Más orden superior

Recordemos los predicados de orden superior que vimos hasta el momento, esos “predicados que relacionan predicados”:

- **not/1**
- **findall/3**
- **forall/2**

¿Eso es todo lo que hay?

No, por supuesto que no. Existen muchos otros predicados de orden superior "pre-construidos" (built-in), pero esta base nos alcanza para lo que queremos ver en la materia.

Sin embargo, es interesante saber que podemos construir nuestros propios predicados de orden superior.

call / 1

El predicado `call/1` nos permite evaluar un predicado pasado por parámetro. Retomando uno de nuestros primeros ejemplos, veamos cómo se usaría:

```
?- call(padre(Padre, Hijo)).
Padre = homero
Hijo = bart ;
...
```

Pero eso no aporta mucho respecto de hacerlo en forma directa:

```
?- padre(Padre, Hijo).
Padre = homero
Hijo = bart ;
...
```

Para sacarle verdadero provecho a este predicado deberíamos usar algunas cosas que quedan fuera del alcance de la materia[1]. Veamos otra variante, entonces, que puede resultarnos más interesante y útil para lo que sí podemos llegar a usar.

call / _

El predicado `call/_` también nos permite evaluar un predicado pasado por parámetro, pero separando los parámetros que el mismo recibe:

```
?- call(padre, Padre, Hijo).
Padre = homero
Hijo = bart ;
...
```

O también:

```
?- call(padre(homero), Hijo).
Hijo = bart ;
...
```

[1] En particular, puede servirnos el operador “**=..**” (igual-punto-punto, no está mal escrito). Con este operador y usando `call/1` podemos implementar algo como `call/_`. Pero, insistimos, va más allá del alcance de la materia.

Momento, momento... entonces, ¿cuál es la aridad de `call/_`?
El predicado `call/_` no tiene definida una aridad fija. Puede tener desde 1 (la versión que vimos antes) hasta N + 1, siendo N la aridad del predicado que se recibe como primer parámetro.

Usando este predicado podemos hacer muchas otras cosas. Juguemos un poco con esto: implementemos predicados que nos permitan mapear o filtrar listas, que por una cuestión filosófica[2] los vamos a llamar maplist e include respectivamente.

Creando nuestros propios predicados de orden superior

maplist

Queremos un predicado que mapee una lista, es decir, un predicado que en base a otro predicado que me relacione un elemento de la lista con "otra cosa", nos permita obtener una nueva lista con los "otra cosa" correspondientes a cada elemento de la lista. Empecemos por lo básico… ¿cuántos parámetros va a tener?

Vamos a tener el predicado de "transformación" y la lista original, por supuesto. Pero también necesitamos un argumento más para unificarlo con la lista resultante del mapeo. Tenemos también que considerar las cosas que relaciona el predicado: un elemento de la lista original con uno de la lista resultante. La consulta tendría esta forma:

```
?- map(Predicado, ListaOriginal, ListaResultante).
```

Y un ejemplo de uso sería:

```
?- map(padre, [bart, homero], Padres).
Padres = [homero, abraham]
```

Ok, vamos a la implementación entonces. Vamos a ver dos formas de implementarlo:

```
mapRecursivo( _ , [] , [] ).
mapRecursivo(Pred, [X|Xs], [Y|Ys]):-
        call(Pred, X, Y),
        mapRecursivo(Pred, Xs, Ys).
```

También podríamos hacer una versión no recursiva:

```
mapNoRecursivo(Pred, ListaOriginal, ListaResultante):-
        findall(Y,
                (member(X, ListaOriginal), call(Pred, X, Y)),
                ListaResultante).
```

Ejemplos de consulta:

```
?- mapRecursivo(padre, [herbert, homero], Hijos).
Hijos = [homero, bart] ;
Hijos = [homero, lisa] ;
Hijos = [homero, maggie] ;
false

?- mapNoRecursivo(padre, [herbert, homero], Hijos).
Hijos = [homero, bart, lisa, maggie] ;
false
```

[2] La cuestión filosófica es que **ya existen** como predicados built-in, como veremos más adelante. El tema es que sólo queremos ver cuál sería su implementación para ver cómo podemos aplicar `call/_` para construir nuestros propios predicados de orden superior.

¡Epa! No son iguales... pero, ¿cuál está bien y cuál está mal?

Vemos cómo con nuestra implementación recursiva obtenemos N respuestas, todas las combinaciones posibles de mapeo, pero siempre con mapeos 1 a 1 para cada elemento de la lista original[3].

En cambio, para nuestra implementación no recursiva la respuesta es única.

¿Cuál está bien? la respuesta, como muchas veces, es "depende"... depende de lo que necesitemos.

Veamos algunos ejemplos más:

```
mapRecursivo(padre, Padres, [bart, lisa, maggie]).
Padres = [homero, homero, homero] ;
false
```

Nuestra versión recursiva es inversible para el segundo o el tercer argumento (aunque no ambos simultáneamente). Si probamos lo mismo con nuestra versión no recursiva, nos vamos a encontrar con un problema.

Recordemos que en la implementación estamos usando `member/2` con la primera lista, y `member/2` no es inversible para la lista.

Bueno, la versión built-in de map en SWI-Prolog es **`maplist/3`**, y se comporta como nuestra versión ***recursiva***.

include

Este predicado me permite *filtrar* una lista en base a un criterio para obtener una nueva lista. Hagamos también las dos versiones, una recursiva y otra en base a `findall/3`.

Versión recursiva:

```
filterRecursivo( _ , [] , [] ).
filterRecursivo(Pred, [X | Xs], [X | Ys]):-
       call(Pred, X),
       filterRecursivo(Pred, Xs, Ys).
filterRecursivo(Pred, [X | Xs], Ys):-
       not(call(Pred, X)),
       filterRecursivo(Pred, Xs, Ys).
```

Versión no recursiva:

```
filterNoRecursivo(Pred, ListaOrigen, ListaResultante):-
       findall(X,
              (member(X, ListaOrigen), call(Pred, X)),
              ListaResultante).
```

Ejemplos de consulta:

```
?- filterRecursivo(padre(homero), [herbert, lisa, maggie, homero, bart], ListaFiltrada).
ListaFiltrada = [lisa, maggie, bart] ;
false
```

[3] Más adelante, para la versión recursiva puede que consideremos que “lista original” y “lista resultante” no sean los nombres más apropiados para los parámetros. Esto es así si tenemos en cuenta que el predicado es inversible, pero considerando cómo lo estamos planteando inicialmente, podemos decir que sí son nombres apropiados.

```
?- filterNoRecursivo(padre(homero), [herbert, lisa, maggie, homero, bart], ListaFiltrada).
ListaFiltrada = [lisa, maggie, bart] ;
false
```

OK, acá no hay diferencia, como sí pasaba en el caso de map. Este predicado también existe como *built-in* y, como el título lo dice, es **`include/3`**.

¿Cómo seguimos?

De la misma forma que armamos los predicados anteriores, podríamos armar algunos predicados algo más complejos, como saber si una lista está ordenada por un criterio dado. Probemos algo de esto... empecemos por ese "está ordenado por":

```
estaOrdenadaPor([ _ ], _ ).                  /*Una lista con un único elemento siempre está ordenada       */
estaOrdenadaPor([ X , Y | Cola ],Criterio):- /*Una lista con 2 o más elementos, ordenada por Criterio      */
    call(Criterio, X, Y),                    /*cumple ese criterio entre sus 2 primeros elementos         */
    estaOrdenadaPor(Pred, [ Y | Cola ]).     /*y está ordenada por él también a partir del segundo elemento*/
```

Vamos a complicarla... Para hacer una sumatoria de una lista, por ejemplo, en lugar de la forma que vimos antes puedo querer *plegar*[4] la lista, es decir, tomar cada elemento y sumarlo al total de los anteriores. Este "pliegue" es de izquierda a derecha, porque empezamos con la cabeza de la lista, luego el segundo elemento, el tercero, etc. Necesita un valor de partida, ya que "la suma de los anteriores" no es algo válido si estamos trabajando con el primer elemento.
Ejemplo, para la lista [1, 2, 3] queremos hacer (((0 + 1) + 2) + 3). Cada paréntesis es la operación sobre un elemento de la lista, y el 0 es el valor de partida.
Esto, siempre y cuando estemos hablando de una suma. Nosotros lo vamos a hacer de forma genérica para cualquier operación.

```
plegarIzquierdo( _ , Valor, [] , Valor).
plegarIzquierdo(Pred, Valor, [X|Xs], Resultado):-
       call(Pred, Valor, X, ValorIntermedio),
       plegarIzquierdo(Pred, ValorIntermedio, Xs, Resultado).
```

Para hacerlo con la suma, podemos crear el predicado `sumar(X,Y,Suma)` y hacer la consulta:

```
?- plegarIzquierdo( sumar , 0, [1, 2, 3] , Suma).
Suma = 6 .
```

En la suma no habría diferencia pero, para otras operaciones, nos puede interesar hacerlo de derecha a izquierda. Para la suma, significaría "sumar cada elemento a la suma de los siguientes", y como no hay un siguiente para el último elemento también necesito un valor de partida.
El mismo ejemplo, para la lista [1, 2, 3], equivaldría a hacer (1 + (2 + (3 + 0))). Nuevamente, cada paréntesis es la operación sobre un elemento de la lista, y el 0 es el valor inicial. Ahora, para hacer la suma del 1 necesito ya conocer la suma del resto, por lo que tenemos que cambiar el orden de las subconsultas:

```
plegarDerecho( _ , Valor, [] , Valor).
plegarDerecho(Pred, Valor, [X|Xs], Resultado):-
       plegarDerecho(Pred, Valor, Xs, ValorIntermedio),
       call(Pred, X, ValorIntermedio, Resultado).
```

[4] Viene del término que se usa en inglés para este tipo de operaciones: "fold". Las mismas operaciones que estamos creando acá las vamos a ver más adelante en funcional, con las funciones `fold*` (hay varias versiones).

Pero no tenemos que olvidarnos de no reinventar la rueda cada vez... ¿qué pasa si queremos el mejor elemento de una lista según un criterio dado (máximo, mínimo u otros)? Podríamos usar `call/_` y recursividad o `forall/3`. Por ejemplo, con `forall/3`:

```
mejorSegun(Pred, Lista, X):-         /* X es el mejor de la Lista según Pred, si   */
    member(X, Lista),                /* X es miembro de la lista                   */
    call(Pred, X, ValorX),           /* Pred relaciona a X con un ValorX           */
    forall(member(Y, Lista),         /* y para todo Y, también miembro de la Lista */
           (call(Pred, Y, ValorY),   /* Pred relaciona a Y con un ValorY           */
            ValorY =< ValorX)).      /* y ValorY es menor o igual a ValorX         */
```

Pero también podemos usar `plegarIzquierdo/4` (o bien `plegarDerecho/4`), que definimos recién. Lo único que necesitamos es la operación adecuada. Antes era sumar/3, ahora sería un predicado que relacione un X y un Y con el mejor de ambos... ¡según el criterio que querramos! Es la gran ventaja de haberlos hecho genéricos.

```
mejorSegun(Pred, [X|Xs], Mejor):- plegarIzquierdo(Pred, X, Xs, Mejor).
```

Lo mismo nos pasa con `maplist/3`, `include/3`, `findall/3`, etc. Ya los tenemos de antemano, ¡está bueno tenerlos presentes! Pero, de paso, notemos que no por no decir "`call`" en algún lugar nuestro predicado deja de ser un predicado de orden superior. Seguimos teniendo "predicados que relacionan predicados" como dijimos al principio.