Programación para la Inteligencia Artificial
Tarea 1

Arturo Márquez Flores
Maestría en Inteligencia Artificial
Universidad Veracruzana
CIIA – Centro de Investigación en Inteligencia Artificial
Sebastián Camacho No 5, Xalapa, Ver., México 91000
arturomf94@gmail.com
https://github.com/arturomf94/pia-mia

16 de Octubre del 2018

1. Pregunta 1

Las respuestas de esta sección tienen su base en el artículo de K. Knight [1]

1.1.

La diferencia entre *unificación* y *matching* es que en el caso de unificación la substitución se aplica a ambos términos, mientras en el *matching* se aplica sólo a uno. Por ejemplo, al unificar dos listas $[X, a]$ y $[a, Y]$ la substitución correspondiente sería $X = Y$, $Y = a$. Sin embargo, el *matching* no funciona aquí porque no existe un substitución tal que aplicada a $[X, a]$ haga que esta igual al término $[a, Y]$.

1.2.

Utilizando el *chequeo de ocurrencias*, los términos X y $f(X)$ no unifican. Esto puede verificarse con la cláusula *unify_with_occurs_check / 2*. Si no se utilizara el chequeo de ocurrencias, Prolog podría hacer una unificación infinita. Por lo tanto, sin el chequeo de ocurrencias, la cláusula $member(X, [f(X)])$ resultaría en $X = f(X)$, lo cual es un resultado erróneo.

1.3.

Una de las aplicaciones comentadas en el artículo es la de inferencia de tipos; es decir, la detección automática del tipo de datos en una expresión de un lenguaje de programación. La inferencia de tipos sirve también en lenguajes

de programación de diferentes paradigmas al lógico. La idea es prácticamente la misma pero en este caso la unificación se hace sobre expresiones de tipos; es decir, la substitución asigna tipos. Por ejemplo, si $[a]$ es una lista de enteros, entonces el proceso de unificación concluiría que a debe ser un entero.

1.4.

- *Monotonía:* La unificación se dice monotónica por su propiedad de combinar información. Aunque la noción de información aquí no es muy clara, la monotonía es intuitiva. Por ejemplo, si unificamos los términos $f(X)$ y $f(a)$ obrendríamos la substitución $X = a$. Sin embargo, el término $f(X)$ no es sobreescrito por la sustitución de la variable X; es decir la *estructura* anterior no es alterada.

- *Bidireccionalidad:* Esto se refiere a que no existe (a diferencia del *matching*) un sólo termino al cuál se aplica la substitución correspondiente de la unificación. Por ejemplo, unificar los términos $f(X, a)$ y $f(a, X)$ resulta en la substitución $X = a$, la cual es aplicada a **ambos** términos. En cambio, matching fallaría en este caso, pues la substitución sólo podría aplicarse a uno de los términos.

- *Conmutatividad y Asociatividad:* La comnutatividad se refiere a que no importa el orden en que se aplica la unificación. Por ejemplo unificar los términos $f(a, Y, c)$ y $f(X, b, c)$ da el mismo resultado de substitución $X = a$ y $Y = b$ si se toma $f(a, Y, c)$ como primer término y $f(X, b, c)$ como segundo o $f(X, b, c)$ como primero y $f(a, Y, c)$ como segundo. Por otro lado, la asociatividad indica que si se unifican tres términos, por ejemplo $f(X, b, c)$, $f(a, Y, c)$ y $f(a, b, Z)$, el resultado será el mismo si se unifican primero $f(X, b, c)$ y $f(a, Y, c)$, y luego $f(a, b, Z)$ o primero $f(a, Y, c)$ y $f(a, b, Z)$ y luego $f(X, b, c)$. En ambos casos, el resultado de la substitución es $X = a$, $Y = b$ y $Z = c$.

1.5.

- *Relevancia del Tema:* 10

- *Calidad Técnica:* 9

- *Redacción:* 10

- *Recomendación:* Recomendaría este artículo a alguien que o está familiarizado ya con el concepto de unificación y, específicamente, a alguien que esté interesado en tener una perspectiva muy general del tema. Me parece que si bien sirve como una revisión de la literatura alrededor del concepto de unificación, el artículo no profundiza en ciertos temas que podrían ser relevantes para alguien que está interesado en saber aspectos más específicos de la unificación. Además, el artículo se publicó en 1989, por lo

que puede ser un buen artículo para hablar de la historia y las bases de la unificación, pero quizá no sobre los desarrollos más recientes.

- *Posibles Mejoras:* Una mejora evidente sería la actualización del artículo para abarcar desarrollos más recientes en esta misma línea de investigación. Por otro lado, en mi opinión, la sección de otras aplicaciones es demasiado corta y pienso que falta desarrollarla o no incluirla como un apartado separado. Por último, aunque los algoritmos ilustrados en las figuras del artículo son relativamente claros, me parece que hace falta que haya comentarios más directos sobre ellos.

2. Pregunta 2

El algoritmo elegido para la implementación de la unificación fue el de Robinson [2], como en la Figura 1, con la modificación de omitir el *chequeo de ocurrencias*. Se basa en varios casos: el primero es cuando ambos términos en cuestión son constantes. En este caso, la unificación solamente es exitosa si las constantes son iguales, y, en dado caso, la substitución es vacía. El segundo es cuando al menos uno de los dos términos en cuestión es una *variable*. Bajo este caso la substitución correspondiente indica que el término variable debe tomar el valor del otro término en cuestión. El tercer y último caso es cuando ambos términos son *funtores*, de la forma $f(t_1, t_2, ..., t_n)$. Si para ambos términos el funtor es el mismo y dicho funtor tiene el mismo número de argumentos entonces se hace una llamada recursiva para unificar cada uno de sus argumentos. Si alguna de estas condiciones falla entonces la unificación falla.

```
function UNIFY(t1, t2) ⇒ (unifiable: Boolean, σ: substitution)
begin
  if t1 or t2 is a variable then
    begin
      let x be the variable, and let t be the other term
      if x = t, then (unifiable, σ) ← (true, ∅)
      else if occur(x, t) then unifiable ← false
      else(unifiable, σ) ← (true, {x ← t})
    end
  else
    begin
      assume t1 = f(x1, ..., xn) and t2 = g(y1, ..., ym)
      if f ≠ g or m ≠ n then unifiable ← false
      else
        begin
          k ← 0
          unifiable ← true
          σ ← nil
          while k < m and unifiable do
            begin
              k ← k + 1
              (unifiable, τ) ← UNIFY(σ(xk), σ(yk))
              if unifiable then σ ← compose(τ, σ)
            end
        end
    end
  return (unifiable, σ)
end
```

Figura 1: Pseudocódigo del Algoritmo de Robinson [2]

En los Cuadros 1 y 2 se encuentra la implementación en Prolog de este algoritmo.

```
%%% UNIFICACION

%%% Caso 1: ambos terminos son constantes.
%%% Esta clausula prueba si ambos terminos son constantes
%%% Tambien se revisa si hay igualdad entre los terminos.
unify(T1, T2) :-
    atomic(T1),
    atomic(T2),
    T1 == T2,
    !.

%%% Caso 2: al menos un termino es variable.
%%% En esta clausula se verifica si el primer termino es
%%% una variable. Tambien se verifica si hay igualdad
%%% entre terminos.
unify(T1, T2) :-
    var(T1),
    T1 == T2,
    !.

%%% En esta clausula se verifica si el primer termino es
%%% una variable. Tambien se verifica si hay una diferencia
%%% en los terminos, Si ambas condiciones se cumplen
%%% entonces el primer termino (variable) toma el valor del
%%% segundo
unify(T1, T2) :-
    var(T1),
    \+ (T1 == T2),
    T1 = T2,
    !.


%%% En esta clausula se verifica si el segundo termino es
%%% una variable. Tambien se verifica si hay una diferencia
%%% en los terminos, Si ambas condiciones se cumplen
%%% entonces el primer termino (variable) toma el valor del
%%% segundo
unify(T1, T2) :-
    var(T2),
    \+ (T1 == T2),
    T2 = T1,
    !.
```

Cuadro 1: Implementación del algoritmo de unificación en Prolog

```
%%% UNIFICACION

%%% Caso 3: ambos terminos son compuestos.
%%% En esta clausula se verifica que ambos terminos sean
%%% compuestos. Se obtiene el numero de argumentos y
%%% se utiliza este numero para unificar los argumentos.
unify(T1, T2) :-
    compound(T1),
    compound(T2),
    functor(T1, Functor, Numargs),
    functor(T2, Functor, Numargs),
    sub_unify(T1, T2, Numargs).

%%% Las siguientes dos clausulas se encargan de unificar
%%% los argumentos de los funtores.
%%% En el caso base cuando el numero de argumentos es 0,
%%% tenemos lo siguiente:
sub_unify(_, _, 0) :- !.

%%% En cualquier otro caso, se obtienen los ultimos argumentos
%%% del funtor y se unifican con 'unify'. Luego se hace una
%%% llamada recursiva a 'sub_unify' con el numero de argumentos
%%% menos uno.
sub_unify(T1, T2, Numargs) :-
    arg(Numargs, T1, Arg1),
    arg(Numargs, T2, Arg2),
    unify(Arg1, Arg2),
    NewNumargs is Numargs - 1,
    sub_unify(T1, T2, NewNumargs).
```

Cuadro 2: Implementación del algoritmo de unificación en Prolog

En el Cuadro 3 se muestran las ejecuciones de esta implementación para los ejemplos de la tarea.

```
%%% ?- unify(q(Y,g(a,b)),p(g(X,X),Y)).
%%% false.

%%% ?- unify(r(a,b,c),r(X,Y,Z)).
%%% X = a,
%%% Y = b,
%%% Z = c.

%%% ?- unify(mayor(padre(Y),Y),mayor(padre(Z),juan)).
%%% Y = Z, Z = juan.

%%% ?- unify(conoce(padre(X),X),conoce(W,W)).
%%% X = W, W = padre(W).
```

Cuadro 3: Ejecuciones del algoritmo.

3. Pregunta 3

En los Cuadros 4 y 5 tenemos el programa en Prolog para convertir un número de peano a su expresión decimal, para sumar dos números de peano, y para restar dos números de peano.

```
%%% PEANO

%%% Esta clausula convierte numeros de peano
%%% de la forma s(s(...(0))) en su expresion
%%% decimal. La primera parte es una 'interfaz'
%%% para hacer llamada a la clausula que es
%%% recursiva a la cola 'peanoToNatAux'.
%%% Esta ultima suma 1 por cada argumento
%%% del numero de peano.

peanoToNat(P, N) :-
    peanoToNatAux(P, N, 0).

peanoToNatAux(0, Acc, Acc) :- !.

peanoToNatAux(P, N, Acc) :-
    AccNew is Acc + 1,
    arg(1, P, Arg),
    peanoToNatAux(Arg, N, AccNew).

%%% Esta clausula suma dos numeros de peano.
%%% Lo primero que sea hace es obtener la
%%% expresion decimal del segundo termino y
%%% recursivamente aplicar la clausula de
%%% sucesion al primer termino las veces
%%% correspondientes al valor del segundo
%%% numero.

sumaPeano(P1, P2, R) :-
    peanoToNat(P2, N),
    sumaNPeano(P1, N, R).

sumaNPeano(P, 0, P) :- !.

sumaNPeano(P, N, R) :-
    Nnew is N - 1,
    sumaNPeano(s(P), Nnew, R).
```

Cuadro 4: Programa de Prolog para números de Peano.

```
%%% PEANO

%%% Esta clausula resta dos numeros de peano.
%%% Lo primero que sea hace es obtener la
%%% expresion decimal del segundo termino y
%%% tambien la del primero. Si el segundo
%%% es menor o igual que el primero entonces
%%% recursivamente se obtiene el primer
%%% argumento del primer termino las veces
%%% correspondientes al valor del segundo
%%% numero.

restaPeano(P1, P2, R) :-
    peanoToNat(P2, N2),
    peanoToNat(P1, N1),
    N2 =< N1,
    restaNPeano(P1, N2, R).

restaNPeano(P, 0, P) :- !.

restaNPeano(P, N, R) :-
    Nnew is N - 1,
    arg(1, P, Arg),
    restaNPeano(Arg, Nnew, R).
```

Cuadro 5: Programa de Prolog para números de Peano.

A continuación, en el Cuadro 6 se muestran las ejecuciones de los ejemplos de la tarea con la implementación de los Cuadros 4 y 5.

```
%%% ?- peanoToNat(s(s(s(0))), N).
%%% N = 3.

%%% ?- peanoToNat(0, N).
%%% N = 0.

%%% ?- sumaPeano(s(s(0)), s(0), R).
%%% R = s(s(s(0))).

%%% ?- restaPeano(s(s(0)), s(0), R).
%%% R = s(0).
```

Cuadro 6: Ejecuciones de ejemplos de Peano.

4. Pregunta 4

En los cuadros 7 y 8 tenemos la implementación en Prolog de las operaciones sobre listas indicadas en la tarea con los correspondientes comentarios.

```
%%% SETS


%%% Esta clausula verifica si una lista es subconjunto
%%% de otra. Para esto revisa, recursivamente, si cada
%%% elemento de la primera lista se contiene en la otra.
subset([], _L2) :- !.

subset([Head|Tail], L2) :-
    member(Head, L2),
    subset(Tail, L2),
    !.

%%% Esta clausula regresa la interseccion entre dos listas.
%%% Para los casos en los cuales cualquiera de las dos
%%% listas es vacia la interseccion es vacia tambien
%%% En cualquier otro caso, se llama la clausula
%%% 'inter_aux' que es recursiva a la cola y que acumula
%%% los elementos de la primera lista que estan tambien
%%% en la segunda.
inter([], _, []) :- !.

inter(_, [], []) :- !.

inter(L1, L2, R) :-
    inter_aux(L1, L2, R_aux, []),
    reverse(R_aux, R).

inter_aux([], _, Acc, Acc) :- !.

inter_aux([Head|Tail], L2, R, Acc) :-
    member(Head, L2),
    append([Head], Acc, AccNew),
    inter_aux(Tail, L2, R, AccNew),
    !.

inter_aux([Head|Tail], L2, R, Acc) :-
    not(member(Head,L2)),
    inter_aux(Tail, L2, R, Acc),
    !.
```

Cuadro 7: Implementacion en Prolog de operaciones de listas

```
%%% SETS

%%% Esta clausula regresa la union entre dos listas.
%%% Si alguna de las dos listas es vacia entonces
%%% regresa la otra lista. En cualquier otro caso se
%%% llama a la clausula 'union_aux' que agrega a la
%%% segunda lista todos los elementos de la primera
%%% que no estan ya en la segunda.

union([], L2, L2) :- !.

union(L1, [], L1) :- !.

union(L1, L2, R) :-
    union_aux(L1, L2, R, L2).

union_aux([], _, Acc, Acc) :- !.

union_aux([Head|Tail], L2, R, Acc) :-
    not(member(Head, L2)),
    append([Head], Acc, AccNew),
    union_aux(Tail, L2, R, AccNew),
    !.

union_aux([Head|Tail], L2, R, Acc) :-
    member(Head, L2),
    union_aux(Tail, L2, R, Acc),
    !.

%%% Esta clausula regresa la diferencia entre dos
%%% listas. Para los casos en los cuales la primera
%%% lista es vacia, tenemos que la diferencia es vacia
%%% para el caso en el cual la segunda es vacia, tenemos
%%% que la diferencia es la primera lista. En cualquier
%%% otro caso se calcula la interseccion y se llama a la
%%% clausula 'dif_aux' que es recursiva a la cola y
%%% crea una nueva lista con todos los elementos que
%%% estan en la primera lista pero no estan en la
%%% interseccion.

dif([], _L2, []) :- !.

dif(L1, [], L1) :- !.

dif(L1, L2, R) :-
    inter(L1, L2, NewL2),
    dif_aux(L1, NewL2, R, []).

dif_aux([], _, Acc, Acc) :- !.

dif_aux([Head|Tail], L2, R, Acc) :-
    not(member(Head, L2)),
    append([Head], Acc, AccNew),
    dif_aux(Tail, L2, R, AccNew),
    !.

dif_aux([Head|Tail], L2, R, Acc) :-
    member(Head, L2),
    dif_aux(Tail, L2, R, Acc),
    !.
```

Cuadro 8: Implementacion en Prolog de operaciones de listas

En el Cuadro 9 encontramos la ejecución de las operaciones de los Cuadros 7 y 8 que se encuentran en la tarea.

```
%%% ?- subset([1,3],[1,2,3,4]).
%%% true.

%%% ?- subset([],[1,2]).
%%% true.

%%% ?- inter([1,2,3],[2,3,4], R).
%%% R = [2, 3].

%%% ?- union([1,2,3,4], [2,3,4,5], R).
%%% R = [1, 2, 3, 4, 5].

%%% ?- dif([1,2,3,4], [2,3,4,5], R).
%%% R = [1].

%%% ?- dif([1,2,3], [1,4,5], R).
%%% R = [3, 2].
```

Cuadro 9: Ejecución de operaciones de lista

5. Pregunta 5

En el Cuadro 10 encontramos la implementación de un algoritmo en Prolog que genera todas las permutaciones de una lista.

```
%%% PERMUTATIONS

%%% Esta clausula utiliza 'findall' para encontrar
%%% todas los resultados de aplicar la clausula
%%% 'permutation' a la lista de entrada L.

permute(L,R) :-
    findall(P, permutation(L,P), R).
```

Cuadro 10: Implementación en Prolog de algoritmo para encontrar todas las permutaciones de una lista.

Finalmente, en el Cuadro 11 tenemos una ejecución del código del Cuadro 10.

```
%%% PERMUTATIONS

%%% ?- permute([1,2,3], R).
%%% R = [[1, 2, 3], [1, 3, 2], [2, 1, 3], [2, 3, 1], [3, 1, 2], [3, 2, 1]].
```

Cuadro 11: Ejecución de algoritmo para permutar lista.

Referencias

[1] K. Knight, *Unification: a multidisciplinary survey.* ACM Comput. Surv., 21(1):93–124, 1989.

[2] J. A. Robinson, *A machine-oriented logic based on the resolution principle*. ACM 12, pp. 23-41, 1965.