpánková O., Csontó J.: Programovací prostředky pro UI. In: Mařík V., Štěpánková O., Lažanský J. a kol.: Umělá inteligence(2). Academia Praha 1997

[22] Wrobel S.: On the proper definition of minimality in specialization and theory revision. In Brazdil P.B.(Ed.): Proceedings of ECML-93 Conference. Lecture Notes in Artificial Intelligence 667, pp. 65-82., Springer-Verlag 1993.