PROGRAMACIÓN DECLARATIVA

LABORATORIO 5 (Parámetros de acumulación y eficiencia)

Para medir lo aprendido durante la Práctica 5, los alumnos deberán enviar los programas y soluciones a las preguntas que a continuación se indican. El envío se realizará en un archivo ZIP de nombre:

gN_Apellidos_Lab5.ZIP

donde N es el número del equipo de trabajo. El archivo ZIP contendrá los ficheros:
Apellidos_Lab5.pl con el código Prolog de todos los programas solicitados.
Apellidos_Lab5.txt con los datos personales de la persona que hace el envío y el enunciado y solución de aquéllas preguntas que (no pudiéndose responder vía programa) se formulen en cada uno de los ejercicios del laboratorio.

En todos los casos, "**Apellidos**" son los apellidos del alumno que hace el envío.

ENUNCIADO

Ejercicio 17.

Sean P1 y P2 dos programas que implementan una misma especificación, es decir, pueden considerarse semánticamente equivalentes. Si el programa P1 se ejecuta en un tiempo T1 y el programa P2 en un tiempo T2, el speedup S de P1 con respecto a P2 se define como: S=T2/T1.

Implementar un predicado speedup(S, G1, G2) que calcule el speedup S de un programa Prolog P1, en el que se lanza el objetivo G1, con respecto al programa P2, en el que se lanza el objetivo G2.

(Ayuda: utilizar el operador cputime, que se evalúa al número de segundos utilizados por Prolog hasta el momento de su ejecución. Así pues, un objetivo como el siguiente:

?- T1 is cputime, Goal, T2 is cputime, T is T2 – T1.

evalúa el tiempo T consumido durante la ejecución del objetivo Goal.)

Ejercicio 18.

La relación "invertir una lista" se puede definir de forma directa en términos del predicado append:

```
% invertir(L,I), I es la lista que resulta de invertir L
invertir1([],[]).
invertir1([H|T],L) :- invertir1(T,Z), append(Z,[H],L).
```

El problema con esta versión es que es muy ineficiente debido a que el número de llamadas al operador "[.|..]" es cuadrático con respecto al número de elementos de la lista que se está invirtiendo, si también contamos las llamadas al operador "[.|..]" producidas por una llamada al predicado append. Se puede lograr que el número de llamadas al operador "[.|..]" sea lineal con respecto al número de elementos de la lista utilizando un parámetro de acumulación. Un parámetro de acumulación es un argumento de un predicado que, como indica su nombre, se utiliza para almacenar resultados intermedios. En el caso que nos ocupa, se almacena la lista que acabará por ser la lista invertida, en sus diferentes fases de construcción.

```
% invertir2(L,I), I es la lista que resulta de invertir L
% Usando un parametro de acumulacion.
invertir2(L,I) :- inv(L, [], I).
% inv(Lista, Acumulador, Invertida)
inv([], I, I).
inv([X|R], A, I) :- inv(R, [X|A], I).
```

Medir el speedup del programa "invertir2" con respecto al programa "invertir1". Compare también la eficiencia de estos programas con respecto al predicado predefinido "reverse". (Ayuda: definir un predicado capaz de generar una lista de miles de elementos.)

(Observación: En el libro PROGRAMACIÓN LÓGICA, TEORÍA Y PRÁCTICA, de María Alpuente y Pascual Julián, página 193, puede encontrar más información sobre este ejercicio y el uso de parámetros de acumulación.)

Ejercicio 19.
La siguiente definición de "longitud1" es una versión ingenua que permite calcular la longitud de una lista:

```
% longitud1(L,N), N es la longitud de la lista L
longitud1([],0).
longitud1([H|T],N) :- longitud1(T,Z), N is Z +1.
```

Utilizando parámetros de acumulación, definir una versión “longitud2(L,N)” más eficiente.

Medir el speedup del programa “longitud2” con respecto al programa “longitud1”. Compare también la eficiencia de estos programas con respecto al predicado “length” predefinido en Prolog.
(Ayuda: definir un predicado capaz de generar una lista de miles de elementos.)

Ejercicio 20.
La siguiente definición de “suma1” es una versión ingenua que permite calcular la suma de los elementos de una lista de enteros:

```
% suma1(L,N), N es la suma de los elementos de una lista de enteros.
suma1([],0).
suma1([H|T],N) :- suma1(T,Z), N is Z +H
```

Utilizando parámetros de acumulación, definir una versión “suma2(L,N)” más eficiente.

Medir el speedup del programa “suma2” con respecto al programa “suma1”.

Ejercicio 21.
La sucesión de Fibonacci está compuesta por los números: a1=1, a2=1, a3=2, a4=3, a5=5, a6=8, ... Esta sucesión puede definirse por intensión en los siguientes términos

a1=1;
a2=1;
an = an-1 + an-2 si n>2.

La anterior definición puede trasladarse a sintaxis Prolog de forma inmediata

```
fib(1,1).
fib(2,1).
fib(N,F) :- N>2, H1 is N-1, H2 is N-2,
fib(H1,F1),fib(H2,F2),
F is F1+F2.
```

Sin embargo, este programa es muy ineficiente, pues tiende a rehacer muchos cálculos previamente efectuados. Podemos confirmar lo anterior sin más que inspeccionar la

traza del objetivo “?- fib(6, F).'', por ejemplo, puede verse que el número fib(3, _) se calcula en tres ocasiones. Utilizando una función auxiliar con parámetros de acumulación, definir un predicado fib2(N, F) que compute eficientemente el N-esimo número de Fibonacci evitando cómputos redundantes. Medir el speedup del programa “fib2” con respecto al programa “fib”.

Última modificación: jueves, 13 de marzo de 2014, 19:25:25