Más orden superior

Pero... en Haskell podía hacer más cosas...

Recordemos los predicados de orden superior que vimos hasta el momento:

- **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.

Pero lo más interesante es que, como hicimos en Haskell, 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 un ejemplo:

```
?- 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).
```

[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.

```
Hijo = bart ;
...
```

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 cosas equivalentes (¡no iguales!) a las que hacíamos con orden superior en Haskell. Juguemos un poco con esto: implementemos map y filter, aunque por una cuestión filosófica[2] los vamos a llamar maplist e include respectivamente.

Creando nuestros propios predicados de orden superior

maplist (el viejo y querido map)

Empecemos por lo básico… ¿cuántos parámetros tenía la ***función*** map?

```
> map f lista
```

Tenía dos parámetros, una función de transformación "f" y una lista, y la función era aplicable a cada elemento de la lista.

Entonces, ¿cuántos argumentos va a tener nuestra ***relación*** map? Vamos a tener el predicado de transformación y la lista, 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.

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

Ejemplo de uso:

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

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

```
mapRecursivo(Pred, [], []).
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:

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

```
?- mapRecursivo(padre, [herbert, homero], Hijos).
Hijos = [homero, bart] ;
Hijos = [homero, lisa] ;
Hijos = [homero, maggie] ;
No

?- mapNoRecursivo(padre, [herbert, homero], Hijos).
Hijos = [homero, bart, lisa, maggie] ;
No
```

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

Vemos como nuestra implementación recursiva nos da N respuestas, todas las combinaciones posibles de mapeo, pero siempre con mapeos 1 a 1 para cada elemento de la lista original.

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] ;
No
```

Nuestra versión recursiva es inversible para el segundo o el tercer argumento (no ambos). 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 (el viejo y querido filter)

La cuestión de los parámetros es igual a la anterior: vamos a necesitar uno más que lo que tenía filter en Haskell para unificar la lista resultante. Hagamos también dos versiones.

Versión recursiva:

```
filterRecursivo(Pred, [], []).
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] ;
No

?- filterNoRecursivo(padre(homero), [herbert, lisa, maggie, homero, bart], ListaFiltrada).
ListaFiltrada = [lisa, maggie, bart] ;
No
```

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 predicados equivalentes a map y filter, podríamos armar un predicado equivalentes a fold, o saber si una lista está ordenada por un criterio dado. Probemos algo de esto... el más sencillo es “está ordenado por”. Veamos:

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

Para fold, volvamos a pensar un poco en los parámetros que recibía: la función, un valor inicial y la lista... bueno, también necesitamos agregarle un parámetro, como a los anteriores, quedándonos una relación entre 4 cosas. Acá podemos ver 2 implementaciones, equivalentes a foldl y foldr de lo que teníamos en funcional.

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

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

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, pero también podemos usar `foldl/4` (o bien `foldr/4`), que definimos recién:

```
mejorSegun(Pred,[X|Xs],M):- foldl(Pred,X,Xs,M).
```

Lo mismo nos pasa con `maplist/3`, `include/3`, `forall/2`, `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 dejamos de tener un predicado de orden superior. Seguimos teniendo “predicados que relacionan predicados” como dijimos al principio.