Taller de Investigación II
Informe de Trabajo

Taller: Análisis de programas por Interpretación Abstracta

Tema: Estimación de los coeficientes del análisis de complejidad mediante técnicas estadísticas.

Director: Dr. Francisco Bueno

Alumno: Edison Mera Menéndez

Madrid, Septiembre de 2004.

Índice

1. Introducción **3**
- 1.1. Descripción del Trabajo 3
- 1.2. Objetivo del Trabajo 3

2. Consideraciones preliminares **4**

3. Funcionamiento del sistema **6**

4. Utilización **6**
- 4.0.1. Archivo `qsort3.pl` 6

5. Conclusiones y Trabajo Futuro **9**

6. Bibliografía **11**

A. Transcripción del Código Fuente **12**
- A.1. Librería `math` 12
  - A.1.1. Archivo `mmatrix.pl` 12
  - A.1.2. Archivo `stat.pl` 17
- A.2. Paquete `costmodel` 21
  - A.2.1. Archivo `costmodel.pl` 21
  - A.2.2. Archivo `costmodel_rt.pl` 22
  - A.2.3. Archivo `costmodel_tr.pl` 24

1. Introducción

Los trabajos realizados en este taller forman parte de los requisitos para la obtención de los créditos necesarios para el Doctorado en Inteligencia Artificial y Ciencias de la Computación.

1.1. Descripción del Trabajo

La interpretación abstracta permite analizar programas en un dominio definido abstracto de valores, y una versión abstracta del programa.

La ventaja que ofrecen los lenguajes de programación lógica frente a otros lenguajes radica en que éstos permiten definir semánticas de forma más sencilla, mediante conceptos lógicos, y así el análisis resulta más fácil.

Existen varios temas que se pueden desarrollar dentro de la interpretación abstracta, y uno de ellos relacionado con el presente trabajo es el análisis de coste. Dicho análisis fue ampliamente desarrollado por P. López en su tesis doctoral, en relación con el control de granularidad de programas paralelizables [1]. Éste puede ser de tres tipos: análisis de cotas inferiores de coste, análisis de cotas superiores de coste y análisis de coste medio.

Estos costes a su vez, están relacionados con el tiempo que tarda un programa en ejecutarse. El coste en este caso dependerá de determinados factores dependientes del sistema.

1.2. Objetivo del Trabajo

El presente trabajo está orientado a la estimación de los parámetros en los modelos de coste medio, con ayuda de un profiler, una vez que se ha obtenido un posible modelo ya sea mediante determinadas técnicas de análisis o manualmente. Adicionalmente, ha sido pensado para establecer aserciones de coste en un determinado procedimiento, y de esa forma podremos detectar problemas del sistema desde el punto de vista del coste.

Es interesante notar que la búsqueda de dichos parámetros está lejos de ser trivial. El presente trabajo lo hemos limitado a modelos lineales generalizados, sin embargo hay que tener presente que en situaciones más reales los modelos pueden ser no lineales.

2. Consideraciones preliminares

En este trabajo, debemos tomar en cuenta varios factores. Primero está la correcta recopilación de los datos que necesitamos. Para la toma de los tiempos de ejecución tenemos que usar la librería de tiempos de alta resolución `hrtimer`. La razón para eso es que si no debido a la baja resolución de las librerías estándares de tiempo no se podrán obtener mediciones válidas.

Este problema ya fue expuesto en el taller previo relacionado con el desarrollo de un profiler para prolog. El tiempo adecuado para hacer este sistema es el llamado "runtime", que se define como el tiempo de usuario asignado por el sistema ignorando el tiempo empleado por el emulador para hacer recolección de basura, reasignamiento de memoria u otras tareas ocultas que puedan alterar la medición del coste.

Lo segundo a tener presente es que dado un predicado, dependiendo del tipo de argumentos y los modos de llamada, se debe conocer la distribución de los datos de entrada y dependiendo de eso se obtiene el coste. Para hacerlo se ha propuesto una forma de especificar dichas distribuciones en forma de aserciones de modelo de coste, lo cual veremos más adelante con un ejemplo.

Debemos además considerar que dado los valores concretos de los parámetros, debemos extraer información relevante para la estimación del coste, como es por ejemplo, la longitud de una lista, el grado de complejidad o profundidad de un término prolog, el valor concreto de un número, etc.

Finalmente, el modelo a plantearse será una expresión matemática cuyas variables independientes son dichos parámetros relevantes para el coste.

El tipo de modelos que hemos considerado en este trabajo son los modelos lineales generalizados.

Definición 2.0.1 (Modelo lineal generalizado) *Sean $x_1, x_2, ..., x_m$ un conjunto de variables independientes, $x_i \in \mathcal{R}^n$, siendo $\mathcal{R}^n$ el espacio real de dimensión n. Supongamos que para cada x_i se ha observado un y_i, la cual es nuestra variable dependiente, es decir:*

$$y_i = \mathcal{F}(x_i), i = 1, 2, ..., m$$

Un modelo lineal generalizado para estos datos será una expresión de la forma:

$$y_i = \beta_1 f_{1i} + \beta_2 f_{2i} + ... + \beta_p f_{pi} + e_i$$

Donde y_i es la i-ésima respuesta, f_{ji} es la j-ésima función base evaluada en la i-ésima observación, y e_i es el i-ésimo error estadístico.

Los coeficientes $\beta_1, ..., \beta_p$ son calculados mediante el método de mínimos cuadrados, que consiste en minimizar $\Sigma_i e_i^2$, el error total o la suma residual de cuadrados.

En otras palabras, debemos minimizar

$$\Sigma_i [y_i - (\Sigma_j \beta_j f_{ji})]^2$$

Sin entrar en detalles, indicaremos brevemente el método empleado para calcular estos coeficientes.

Definimos la siguiente matriz:

$$Z = \begin{bmatrix} y_1 & f_{11} & f_{12} & ... & f_{1p} \\ y_2 & f_{21} & f_{22} & ... & f_{2p} \\ ... & ... & ... & ... & ... \\ y_m & f_{m1} & f_{m2} & ... & f_{mp} \end{bmatrix}$$

Recordemos que las f_{ij} son funciones evaluadas en x_i. El producto $Z^T * Z$ está compuesto de las siguientes matrices, que definimos a continuación:

$$Z^T * Z = \left[\begin{array}{c|c} \Sigma_i yi & Y^T \\ \hline Y & X \end{array} \right]$$

Finalmente, si

$$B = \begin{bmatrix} \beta_1 \\ \beta_2 \\ ... \\ \beta_p \end{bmatrix}$$

El vector compuesto por los coeficientes que queremos estimar, éste se calculará de la siguiente forma:

$$B = X^{-1} * Y$$

Siendo X^{-1} la matriz inversa de X. Aunque esa es la expresión que define B, en la práctica lo que se hace es resolver B en la ecuación:

$$X * B = Y$$

Usando métodos conocidos del álgebra lineal.

3. Funcionamiento del sistema

El sistema consta de dos partes importantes. Una es la implementación de la librería estadística encargada de calcular el modelo lineal, que se encuentra implementada en el subdirectorio 'contrib/math' y otra es un paquete que nos permite escribir aserciones de coste en el programa, implementado en el subdirectorio 'contrib/costmodel', esto es para facilitar el manejo del sistema. En dicho paquete se encuentran definidos predicados para evaluar un determinado programa y estimar funciones de coste, según nuestro modelo planteado.

Cuando una aserción de coste es encontrada, ésta se expande a un predicado que contiene la información puesta por dicha aserción de coste, y luego si queremos encontrar el modelo llamamos al predicado `docostmodel/6` con los parámetros adecuados, que son el módulo, el predicado, la distribución usada para generar datos de prueba, el modelo, y nos devuelve la lista de datos y los coeficientes calculados, como veremos en la siguiente sección.

4. Utilización

Para ver cómo usar este sistema, vamos a desarrollar un ejemplo.

Supongamos que tenemos el módulo `qsort3.pl`, que listamos a continuación:

4.0.1. Archivo `qsort3.pl`

```
:- module(qsort3, _, [costmodel]).

:- costmodel qsort(list(number), term) :: [list(uniform(-1,1),
   uniform(1,4000)), none] => model(100, [X,_Y], [N], (length(X,N)),
   linearmodel([1, N, N * log(N)])).

qsort(A, B) :-
qsort_(A, B, []).

qsort_([], R, R).
qsort_([X|L], R, R0) :-
split(L, X, L1, L2),
qsort_(L2, R1, R0),
qsort_(L1, R, [X|R1]).

split([],_,[],[]).
split([X|L],Y,[X|L1],L2) :-
X =< Y, !,
split(L,Y,L1,L2).
split([X|L],Y,L1,[X|L2]) :-
split(L,Y,L1,L2).
```

Como se puede observar, en él se ha puesto una aserción de coste:

```
:- costmodel qsort(list(number), term) :: [list(uniform(-1,1),
   uniform(1,4000)), none] => model(100, [X,_Y], [N], (length(X,N)),
   linearmodel([1, N, N * log(N)])).
```

La aserción es de la forma
`:- costmodel Pred :: Distribution =>Model`
y especifica lo siguiente:

1. `Pred` es el nombre del predicado y los tipos de los argumentos.

2. `Distribution` es la lista con la distribución utilizada para generar datos de prueba. Actualmente se encuentran implementadas distribuciones uniformes y gausianas. Si se especifica 'none' entonces no se generarán datos de prueba para el argumento correspondiente. Veamos cómo se definen estas distribuciones:

   - `uniform(A,B)` indica que los valores de ese parámetro se generarán siguiendo una distribución uniforme en el intervalo $[A, B]$.
   - `list(DistValue, DistLength)` es usada para un tipo de datos lista, e indica que la longitud de la lista sigue una distribución `DistLength` y que los datos de dicha lista sigue una distribución `DistValue`.
   - `gausian(Mean, StdDev)` indica que los valores de dicho parámetro se generan siguiendo una distribución gausiana de media `Mean` y desviación estándar `StdDev`.

3. `Model` indica el modelo de coste a ser utilizado, y tiene la siguiente expresión:
   `model(Number, ArgList, ParamList, GoalCost, SpecificModel)`.

   - `Number` es el número de datos de prueba a ser generados.
   - `ArgList` es una lista de variables que representa los argumentos del predicado `Pred`
   - `ParamsList` es la lista de los parámetros de nuestro modelo de coste, es decir los x_i.
   - `GoalCost` es una expresión de Prolog que sirve para calcular los valores de las variables en `ParamsList` cuando las variables en `ArgList` están instanciadas.

- `SpecificModel` indica qué modelo emplear. Actualmente se encuentra implementado el modelo lineal generalizado, cuya expresión es la siguiente: `linearmodel(BaseList)`, donde BaseList es una lista de expresiones aritméticas que representan las funciones base f_{ji}.

Si ejecutamos nuestro ejemplo, obtendremos los coeficientes de nuestro modelo:

```
Ciao-Prolog 1.11 #257: mar ago 31 09:53:16 CEST 2004
?- use_module(library(hrtimer)).

yes
?- use_module(library('costmodel/costmodel_rt')).

yes
?- use_module(qsort3).

yes
?- docostmodel(qsort3,Pred,Dist,Model,_Data,B).
yes

B = [0.07960826969492942,-0.0006914459724951319,0.0006245282789956145],
Dist = [list(uniform(-1,1),uniform(1,4000)),none],
Model = model(100,[_B,_],[_A],length(_B,_A),linearmodel([1,_A,_A*log(_A)])),
Pred = qsort(list(number),term) ?
yes
```

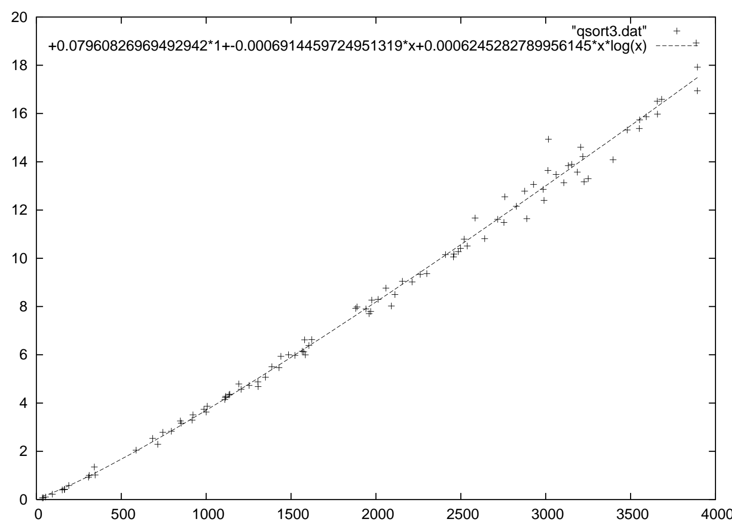

Fig. 1: Comparación entre el modelo y los datos observados

La figura muestra los datos generados y la gráfica del modelo propuesto, donde podemos observar cómo el modelo se ajusta a los datos.

5. Conclusiones y Trabajo Futuro

Hemos descrito un sistema que nos permite especificar restricciones de coste en los predicados de prolog, y que dichas restricciones realmente se verifiquen de forma práctica. Además expusimos un sistema práctico basado en conceptos estadísticos bien estudiados y conocidos.

Sin embargo, algunos puntos aún quedan sin abordar. en [1] se prouso crear un analizador que encuentre expresiones para el coste medio. El problema es que se requiere saber de antemano la distribución de los datos para que dicho coste medio se ajuste al coste observado en la práctica.

En este trabajo se presentó un estimador de parámetros basado en modelos lineales. Sería interesante incorporar a dicho sistema un estimador de parámetros basado en modelos no lineales [2,3,4].

Una consecuencia interesante del uso de esta herramienta es que nos permite detectar errores de nuestro sistema relacionados con el coste. Por ejemplo, si estamos creando un programa cuyo coste medio sabemos que es $C_1 * O(n)$, pero luego al usar realmente dicho sistema el coste observado es mayor, por ejemplo $C_2 * O(n^2)$, sabremos que dos cosas pueden estar mal: o la aserción que hemos puesto, o nuestro programa.

Otra característica interesante que se puede incorporar al sistema es el idear pruebas que nos permitan detectar si el modelo escogido es el adecuado para representar la complejidad. Existen análisis estadísticos que nos pueden decir si un modelo está sobreajustado (exceso de parámetros tomados en cuenta), o si los parámetros seleccionados son insuficientes, y nos ayudan a evaluar la calidad del modelo escogido.

6. Bibliografía

1. P. López García. Control de Granularidad en la ejecución Paralela de Programas Lógicos Mediante Técnicas de Análisis y Transformación. Tesis Doctoral presentada en la Universidad Politécnica de Madrid previa a la obtención del título de Doctor en Informática. Junio 2000.

2. Bates, D. M. and Watts, D. G. Nonlinear Regression Analysis and Its Applications, John Wiley & Sons, New York. 1988.

3. Meyer, R. R. and Roth, P. M. Modified Damped Least Squares: An Algorithm for Nonlinear Estimation. J. Inst. Math. Appl. 9, 218-233. 1972.

4. Ratkowsky, David A. Nonlinear Regression Modeling, A Unified Practical Approach. Marcel Dekker. New York. 1983.

5. Johnson, J. D. Applied Multivariante Data Analyisis. Volume I: Regression and Experimental Design, Springer - Verlang. New York. 1991.

6. Sahai, H, and Ageel, M. The Analyisis of Variance: Fixed, Random and Mixed Models, Birkhauser, Boston. 2000.

7. Searle, S. R. Linear Models for Unbalanced Data. John Wiley & Sons. New York. 1987.

A. Transcripción del Código Fuente

La implementación de este estimador de modelos de coste consta de varias partes, todas implementadas en prolog. Sin embargo, para su correcto funcionamiento es imprescindible el empleo de la librería hrtimer, que obviaremos aquí por ya haber sido descrita en el taller I.

A.1. Librería math

A.1.1. Archivo mmatrix.pl

```
:- module(mmatrix,[
lsum/2,
lsum2/2,
mtrans/2,
mmultiplyt/3,
vmultiply/3,
rmultiply/3,
rdiv/3,
rmuladd/4,
vadd/3,
madd/3,
mreverse/2,
msolve/3,
mmsolve/3,
mtriang_to_rect/2
],[assertions, regtypes]).

:- use_module(library(lists)).

:- use_module(library('truster/truster_rt')).

:- comment(title, "Matrix operations").

:- comment(author, "Edison Mera").

:- comment(module, "A complete package of vector and matrix
   operations.  For use with the statistic library.").

%:- entry mmultiply(X,Y,Z): ( var(Z), list(X,list(num)), list(X,list(num)) ).
:- entry mmultiply(X,Y,Z): ( var(Z), ground(X), ground(Y)).

% :- regtype vector(X) # "@var{X} is a vector of numbers.".

% vector(X) :- list(X,number).

% :- regtype matrix(X) # "@var{X} is a matrix, i.e., a vector of vector of numbers.".

% matrix(X) :- list(X,vector).

:- true pred lsum(List,Sum): list(number) * term => list(number) *
   number # "Unifies @var{Sum} with the total sum of the numbers in
   the list @var{List}.".

lsum([],0).
lsum([X|Xs],S):-
lsum(Xs, S1),
S is X + S1.
```

```
:- true pred lsum2(List,Sum2): list(number) * term => list(number) *
   number # "Unifies @var{Sum2} with the total sum of the square of
   the numbers in the list @var{List}.".

lsum2([],0).
lsum2([X|Xs],S) :-
lsum2(Xs, S1),
S is X*X + S1.

:- true pred vmultiply(Vector1,Vector2,Result): list(number) *
   list(number) * term => list(number) * list(number) * number #
   "Unifies @var{Result} with the scalar product between the vectors
   @var{Vector1} and @var{Vector2}.".

%scalar product between two vectors
vmultiply([],[],0).
vmultiply([H1|T1], [H2|T2], Result):-
vmultiply(T1,T2, Newresult),
Product is H1*H2,
Result is Product+Newresult.

:- true pred addcol(Vector,Matrix,Result): list(number) *
   list(list(number)) * term => list(number) * list(list(number)) *
   list(list(number)) # "Unifies @var{Result} with the matrix obtained
   adding @var{Vector} as the first column of @var{Matrix}.".

addcol([],[],[]).
addcol([A|As],[],[[A]|Cs]) :-
addcol(As,[],Cs).
addcol([A|As],[B|Bs],[[A|B]|Cs]) :-
addcol(As,Bs,Cs).

:- true pred mtrans(Matrix,Trans): list(list(number)) * term =>
   list(list(number)) * list(list(number)) # "Unifies @var{Trans} with
   the transposed matrix of @var{Matrix}.".

mtrans([],[]).
mtrans([V|Ms],T) :-
mtrans(Ms,Xs),
addcol(V,Xs,T).

:- true pred mmultiplyt(Matrix1,Matrix2,Result) : list(list(number)) *
   list(list(number)) * term => list(list(number)) *
   list(list(number)) * list(list(number)) # "Unifies @var{Result}
   with the matricial product between the matrices @var{Matrix1} and
   the transposed matrix of @var{Matrix2}.".

mmultiplyt([],_,[]).
mmultiplyt([V0|Rest], V1, [Result|Others]):-
mmultiplyt(Rest, V1, Others),
multiplyt(V1,V0,Result).

multiplyt([],_,[]).
multiplyt([V0|Rest], V1, [Result|Others]):-
multiplyt(Rest, V1, Others),
vmultiply(V0,V1,Result).

:- true pred mmultiply(Matrix1,Matrix2,Result): list(list(number)) *
   list(list(number)) * term => list(list(number)) *
   list(list(number)) * list(list(number)) # "Unifies @var{Result}
   with the matricial product between the matrices @var{Matrix1} and
```

```
   @var{Matrix2}.".

mmultiply(A,B,C) :-
mtrans(B,B2),
mmultiplyt(A,B2,C).

:- true pred rmultiply(Scalar,Vector,Result): number * list(number) *
   term => number * list(number) * list(number) # "Unifies
   @var{Result} with the scalar product between @var{Scalar} and
   @var{Vector}.".

rmultiply(_,[],[]).
rmultiply(R,[X|Xs],[Y|Ys]) :-
  Y is R*X,
  rmultiply(R,Xs,Ys).

:- true pred rdiv(Scalar,Vector,Result): number * list(number) * term
   => number * list(number) * list(number) # "Unifies @var{Result}
   with the scalar product between 1.0/@var{Scalar} and
   @var{Vector}.".

rdiv(_,[],[]).
rdiv(R,[X|Xs],[Y|Ys]) :-
Y is X/R,
rdiv(R,Xs,Ys).

:- true pred rmuladd(Scalar,Mul,Add,Result): number * list(number) *
   list(number) * term => number * list(number) * list(number) *
   list(number) # "Unifies @var{Result} with the scalar product
   between @var{Scalar} and @var{Mul}, plus @var{Add}.  In other
   words, @var{Result}=@var{Scalar}*@var{Mul}+@var{Add}.".

rmuladd(_,[],B,B).
rmuladd(R,[A|As],[B|Bs],[C|Cs]) :-
C is R*A+B,
rmuladd(R,As,Bs,Cs).

:- true pred vadd(X,Y,Z): list(number) * list(number) * term =>
   list(number) * list(number) * list(number) # "Unifies @var{Z} with
   the sum of the vectors @var{X} and @var{Y}.".

vadd([],Y,Y).
vadd(X,[],X).
vadd([X|Xs],[Y|Ys],[Z|Zs]) :-
Z is X+Y,
vadd(Xs,Ys,Zs).

:- true pred madd(A,B,C): list(list(number)) * list(list(number)) *
   term => list(list(number)) * list(list(number)) *
   list(list(number)) # "Unifies @var{Z} with the sum of the matrices
   @var{X} and @var{Y}.".

madd([],Y,Y).
madd(X,[],X).
madd([X|Xs],[Y|Ys],[Z|Zs]) :-
vadd(X,Y,Z),
madd(Xs,Ys,Zs).

pivot([_|_],_,[],[],[],[]).
pivot([A|Ap],Bp,[[A1|Ar]|As],[B|Bs],[C|Cs],[D|Ds]) :-
R is -A1/A,
rmuladd(R,Ap,Ar,C),
```

```
D is R*Bp+B,
pivot([A|Ap],Bp,As,Bs,Cs,Ds).

selpivot([A],[B],A,B,[],[]).
selpivot([[A|Ar]|As],[B|Bs],Ap,Bp,C,D) :-
selpivot(As,Bs,[Ap1|Ap1s],Bp1,C1,D1),
(
    abs(A) >= abs(Ap1) ->
    Ap=[A|Ar],
    Bp=B,
    C=[[Ap1|Ap1s]|C1],
    D=[Bp1|D1];
    Ap=[Ap1|Ap1s],
    Bp=Bp1,
    C=[[A|Ar]|C1],
    D=[B|D1]
).

reduce([],[],[],[]).
reduce([A|As],[B|Bs],[C|Cs],[D|Ds]) :-
selpivot([A|As],[B|Bs],C,D,As2,Bs2),
pivot(C,D,As2,Bs2,C1,D1),
reduce(C1,D1,Cs,Ds).

reduceback(_,[],[],[],[]).
reduceback(R,[[A|Ar]|As],[B|Bs],[Ar|Cs],[D|Ds]) :-
D is B-R*A,
reduceback(R,As,Bs,Cs,Ds).

eliminate([],[],[]).
eliminate([[A]|As],[B|Bs],[C|Cs]) :-
C is B/A,
reduceback(C,As,Bs,Cs1,Ds1),
eliminate(Cs1,Ds1,Cs).

:- true pred mreverse(A,B): list(list(number)) * term =>
   list(list(number)) * list(list(number)) # "Takes the matrix
   @var{A}, reverses the order of the columns, and unifies it with
   @var{B}.".

mreverse([],[]).
mreverse([A|As],[B|Bs]) :-
reverse(A,B),
mreverse(As,Bs).

:- true pred msolve(A,B,X): list(list(number)) * list(number) * term
   => list(list(number)) * list(number) * list(number) # "Solve
   @var{X} in the linear system @var{A}*@var{X}=@var{B}, where @var{B}
   is a vector.".

msolve(A,B,X) :-
reduce(A,B,A1,B1),
mreverse(A1,A2),
reverse(A2,A3),
reverse(B1,B2),
eliminate(A3,B2,X1),
reverse(X1,X).

mpivot([_|_],_,[],[],[],[]).
mpivot([A|Ap],Bp,[[A1|Ar]|As],[B|Bs],[C|Cs],[D|Ds]) :-
R is -A1/A,
rmuladd(R,Ap,Ar,C),
```

```
rmuladd(R,Bp,B,D),
mpivot([A|Ap],Bp,As,Bs,Cs,Ds).

mreduce([],[],[],[]).
mreduce([A|As],[B|Bs],[C|Cs],[D|Ds]) :-
selpivot([A|As],[B|Bs],C,D,As2,Bs2),
mpivot(C,D,As2,Bs2,C1,D1),
mreduce(C1,D1,Cs,Ds).

mreduceback(_,[],[],[],[]).
mreduceback(R,[[A|Ar]|As],[B|Bs],[Ar|Cs],[D|Ds]) :-
rmuladd(-A,R,B,D),
mreduceback(R,As,Bs,Cs,Ds).

meliminate([],[],[]).
meliminate([[A]|As],[B|Bs],[C|Cs]) :-
rdiv(A,B,C),
mreduceback(C,As,Bs,Cs1,Ds1),
meliminate(Cs1,Ds1,Cs).

:- true pred mmsolve(A,B,X): list(list(number)) * list(list(number)) *
   term => list(list(number)) * list(list(number)) *
   list(list(number)) # "Solve the linear system
   @var{A}*@var{X}=@var{B}, where @var{B} is a matrix.".

mmsolve(A,B,X) :-
mreduce(A,B,A1,B1),
mreverse(A1,A2),
reverse(A2,A3),
reverse(B1,B2),
meliminate(A3,B2,X1),
reverse(X1,X).

addcolumnzero([],[],[]).
addcolumnzero([A|As],[[0|A]|Bs],[0|Cs]) :-
addcolumnzero(As,Bs,Cs).

midentity(0,[]).
midentity(N,[[1|I]|Is]) :-
N1 is N-1,
midentity(N1,Is1),
addcolumnzero(Is1,Is,I).

minverse(A,B) :-
length(A,N),
midentity(N,I),
mmsolve(A,I,B).

:- true pred mtriang_to_rect(A,B): list(list(number)) *
   list(list(number)) => list(list(number)) * list(list(number)) #
   "Converts the triangular matrix @var{A} to the rectangular matrix
   @var{B}.  Note that a triangular matrix is as follow:

   A = [[A11,A12,    ...,A1N],
            [A22,A23,...,A2N],
                         ....
                        [ANN]]

   And the respective rectangular matrix is

   B = [[A11,A12,    ...,A1N],
        [A12,A22,A23,...,A2N],
```

```
        [A13,A23,A33,...,A3N],
                    ...
        [A1N,A2N,A3N,...,ANN]]

".

mtriang_to_rect([],[]).
mtriang_to_rect([[A|Ar]|As],[[A|Ar]|Bs]) :-
mtriang_to_rect(As,Bs1),
addcol(Ar,Bs1,Bs).


:- comment(version_maintenance,dir('../../version')).

:- comment(version(1*11+210,2004/03/18,13:01*01+'CET'), "Added mmatrix
   to the changelog.  This library contains the methods to work with
   vectors and matrices.  (Edison Mera)").
```

A.1.2. Archivo stat.pl

```
:- module(stat, [
lsumstat/3,
lsumstat_aux/5,
lsumstat/4,
lsumstat_aux/7,
laverage/2,
lvariance/2,
variance/4,
lsumstat2/7,
lsumstat2_aux/13,
covariance/5,
lcovariance2/2,
regression/9,
lregression/4,
mlregression/2,
genregression/4
], [assertions,regtypes]).

:- use_module(library(lists)).
:- use_module(library('math/mmatrix'), [rmultiply/3, madd/3,
   mtriang_to_rect/2, msolve/3, vmultiply/3, rmultiply/3, vadd/3]).

:- comment(title,"Statistical Utilities.").

:- comment(author,"Edison Mera").

:- comment(module,"A complete library of statistical utilities.  This
   library will be used in conjunction with the profiler tools.").

:- push_prolog_flag(multi_arity_warnings, off).


:- true pred lsumstat(List,Number,Sum): list(number) * term * term =>
   list(number) * number * number # "Unifies @var{Number} with the
   length of the list and @var{Sum} with the total sum of the numbers
   in the list @var{List}.".

lsumstat(X,N,S):-
lsumstat_aux(X,0,0,N,S).

:- true pred lsumstat_aux(List,Number0,Sum0,Number,Sum): list(number)
```

```
   * number * number * term * term => list(number) * number * number *
   number * number # "Similar to @pred{lsumstat/3}, but takes
   @var{Number0} and @var{Sum0} as the initial values of @var{Number}
   and @var{Sum}.".

lsumstat_aux([],N,S,N,S).

lsumstat_aux([X|Xs],N1,S1,N,S):-
N2 is N1+1,
S2 is S1+X,
lsumstat_aux(Xs,N2,S2,N,S).

:- true pred lsumstat(List,Number,Sum1,Sum2): list(number) * term *
   term * term => list(number) * number * number * number # "Unifies
   @var{Number} with the length of the list, @var{Sum1} with the total
   sum of the numbers in the list and @var{Sum2} with the total sum of
   the squarers of the numbers in the list @var{List} respectively.".

lsumstat(X,N,S1,S2):-
lsumstat_aux(X,0,0,0,N,S1,S2).

:- true pred lsumstat_aux(List,Number0,Sum10,Sum20,Number,Sum1,Sum2):
   list(number) * number * number * number * term * term * term =>
   list(number) * number * number * number * number * number * number
   # "Similar to @pred{lsumstat/4}, but takes @var{Number0},
   @var{Sum10} and @var{Sum20} as the initial values of @var{Number},
   @var{Sum1} and @var{Sum2} respectively.".

lsumstat_aux([],N,S1,S2,N,S1,S2).
lsumstat_aux([X|Xs],N1,S11,S21,N,S1,S2) :-
        N2 is N1+1,
S12 is S11+X,
S22 is S21+X*X,
lsumstat_aux(Xs,N2,S12,S22,N,S1,S2).

:- pop_prolog_flag(multi_arity_warnings).

:- true pred laverage(List,Average): list(number) * term =>
   list(number) * number # "Unifies @var{Average} with the average of
   the list @var{List}.".

laverage(L,A) :-
lsumstat(L,N,S),
A is S/N.

:- true pred lvariance(List,Variance): list(number) * term =>
   list(number) * number # "Unifies @var{Variance} with the variance
   of the list @var{List}.".

lvariance(L,V) :-
lsumstat(L,N,S1,S2),
variance(N,S1,S2,V).

:- true pred variance(N,Sum1,Sum2,Variance): number * number * number *
   term => number * number * number * number # "Unifies @var{Variance} with the
   variance, being @var{N} the number of data, @var{Sum1} the sum of
   the data and @var{Sum2} the sum of the square of the data".

variance(N,S1,S2,V):-
V is (S2 - (S1*S1)/N)/N.

:- regtype point(P).
```

```
point(P) :-
P=(X,Y),
number(X),
number(Y).

:- true pred lsumstat2(List,N,Sx,Sy,Sx2,Sxy,Sy2): list(point) * term *
   term * term * term * term * term => list(point) * number * number *
   number * number * number * number # "Unifies N with the length, Sx
   with the sum of the independent variable, Sy with the sum of the
   dependent variable, Sx2 with the sum of squares of the independent
   variable, Sxy with the sum of the product between the independent
   and the dependent variables, and Sy2 with the sum of squares of the
   dependent variable.".

lsumstat2(L,N,Sx,Sy,Sx2,Sxy,Sy2):-
lsumstat2_aux(L,0,0,0,0,0,0,N,Sx,Sy,Sx2,Sxy,Sy2).

:- true pred lsumstat2_aux(List, N0, Sx0, Sy0, Sx20, Sxy0, Sy20, N,
   Sx, Sy, Sx2, Sxy, Sy2) : list(point) * number * number * number *
   number * number * number * term * term * term * term * term * term
   => list(point) * number * number * number * number * number *
   number * number * number * number * number * number * number #
   "Same as @pred{lsumstat/7} but taken initial values for the
   returned data.".

lsumstat2_aux([],N,Sx,Sy,Sx2,Sxy,Sy2,N,Sx,Sy,Sx2,Sxy,Sy2).
lsumstat2_aux([L|Ls],N1,Sx1,Sy1,Sx21,Sxy1,Sy21,N,Sx,Sy,Sx2,Sxy,Sy2):-
(X,Y)=L,
        N_2 is 1+N1,
Sx_2 is X+Sx1,
Sy_2 is Y+Sy1,
Sx2_2 is X*X+Sx21,
Sxy_2 is X*Y+Sxy1,
Sy2_2 is Y*Y+Sy21,
lsumstat2_aux(Ls,N_2,Sx_2,Sy_2,Sx2_2,Sxy_2,Sy2_2,N,Sx,Sy,Sx2,Sxy,Sy2).

:- true pred covariance(N,Sx,Sy,Sxy,C): number * number * number *
   number * term => number * number * number * number * number #
   "Unifies @var{C} with the covariance.".

covariance(N,Sx,Sy,Sxy,C):-
C is (Sxy - (Sx*Sy)/N)/N.

:- true pred lcovariance2(List,C): list(point) * term => list(point) *
   number # "Unifies @var{C} with the covariance of the point list
   @var{L}.".

lcovariance2(L,C) :-
lsumstat2(L,Sx,Sy,_Sx2,Sxy,_Sy2,N),
covariance(N,Sx,Sy,Sxy,C).

:- true pred regression(N, Sx, Sy, Sx2, Sxy, Sy2, B0, B1, R): number *
   number * number * number * number * number * term * term * term =>
   number * number * number * number * number * number *
   number * number # "Unifies B0, B1 with the parameters of the linear
   regression, and R with the correlation rate.".

regression(N,Sx,Sy,Sx2,Sxy,Sy2,B0,B1,R):-
covariance(N,Sx,Sy,Sxy,C),
variance(N,Sx,Sx2,Vx),
B1 is C/Vx,
```

```
B0 is (Sy-B1*Sx)/N,
variance(N,Sy,Sy2,Vy),
R is C/sqrt(Vx*Vy).

:- true pred lregression(L, B0, B1, R): list(point) * term * term *
   term => list(point) * number * number * number # "Given the point
   list @var{L}, Unifies B0, B1 with the parameters of the linear
   regression, and R with the correlation rate.  Remember that the
   linear model is as follows: Y^ = B0 + B1 * X, where Y^ is the
   expected value of the dependent variable Y.".

lregression(L, B0, B1, R) :-
lsumstat2(L, N, Sx, Sy, Sx2, Sxy, Sy2),
regression(N, Sx, Sy, Sx2, Sxy, Sy2, B0, B1, R).

%triangular product combinations.
tcom([],[]).
tcom([X|Xs],[C2|C1]) :-
rmultiply(X,[X|Xs],C2),
tcom(Xs,C1).

:- true pred ttransmul(X,S): list(list(number)) * term => list(list(number)) * list(list(number)) #
   "Unifies @var{S} with the matrix product @var{X}' * @var{X}, where @var{X} is triangular.".

ttransmul([],[]).
ttransmul([L|Ls],S) :-
ttransmul(Ls,S2),
% (X,Y)=L,
% tcom([Y|X],C),
tcom(L,C),
madd(C,S2,S).

:- true pred mlregression(L,B): list * term => list * list #
   "Multivariant linear regression.  @var{L} contains the data and @var{B} the
   parameter list.".

mlregression(L,B):-
ttransmul(L,[[_|Y]|X]),
mtriang_to_rect(X,X1),
msolve(X1,Y,B).

:- true pred genregression(Data, Pattern, Vars, B): list * term * term
   * term => list * term * term * list # "Generic linear regression.
   @var{Data} contains the data, @var{Pattern} contains a list of
   arithmetic expressions that represent the components of the linear
   model. @var{Vars} contains the list of variables used in the
   expression Pattern.  For example, if the model is:

   Y = B00+B10*X1+B01*X2+B11*X1*X2

   To use @var{genregression/4} with a given data list we must write:

?- genregression([[3,2,2],[5,1,2],[7,1,1],[9,2,1]], [1,X1,X2,X1*X2],[X1,X2],B).

B = [2.99999999999959,6.000000000000256,2.000000000000256,-4.00000000000016] ?

yes

The exact model is:

   Y=3+6*X1+2*X2-4*X1*X2
```

```
".

genregression(Data, Pattern, Vars, B) :-
applydatatovar(Data, Pattern, Vars, L),
mlregression(L, B).

:- data genregression_elem/1.

applydatatovar([], _, _, []).
applydatatovar([[Y|D]|Ds], Pattern, Vars, [[Y|L]|Ls]) :-
(   Vars = D,
    evalpattern(Pattern, L),
    assertz_fact(genregression_elem(L)),
    fail
)
 ;
current_fact(genregression_elem(L)),
retract_fact(genregression_elem(L)),
applydatatovar(Ds, Pattern, Vars, Ls).

evalpattern([], []).
evalpattern([P|Ps],[L|Ls]) :-
L is P,
evalpattern(Ps, Ls).

% now we will define the equation that find the solution in (X'X)*B = (X'*Y)

tmultiply([],[],[]).
tmultiply([X|Xs],[B|Bs],[Y|Ys]) :-
vmultiply(X,[B|Bs],Y),
tmultiply(Xs,Bs,Ys1),
rmultiply(B,X,Ys2),
vadd(Ys1,Ys2,Ys).

% ?- main.
% [9.053849885836955,0.5202789721838336,0.2397463704130682]

% main :-
%  mlregression([[64,1,58,111],[78,1,84,131],[83,1,78,158],[88,1,81,147],
%   [89,1,82,121],[99,1,102,165],[101,1,85,174],[102,1,102,169]],A),
%  write(A).


:- comment(version_maintenance,dir('../../version')).

:- comment(version(1*11+211,2004/03/18,13:02*53+'CET'), "Added to the
   changelog.  This library provides statistic facilities, such as a
   complete set of multivariant linear regression predicates.  (Edison
   Mera)").
```

A.2. Paquete `costmodel`

A.2.1. Archivo `costmodel.pl`

```
:- load_compilation_module(library('costmodel/costmodel_tr')).

:- use_module(library('costmodel/costmodel_rt')).

:- add_sentence_trans(costmodel_def/3).
```

```
:- op(1050, xfy, ['->']).

:- op(1150, fx, [costmodel]).

:- op(1050, xfy, ['::']).

:- op(1050, xfy, ['=>']).
```

A.2.2. Archivo costmodel_rt.pl

```
:- module(costmodel_rt, _, [assertions, regtypes]).

:- use_module(library(random)).
:- use_module(library('math/stat')).
:- use_module(library('profiler/profiler_utils')).
:- use_module(library(terms)).
:- use_module(library(lists)).
:- use_module(library('make/system_extra')).


:- multifile costmodel/4.

interval(A,B,N,Data) :-
interval_(A,B,0,N,Data).

interval_(_A,B,N,N,[B]).
interval_(A,B,I,N,[D|Ds]) :-
D is (A * (N - I) + B * I ) / N,
J is I + 1,
interval_(A,B,J,N,Ds).

uniform(A, B, D) :-
random(R),
D is A + (B - A) * R.

gaussian(Mean, StdDev, D) :-
random(R1),
random(R2),
U1 is 2*R1 - 1,
S2 is U1**2 + (2*R2-1)**2,
(   S2 < 1 ->
    D is sqrt(-2*log(S2)/S2) * U1 * StdDev + Mean
;
    gaussian(Mean, StdDev, D)
).

% type_to_value(number, interval(A,B,N), Data) :-
%  interval(A,B,N,Data).

:- dynamic type_to_value/3.

type_to_value(int, uniform(A,B), Data) :-
random(A, B, Data).
type_to_value(number, uniform(A,B), Data) :-
uniform(A,B,Data).
type_to_value(number, gaussian(M,S), Data) :-
gaussian(M,S,Data).
type_to_value(list(T), list(DistValue, DistLength), Data) :-
type_to_value(int, DistLength, Length),
type_to_value_list(T, DistValue, Length, Data).
```

```
type_to_value(term, none, _).

type_to_value(cuadmatrix(T), cuadmatrix(DistValue, DistLength), Data) :-
type_to_value(int, DistLength, Length),
type_to_value_list2(T, DistValue, Length, Length, Data).

type_to_value_list(_T, _DistValue, 0, []).
type_to_value_list( T,  DistValue, N, [D|Ds]) :-
type_to_value(T, DistValue, D),
N2 is N - 1,
type_to_value_list(T, DistValue, N2, Ds).

:- regtype cuadmatrix(T).
cuadmatrix(Type,X) :-
list(list(Type),X).

type_to_value_list2(_T, _DistValue, _M, 0, []).
type_to_value_list2( T,  DistValue, M, N, [D|Ds]) :-
type_to_value_list(T, DistValue, M, D),
N2 is N - 1,
type_to_value_list2(T, DistValue, M, N2, Ds).

types_to_values([],[],[]).
types_to_values([T|Ts],[D|Ds],[V|Vs]) :-
type_to_value(T, D, V),
types_to_values(Ts, Ds, Vs).

types_to_values_list(_T, _D, 0, []).
types_to_values_list(T, D, N, [V|Vs]) :-
types_to_values(T, D, V),
N2 is N - 1,
types_to_values_list(T, D, N2, Vs).

% :- meta_predicate tests(goal,?,?).

% tests(_Pred, [], []).
% tests(Pred, [Value|Values], [[T|Value]|Ds]) :-
%  Pred =.. [Functor2|[Pred2]],
%  Pred2 =.. [Functor|_Value1],
%  Pred3 =.. [Functor|Value],
%  Pred4 =.. [Functor2|[Pred3]],
%  measure(Pred4, T),
%  tests(Pred, Values, Ds).

tests(_Module, _Functor, _Args, _Params, _GoalCost, [], []).
tests( Module,  Functor,  Args,  Params,  GoalCost, [Value|Values], [[T|Data]|Ds]) :-
atom_concat([Module, ':', Functor], F),
Pred =.. [F|Value],
measure(runtime, '$:'(Pred),T),
apply_cost_params(Value, Args, Params, GoalCost, Data),
tests(Module, Functor, Args, Params, GoalCost, Values, Ds).

:- data cost_param/1.

:- meta_predicate apply_cost_params(?,?,?,goal,?).

apply_cost_params(Value, Args, Params, GoalCost, Data) :-
(
    Args = Value,
    call(GoalCost),
    assertz_fact(cost_param(Params)),
    fail
```

```
;
    current_fact(cost_param(Data)),
    retract_fact(cost_param(Data))
).

estimate_costmodel(Module, Pred, Dist, model(N, Args, Params, GoalCost, linearmodel(Terms)), Data, B) :-
Pred =.. [Functor|Types],
types_to_values_list(Types, Dist, N, Values),
tests(Module, Functor, Args, Params, GoalCost, Values, Data),
genregression(Data, Terms, Params, B).

docostmodel(Module, Pred, Dist, Model, Data, B) :-
costmodel(Module, Pred, Dist, Model),
estimate_costmodel(Module, Pred, Dist, Model, Data, B).

dumpdataS([], []).
dumpdataS([D|Ds], [L|Ls]) :-
[Y,X] = D,
number_codes(X,XS),
number_codes(Y,YS),
list_concat([XS, " ", YS, "\n"], L),
dumpdataS(Ds,Ls).

dumpdata(Data, FileName) :-
dumpdataS(Data, StringList),
writef_list(StringList, FileName).

dumpmodelS([], [], []).
dumpmodelS([Coeff|Coeffs], [Func|Funcs], [L|Ls]) :-
number_codes(Coeff, CoeffS),
list_concat(["+", CoeffS, "*", Func], L),
dumpmodelS(Coeffs, Funcs, Ls).

dumpmodel(Coeffs, Funcs, FileName) :-
dumpmodelS(Coeffs, Funcs, StringList),
writef_list(StringList, FileName).

%:- costmodel qsort(list(number,N), term) :: list(uniform(-1,1,100),
%   uniform(0,100,100)) * none => model(100, linearmodel([1, N, N *
%   log(N)])).

% estimate_costmodel(mmatrix, mtrans(cuadmatrix(number),term),
% [cuadmatrix(uniform(-10,10),uniform(1,99)),none],
% model(100,[_B,_],[_A],length(_B,_A),linearmodel([1,_A,_A**2])), Data,
% B), dumpdata(Data, 'mmatrix_4.dat').
```

A.2.3. Archivo costmodel_tr.pl

```
:- module(costmodel_tr, [costmodel_def/3],[assertions]).

:- use_module(library(lists)).
:- use_module(library(aggregates)).

costmodel_def(end_of_file, Clauses, _M) :-
Clauses = [(:- multifile(costmodel/4)),
end_of_file].
costmodel_def((:- costmodel('::'(Pred, '=>'(Dist, Model)))), Clauses, M) :-
Clauses = [(costmodel(M, Pred, Dist, Model))].
% assertz_fact(cost(Pred, Dist, Model)).
```