Metody programowania 2016

Lista zadań na pracownię nr 2

Termin zgłaszania w KNO: 4 kwietnia 2016, godzina 6:00 AM CET

Na zajęciach z *Logiki dla informatyków* rozważaliśmy rezolucję dla rachunku zdań. Przyjęliśmy wówczas następujące definicje (por. str. 36–37 *Materiałów do zajęć*):

zmienna zdaniowa: element pewnego, przeważnie przeliczalnego nieskończonego zbioru V zmiennych zdaniowych;

literał pozytywny: zmienna zdaniowa $p \in V$;

literał negatywny: zanegowana zmienna zdaniowa $\neg p$, gdzie $p \in V$;

literał: literał pozytywny lub negatywny;

klauzula: skończony zbiór literałów.

Zamiast zapisywać klauzule za pomocą notacji dla zbiorów, umówiliśmy się, że będziemy je zapisywać w postaci formuł zdaniowych zbudowanych z literałów za pomocą symbolu alternatywy „$\vee$”. Notacja ta jest niejednoznaczna — umówiliśmy się, że np. każdy z napisów $p \vee q$, $q \vee p$, $q \vee p \vee q$ itd. oznacza tę samą klauzulę $\{p, q\}$.

Aby zapisywać klauzule w Prologu, przyjmiemy notację bardzo zbliżoną do opisanej powyżej. Nadamy priorytety i kierunki łączności funktorom `~/1` i `v/2`:

```
:- op(200, fx, ~).
:- op(500, xfy, v).
```

Zmienne zdaniowe będziemy reprezentować za pomocą atomów różnych od `[]/0`. W naszej notacji klauzule będą więc zapisywane następująco:

zmienna zdaniowa: dowolny atom inny niż `[]/0`;

literał pozytywny: zmienna zdaniowa;

literał negatywny: struktura $\sim p$, gdzie p jest zmienną zdaniową (ponieważ funktor `~/1` jest operatorem prefiksowym, to nie musimy zmiennej otaczać nawiasami);

literał: literał pozytywny lub negatywny;

klauzula pusta: atom `[]/0`;

klauzula niepusta: pojedynczy literał l lub struktura l `v` k, gdzie l jest literałem, a k — klauzulą niepustą;

klauzula: klauzula niepusta lub klauzula pusta.

Oto przykłady struktur, które są klauzulami w naszej notacji:

```
p v q v r
~r v ~q v ~p
~q v r
p
~s
[]
```

Zauważmy, że definicja klauzul niepustych bardzo przypomina definicję list (tutaj dodatkowo niepustych i których elementy są literałami). Podobnie jak nie wszystkie struktury zbudowane za pomocą funktora `./2` są listami, tak tutaj nie wszystkie struktury zbudowane za pomocą funktora `v/2` są klauzulami. Nie są nimi np. struktury:

```
(p v q) v r
~(~s)
p v []
[] v []
```

Zauważmy, że `~ ~ s` nawet nie jest strukturą — to wyrażenie jest niepoprawne składniowo.

Wiemy z kursu logiki, że dowolną formułę można przedstawić w postaci zbioru klauzul (interpretowanego logicznie jako ich koniunkcja). W Prologu zbiory klauzul będziemy zapisywać w postaci list, np.

```
[ ~p v q, ~p v ~r v s, ~q v r, p, ~s ]
[ p v r, ~r v ~s, q v s, q v r, ~p v ~q, s v p ]
[ p v q v r, ~r v ~q v ~p, ~q v r, ~r v p ]
```

Za pomocą reguły rezolucji

$$\frac{C \cup \{p\} \quad D \cup \{\neg p\}}{C \cup D}$$

możemy znajdować *rezolwenty* dwóch klauzul. Wiemy, że fakt, iż wśród rezolwent znajduje się klauzula pusta jest równoważny temu, że podany zbiór klauzul jest sprzeczny.

Zadanie polega na zaprogramowaniu predykatu

```
prove(+Clauses, -Proof)
```

który dla podanej niepustej listy niepustych klauzul `Clauses` w przedstawionym wyżej formacie utworzy listę `Proof` zawierającą pary postaci `(Clause,Origin)`, gdzie `Clause` jest klauzulą, zaś `Origin` określa, jak powstała dana klauzula: jeśli klauzula jest elementem wejściowego zbioru klauzul, to jest atomem `axiom`, w przeciwnym razie jest parą liczb naturalnych określających położenie na liście `Proof` klauzul których dana klauzula jest rezolwentą (klauzule na liście numerujemy od jedynki). Oczywiście wolno korzystać tylko z wcześniej wyprowadzonych klauzul, więc te dwie liczby powinny być mniejsze niż indeks bieżącej klauzuli. Dla przykładu predykat `prove/2` może wygenerować następujące rozwiązanie:

```
?- prove([~p v q, ~p v ~r v s, ~q v r, p, ~s], Proof).
Proof = [ (~p v q, axiom), (~p v ~r v s, axiom), (~q v r, axiom), (p, axiom), (~s, axiom), (q, (1,4)),
          (r, (3,6)), (~p v ~r, (2,5)), (~p, (7,8)), ([], (4,9)) ].
```

Ostatnią klauzulą na liście powinna być klauzula pusta wieńcząca dowód. Szczegóły sposobu wypisywania listy zależą od ustawień interpretera. Mimo iż dla czytelności odpowiedź Prologu została nieco „poprawiona”, to i tak dowód w tej postaci nie jest zbyt czytelny. Chcielibyśmy też wczytywać zbiory klauzul z pliku, a nie wpisywać je w każdym poleceniu. Dlatego do zadania dołączyliśmy definicje kliku predykatów, w tym `main/1`, które zajmują się wczytywaniem danych i wypisywaniem wyników w czytelnej postaci. Predykat `prove/2`, który należy napisać, jest wywoływany przez predykat `main/1`. Argumentem predykatu `main/1` jest atom będący nazwą pliku z zadaniem. Przykład uruchomienia:

```
?- main('zad125.txt').
```

Plik z zadaniem powinien być plikiem tekstowym, na którego początku znajduje się lista niepustych klauzul w opisanej wyżej notacji. Za nawiasem „`]`” kończącym listę należy postawić kropkę i dowolny biały znak (np. znak nowego wiersza). Wszelkie inne znaki umieszczone w pliku są pomijane przez program (można tam umieścić np. komentarz do zadania). Pliki zawierające zadania 123–125 ze skryptu z *Logiki* zostały dołączone do naszego zadania. Na stronie zajęć mogą zostać ogłoszone dalsze przykłady. Np. plik `zad123.txt` zawiera następujący tekst:

```
[ ~p v q, ~p v ~r v s, ~q v r, p, ~s ].
WCH: Logika dla Informatyków 2015, str. 37, zadanie 123.
```

Po dołączeniu do pliku `theorem_prover.pl` własnej implementacji predykatu `prove/2` można ją testować jak poniżej:

```
?- [theorem_prover].
% theorem_prover compiled 0.00 sec, 17 clauses
true.

?- main('zad123.txt').

 1.  ~p v q       (axiom)
 2.  ~p v ~r v s  (axiom)
 3.  ~q v r       (axiom)
 4.  p            (axiom)
 5.  ~s           (axiom)
 6.  q            (1,4)
 7.  r            (3,6)
 8.  ~p v ~r      (2,5)
 9.  ~p           (7,8)
10.  []           (4,9)

true ;
```

Predykat `prove/2` może nawracać wiele razy, wyszukując różne dowody.

Zbiór klauzul zawierający n literałów posiada co najwyżej 2^n różnych rezolwent (każda rezolwenta jest podzbiorem zbioru literałów występujących w oryginalnych klauzulach). Przestrzeń przeszukiwania, choć wykładnicza, jest więc skończona. Oznacza to, że przynajmniej teoretycznie, predykat `prove/2` mógłby znaleźć wszystkie możliwe dowody a następnie zawieść. W szczególności dla niesprzecznego zbioru klauzul powinien po prostu zawieść:

```
?- main('zad125.txt').
false.

?-
```

gdzie plik `zad125.txt` zawiera tekst:

```
[ p v q v r, ~r v ~q v ~p, ~q v r, ~r v p ].
WCH: Logika dla Informatyków 2015, str. 37, zadanie 125.
```

Ze względu na wykładniczy względem rozmiaru problemu czasu działania, dla większych zbiorów klauzul program może jednak działać bardzo długo.

Zakładamy, że pliki wejściowe są poprawne. Zachowanie predykatu `prove/2` w razie wywołania z niepoprawnymi argumentami może być dowolne. Nie gwarantujemy sensownego wypisywania wyników, jeśli predykat `prove/2` dostarcza wyniki w niepoprawnym formacie. Nie precyzujemy sposobu nawracania i generowania kolejnych dowodów. W szczególności nie precyzujemy warunków, kiedy dwa dowody uważamy za równoważne. Te szczegóły wynikają z przyjętej metody poszukiwania dowodu, a tej nie chcemy narzucać. Szczegóły należy skonsultować z prowadzącym.

Poniżej znajdują się definicje predykatów dołączonych do zadania.

```
/* Funktory do budowania klauzul */

:- op(200, fx, ~).
:- op(500, xfy, v).

/* Główny program: main/1. Argumentem jest atom będący nazwą pliku
 * z zadaniem. Przykład uruchomienia:
 *    ?- main('zad125.txt').
 * Plik z zadaniem powinien być plikiem tekstowym, na którego
 * początku znajduje się lista klauzul zbudowanych za pomocą funktorów
 * v/2 i ~/1 (szczegóły znajdują się w opisie zadania). Listę zapisujemy
 * w notacji prologowej, tj. rozdzielając elementy przecinkami
 * i otaczając listę nawiasami [ i ]. Za nawiasem ] należy postawić
 * kropkę. Wszelkie inne znaki umieszczone w pliku są pomijane przez
 * program (można tam umieścić np. komentarz do zadania).
 */

main(FileName) :-
   readClauses(FileName, Clauses),
   prove(Clauses, Proof),
   writeProof(Proof).

/* Silnik programu: predykat prove/2  do napisania w ramach zadania.
 * Predykat umieszczony poniżej nie rozwiązuje zadania. Najpierw
 * wypisuje klauzule wczytane z pliku, a po nawrocie przykładowy dowód
 * jednego z zadań. Dziewięć wierszy następujących po tym komentarzu
 * należy zastąpić własnym rozwiązaniem. */

prove(Clauses, Proof) :-
   maplist(addOrigin, Clauses, Proof).
prove(_, [(~p v q,axiom), (~p v ~r v s, axiom), (~q v r, axiom),
          (p, axiom), (~s, axiom), (q, (1,4)), (r, (3,6)),
          (~p v ~r, (2,5)), (~p, (7,8)), ([], (4,9))]).

addOrigin(Clause, (Clause, axiom)).

/* Pozostała część pliku zawiera definicje predykatów wczytujących listę
 * klauzul i wypisujących rozwiązanie. Wykorzystane predykaty
 * biblioteczne SWI-Prologu (wersja kompilatora: 6.6.6):
```

```
 *
 *    close/1
 *    format/2
 *    length/2
 *    maplist/3
 *    max_list/2
 *    nl/0
 *    open/3
 *    read/2
 *    write_length/3
 *
 * Dokumentację tych predykatów można uzyskać wpisując powyższe napisy
 * na końcu następującego URL-a w przeglądarce WWW:
 *    http://www.swi-prolog.org/pldoc/doc_for?object=
 * np.
 *    http://www.swi-prolog.org/pldoc/doc_for?object=write_length/3
 * lub jako argument predykatu help/1 w konsoli interpretera SWI
 * Prologu, np.
 *    ?- help(write_length/3).
 */

readClauses(FileName, Clauses) :-
   open(FileName, read, Fd),
   read(Fd, Clauses),
   close(Fd).

/* Wypisywanie dowodu */

writeProof(Proof) :-
   maplist(clause_width, Proof, Sizes),
   max_list(Sizes, ClauseWidth),
   length(Proof, MaxNum),
   write_length(MaxNum, NumWidth, []),
   nl,
   writeClauses(Proof, 1, NumWidth, ClauseWidth),
   nl.

clause_width((Clause, _), Size) :-
   write_length(Clause, Size, []).

writeClauses([], _, _, _).
writeClauses([(Clause,Origin) | Clauses], Num, NumWidth, ClauseWidth) :-
   format('~t~d~*|.  ~|~w~t~*+  (~w)~n',
          [Num, NumWidth, Clause, ClauseWidth, Origin]),
   Num1 is Num + 1,
   writeClauses(Clauses, Num1, NumWidth, ClauseWidth).

/* twi 2016/03/13 vim: set filetype=prolog fileencoding=utf-8 : */
```

Ninejszy plik wraz z programem prologowym i przykładowym danymi jest dostępny w serwisie KNO na stronie zajęć.