Kurs języka Prolog 2019

Lista zadań nr 2

Na zajęcia 13 marca 2019

Zadanie 1 (2 pkt). Rozważmy binarne rozwinięcia liczb naturalnych. Rozwinięciem liczby 0 jest ciąg złożony z pojedynczej cyfry `0`, a rozwinięcia liczb dodatnich nie zawierają zer nieznaczących (zatem ich najbardziej znaczącą cyfrą zawsze jest `1`). Napisz w Prologu predykaty `bin/1` oraz `rbin/1` generujące binarne rozwinięcia kolejnych liczb naturalnych reprezentowane w postaci list literałów `0` i `1`. Predykat `bin` powinien tworzyć listy cyfr dwójkowych w kolejności od najbardziej do najmniej znaczącej pozycji, `rbin` zaś — odwrotnie. Przeciętna liczba wnioskowań pomiędzy kolejnymi sukcesami powinna być ograniczona przez stałą, nie wolno więc kopiować ani odwracać generowanych list. Nie warto też używać żadnych predykatów standardowych. Pomimo iż sformułowanie zadania odwołuje się do pojęcia liczby, nie należy używać prologowej arytmetyki. Wynikiem zapytania `bin(X)` powinien być ciąg odpowiedzi:

```
X = [0];
X = [1];
X = [1,0];
X = [1,1];
X = [1,0,0];
X = [1,0,1];
...
```

zaś wynikiem zapytania `rbin(X)` powinien być ciąg odpowiedzi:

```
X = [0];
X = [1];
X = [0,1];
X = [1,1];
X = [0,0,1];
X = [1,0,1];
...
```

Zadanie 2 (2 pkt). Niech będzie dana n-elementowa lista $L = [x_1, x_2, \ldots, x_n]$. *Prefiksem* listy L jest lista pusta oraz każda z list $[x_1, x_2, \ldots, x_j]$, gdzie $1 \leq j \leq n$. *Sufiksem* listy L jest lista pusta oraz każda z list $[x_i, x_{i+1} \ldots, x_n]$, gdzie $1 \leq i \leq n$. Lista n-elementowa posiada $n+1$ prefiksów i sufiksów. *Odcinkiem* listy L jest lista pusta oraz każda z list $[x_i, x_{i+1}, \ldots, x_j]$ gdzie $1 \leq i \leq j \leq n$. Lista n-elementowa posiada $\frac{n(n+1)}{2} + 1$ odcinków. *Podlistą* listy L jest lista $[x_{i_1}, \ldots, x_{i_k}]$, gdzie $1 \leq i_1 < \ldots < i_k \leq n$. Podlista powstaje zatem przez usunięcie z listy niektórych elementów. Lista n-elementowa ma 2^n podlist. Zaprogramuj w Prologu predykaty `prefix/2`, `sufix/2`, `segment/2` i `sublist/2` spełnione wówczas, gdy ich pierwszy argument jest, odpowiednio, prefiksem, sufiksem, odcinkiem i podlistą listy będącej drugim argumentem.

Zadanie 3 (2 pkt). *Permutacją* listy $[x_1, \ldots, x_n]$ nazywamy dowolną listę postaci $[x_{i_1}, \ldots, x_{i_n}]$, gdzie $i_1, \ldots, i_n$ są parami różnymi liczbami z przedziału $1, \ldots, n$. Permutacja powstaje zatem przez przestawienie kolejności elementów na liście. Lista n-elementowa ma $n!$ permutacji. Permutacje można zdefiniować formalnie na wiele sposobów. Poniżej podajemy dwie definicje, oparte na indukcji strukturalnej względem list.

Definicja *przez wybieranie*:

- Jedyną permutacją listy pustej jest lista pusta.
- Głową permutacji niepustej listy jest dowolnie wybrany element tej listy, ogonem zaś dowolna permutacja listy powstałej przez usunięcie wybranego elementu z oryginalnej listy.

Definicja *przez wstawianie*:

- Jedyną permutacją listy pustej jest lista pusta.
- Permutacją listy niepustej jest lista powstała przez wstawienie głowy danej listy w dowolne miejsce listy będącej dowolną permutacją ogona danej listy.

Zaprogramuj predykaty `iperm/2` i `sperm/2` oparte na powyższych definicjach, spełnione wówczas, gdy ich drugi argument jest permutacją pierwszego. *Uwaga:* relacja „jest permutacją" jest symetryczna, więc wolelibyśmy powiedzieć „gdy ich argumenty są wzajemnie swoimi permutacjami". To, czemu wybraliśmy inne sformułowanie i dlaczego predykaty `iperm/2` i `sperm/2` nie są równoważne wyjaśnia zadanie 7.

Zadanie 4 (2 pkt). Wszystkie termy w Prologu są uporządkowane relacją liniowego porządku `@=</2`. Dla termów stałych jest to przyzwoicie zdefiniowana relacja porządku leksykograficznego. Co się dzieje dla nieukonkretnionych zmiennych, tego lepiej nie wiedzieć. Zaprogramuj predykat `monotone/1` spełniony wówczas, gdy jego argument jest niemalejącą listą termów. Możesz mieć problem z niepotrzebnie pozostawianym punktem nawrotu. Na razie zignoruj go. W przyszłym tygodniu nauczymy się, jak go *odciąć*.

Lista L_2 jest *uporządkowaniem* listy L_1, jeśli jest jej niemalejącą permutacją. Zgodnie z typowym dla Prologu podejściem *wygeneruj i sprawdź* możemy zdefiniować predykat

```
sort(X,Y) :-
   perm(X,Y),
   monotone(Y).
```

gdzie `perm/2` jest dowolnie zdefiniowanym predykatem implementującym relację „jest permutacją". Ta elegancka specyfikacja predykatu implementującego relację „jest uporządkowaniem" ma jedną wadę: działa w czasie wykładniczym względem długości listy. Trudno ją więc nazwać specyfikacją *wykonywalną*.

Ważną metodą optymalizacji programów prologowych jest *obcinanie gałęzi* (*pruning*): jeśli w pewnym punkcie nawrotu wszystkie liście poddrzewa przeszukiwania są niepowodzeniami i można to efektywnie stwierdzić *a priori* (bez przeszukiwania tego drzewa), to można całe to poddrzewo pominąć podczas obliczeń nie zmieniając semantyki programu. Pruning realizuje się zwykle przesuwając cel wywołujący niepowodzenie w kierunku korzenia drzewa przeszukiwania. W przypadku predykatu `sort/2` niepowodzenie spowoduje wywołanie predykatu `@=</2` w trakcie obliczenia predykatu `monotone/1`, jeśli permutacja wygenerowana przez predykat `perm/2` okaże się niemonotoniczna. Jeśli predykat `@=</2` będziemy wywoływać już na etapie tworzenia nowej permutacji, to oszczędzimy sobie wielu niepotrzebnych niepowodzeń. Jeśli predykat `sort/2` będziemy wywoływać w trybie `(+,?)`, to taka transformacja nie zmieni jego semantyki. Przerób w ten sposób predykaty `iperm/2` i `sperm/2` z zadania 3, tj. zaprogramuj predykaty `isort/2` i `ssort/2`. Zauważ, że otrzymałeś znane algorytmy sortowania przez wstawianie i wybieranie (gdybyśmy zaprogramowali wcześniej algorytm tworzenia permutacji przez składanie transpozycji, to tutaj moglibyśmy otrzymać z niego znany algorytm sortowania bąbelkowego).

Zadanie 5 (8 pkt). W zadaniu 4 zaprogramowaliśmy predykaty sortujące działające w czasie rzędu $n \cdot n$, gdzie n jest długością sortowanej listy. Wykonują one bowiem n iteracji wybierania/wstawiania, z których każda trwa $\Theta(n)$ kroków. Aby otrzymać algorytm optymalny (działający w czasie $O(n \log n)$), jedno z wystąpień zmiennej n w powyższym iloczynie trzeba zastąpić przez $\log n$: możemy wykonać $\log n$ iteracji, z których każda kosztuje $O(n)$, bądź wykonać n iteracji, każdą w czasie $O(\log n)$. W pierwszym przypadku, aby zakończyć pracę po $\log n$ iteracjach musimy wybierać/wstawiać nie pojedyncze elementy (jest ich n), tylko całe ich grupy. Jeśli ulepszymy w ten sposób algorytm sortowania przez

wstawianie, to otrzymamy algorytm *Mergesort* (najszybszy znany algorytm sortowania list). Ulepszając sortowanie przez wybieranie otrzymamy algorytm *Quicksort* (najszybszy znany algorytm sortowania tablic). W drugim przypadku, jeśli wstawianie/wybieranie pojedynczego elementu do/z struktury przechowującej uporządkowane elementy ma się odbywać w czasie logarytmicznym, to musimy użyć innej struktury danych niż lista. Wstawianie elementów w czasie logarytmicznym umożliwiają binarne drzewa poszukiwań (pesymistycznie tylko zbalansowane), wybieranie zaś — kopce. Z algorytmu sortowania przez wstawianie otrzymamy więc algorytm *Treesort*, a z algorytmu sortowania przez wybieranie — *Heapsort*. Zaprogramuj te cztery algorytmy sortowania.

Zadanie 6 (2 pkt). Najprostsza implementacja predykatu `reverse/2`:

```
reverse(X,Y) :-
   reverse(X,[],Y).

reverse([],A,A).
reverse([H|T],A,Y) :-
   reverse(T,[H|A],Y).
```

zapętla się w trybie `(-,+)`, tj. gdy pierwszy argument jest nieukonkretniony, a drugi ukonkretniony. Relacja „jest odwróceniem” jest symetryczna, oczekiwalibyśmy więc, że obliczenie celów `reverse`(a, b) oraz `reverse`(b, a) powinno dawać dokładnie ten sam efekt. Zaprogramuj predykat `reverse/2` w taki sposób, by nie zapętlał się dla żadnych danych.

Zadanie 7 (2 pkt). Relacja „jest permutacją”, podobnie jak „jest odwróceniem”, jest symetryczna. Niestety w trybie `(-,+)` implementacje predykatów `iperm/2` i `sperm/2` rozważane w zadaniu 3 cierpią na ten sam defekt, co naiwna implementacja predykatu `reverse/2`. Popraw je tak, by ich wywołania w dowolnym trybie zawsze dawały właściwy efekt.