Paradigma Lógico

Módulo 7:
Explosión combinatoria.
Orden superior (definición).
Efecto colateral.

por Fernando Dodino
Matías Freyre

Versión 2.0
Diciembre 2016

Contenido

1 Explosión combinatoria

1.1 Ejemplo Actividades

En la base de conocimientos se registraron distintas salidas o actividades y su respectivo costo:

```
actividad(cine).                costo(cine, 400).
actividad(arjona).              costo(arjona, 1750).
actividad(princesas_on_ice).    costo(princesas_on_ice, 2500).
actividad(pool).                costo(pool, 350).
actividad(bowling).             costo(bowling, 300).
```

Se pide definir el predicado
`actividades(Plata, ActividadesPosibles)`
que relacione una determinada cantidad de plata con la combinación de actividades que podemos hacer.

```
?- actividades(2350, Actividades).
Actividades = [cine, arjona] ;
Actividades = [cine, pool, bowling] ;
Actividades = [cine, pool] ;
Actividades = [cine, bowling] ;
Actividades = [cine] ;
Actividades = [arjona, pool] ;
Actividades = [arjona, bowling] ;
Actividades = [arjona] ;
Actividades = [pool, bowling] ;
Actividades = [pool] ;
Actividades = [bowling] ;
Actividades = [] ;  % => me quedo en casa y no gasto nada
```

Cada actividad está definida en predicados individuales, para poder trabajar la lista de actividades posibles necesitamos utilizar el predicado findall/3:

```
actividades(Plata, ActividadesPosibles):-
   findall(Actividad, actividad(Actividad), Actividades),
   actividadesPosibles(Actividades, Plata, ActividadesPosibles).
```

Ahora escribiremos el predicado actividadesPosibles/3. Tenemos 3 alternativas:

Caso 1 | Base: Cuando no tengo actividades, no hay actividades posibles, no importa cuánta plata tenga:

```
actividadesPosibles([] , _ , []).
```

Caso 2 | Recursivo: Una actividad posible va a formar parte del conjunto solución si tengo plata suficiente. El resto de las actividades posibles se relacionará con la plata que me quede tras hacer esa actividad:

```
actividadesPosibles([Actividad|Actividades], Plata,
[Actividad|Posibles]):-
      costo(Actividad, Costo),
      Plata > Costo,
      PlataRestante is Plata - Costo,
      actividadesPosibles(Actividades, PlataRestante, Posibles).
```

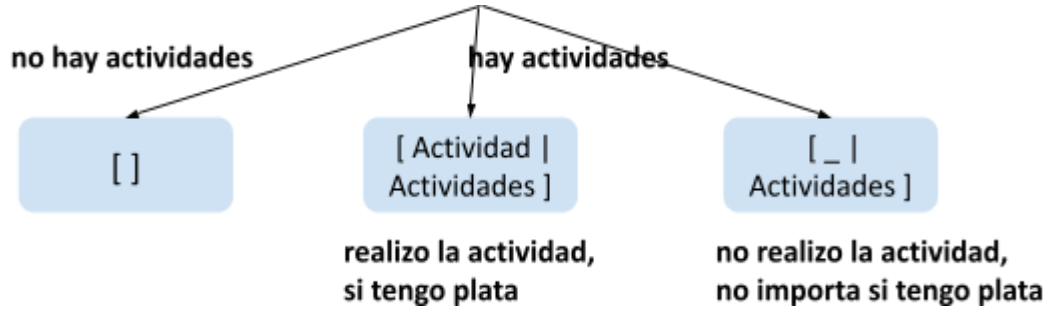
Caso 3 | Recursivo: Si decido no hacer esa actividad, tendré la misma plata para las actividades restantes:

```
actividadesPosibles([_|Actividades], Plata, Posibles):-
      actividadesPosibles(Actividades, Plata, Posibles).
```

En la última cláusula puede pasar que no haga la actividad porque:

1) no tengo plata para hacer esa actividad, o bien
2) tengo plata para hacer esa actividad pero no me interesa.

Si revisamos el pattern matching de las tres definiciones:



El pattern matching por el caso base es excluyente respecto a los otros dos casos: hay o no hay actividades. Ahora bien, si hay actividades Prolog considera ambos patrones coincidentes. Solo descartará las soluciones en las que no hay suficiente plata para hacer la actividad.

El efecto que tiene es que hace una explosión combinatoria con todas las soluciones posibles, siempre que no me pase del presupuesto. Eso incluye:

ahorrarme la plata y quedarme en casa o ir solo al cine y no gastar “tanto”, ir al cine y a ver a Arjona (y gastar más pero tener más salidas), etc. Todos esos criterios se combinan para dar el árbol de soluciones posible.

1.2 Ejemplo: Barco pirata

Tenemos esta base de conocimientos:

```
cocinero(donato).
cocinero(pietro).
pirata(felipe, 27).
pirata(marcos, 39).
pirata(facundo, 45).
pirata(tomas, 20).
pirata(betina, 26).
pirata(gonzalo, 22).

bravo(tomas).
bravo(felipe).
bravo(marcos).
bravo(betina).
personasPosibles([donato, pietro, felipe,
marcos, facundo, tomas, gonzalo, betina]).
```

Un pirata quiere armar la tripulación para su barco, él solo quiere llevar

- cocineros
- piratas bravos
- piratas de más de 40 años (no pagan impuestos)

```
tripulacionBarco(Tripulacion):-
   personasPosibles(Personas),
   tripulacion(Personas, Tripulacion).
```

Abstraemos en un predicado auxiliar qué personas pueden subir al barco. Los tres casos son similares al ejemplo de las actividades:

- cuando no hay personas, nadie más sube al barco
- puedo subir al barco a una persona si cumple el criterio del pirata
- o puedo decidir que no suba, porque no cumple las condiciones o “porque sí”

```
tripulacion([], []).
tripulacion([Posible|Posibles], [Posible|Tripulantes]):-
     puedeSubirAlBarco(Posible),
     tripulacion(Posibles, Tripulantes).
tripulacion([_|Posibles], Tripulantes):-
     tripulacion(Posibles, Tripulantes).

puedeSubirAlBarco(Persona):-cocinero(Persona).
puedeSubirAlBarco(Persona):-pirata(Persona, _), bravo(Persona).
puedeSubirAlBarco(Persona):-pirata(Persona, Edad), Edad > 40.
```

El predicado que generamos es inversible:

```
? tripulacionBarco(Quienes).
Quienes = [donato, pietro, felipe, marcos, facundo, tomas, betina]
Quienes = [donato, pietro, felipe, marcos, facundo, tomas]
Quienes = [donato, pietro, felipe, marcos, facundo, betina]
Quienes = [donato, pietro, felipe, marcos, facundo]
...
```

Puedo no llevar a nadie si yo quiero, pero gonzalo nunca formará parte de la tripulación, porque no es un pirata bravo ni tiene más de 40 años.

1.3 Barco pirata con restricciones generales

Al ejemplo anterior le agregamos una restricción adicional: la tripulación del barco debe tener 10 ó más grados de ferocidad, donde

- la ferocidad de un cocinero es 2
- la ferocidad de un pirata bravo es 5

Definimos el predicado ferocidad/2:

```
ferocidad(Persona, 2):-cocinero(Persona).
ferocidad(Persona, 5):-pirata(Persona, _), bravo(Persona).
```
[1]

Ahora, hasta tanto no ligue la tripulación del barco, no puedo saber cuánto suma su grado de ferocidad. Entonces la primera parte de tripulacionBarco/1 queda igual:

```
tripulacionBarco(Tripulacion):-personasPosibles(Personas),
   tripulacion(Personas, Tripulacion), ...
```

solo que ahora sí vamos a relacionar la tripulación con su ferocidad, para eso contamos con el predicado maplist/3:

```
tripulacionBarco(Tripulacion):-
   personasPosibles(Personas),
   tripulacion(Personas, Tripulacion),
   maplist(ferocidad, Tripulacion, Ferocidades),
   sumlist(Ferocidades, GradoFerocidad),
   GradoFerocidad > 10.
```

[1] Así como definimos el predicado, una persona que no tiene ferocidad falla en lugar de relacionarse con 0 grados de ferocidad. Una forma de resolverlo mejor podría ser agregando: `ferocidad(Persona, 0):-pirata(Persona, _), not(bravo(Persona)).`

maplist/3 es equivalente a la función map/2 que vimos en el paradigma funcional:

```
? maplist(ferocidad, [betina, pietro, tomas], Ferocidades).
Ferocidades = [5, 2, 5]
```

Al hacer la consulta, vemos que tanto `[felipe, tomas, betina]` como `[felipe, marcos, tomas]` son soluciones válidas, aunque `[felipe, tomas]` no suma más de 10 grados de ferocidad (justo 10), ni la lista vacía (que no tiene ferocidad).

2 mapList/3

mapList relaciona

- un predicado de aridad n (donde n > 1) ,
- una lista
- y otra lista que resulta de evaluar el predicado

```
?- maplist(length, [[1], [2, 6], []],  Transf).
Transf = [1, 2, 0]

?- maplist(succ, [5, 6, 7], Transf).
Transf = [6, 7, 8]
```

La asociación con la función map vista en Haskell es evidente:

```
map :: (a -> b) -> [a] -> [b]
```

La función de transformación va de a en b. Veamos qué predicados enviamos como parámetro en los ejemplos del maplist de Prolog: length/2 y succ/2.

2.1 Predicados que puede recibir maplist

¿Qué restricciones de tipo aplican sobre el predicado que enviamos como parámetro? Bueno, tiene sentido que cada elemento de la lista sea “adecuado” para dicho predicado. En el caso de length, esperamos que cada elemento sea una lista. El Prolog que usamos no hace chequeos estáticos, sin embargo cuando hacemos:

```
?- maplist(length, [[2], 8], X).
ERROR: length/2: Type error: `list' expected, found `8'
```

Hay un pequeño control que determina que length(8, X) no puede satisfacerse con un valor que no es una lista. El lector puede advertir cierta similitud entre Haskell y Prolog:

```
> map length [[2], 8]
```

esa expresión no pueda compilarse:

```
ERROR - Illegal Haskell 98 class constraint in inferred type
*** Expression : map length [[2],8]
*** Type       : (Num a, Num [a]) => [Int]
```

También podemos utilizar como predicado una función, como abs:

```
?- maplist(abs, [5, -6, 7], X).
X = [5, 6, 7]


?- maplist(sign, [5, -6, 7], X).
X = [1, -1, 1]
```

Y si evaluamos:

```
?- maplist(plus(1), [5, 6, 7], Result).
Result = [6, 7, 8]
```

Este ejemplo nos permite observar que, si bien plus es un predicado de aridad 3, en maplist se puede fijar el primer parámetro para dejar libres los últimos dos: el primer argumento libre se asocia a cada elemento de la lista y el otro argumento se relaciona con el resultado final.

El corolario es que estamos usando un concepto ya conocido del paradigma funcional: la aplicación parcial:

```
> map (+ 1) [5, 6, 7]
[6, 7, 8]
```

El lector recordará las ventajas de no tener que definir un predicado auxiliar sumarUno/2, y directamente para ello aprovecharse de la definición original de plus/3.

2.2 Inversibilidad

Como hemos visto antes, en el paradigma funcional no existe el concepto de inversibilidad. Resulta interesante pensar qué argumentos de maplist podríamos dejar libres.

Como el primer argumento es el predicado de transformación, requiere cierto trabajo generar el conjunto de predicados built-in + los desarrollados por nosotros. Vamos a pensar que el predicado es obligatorio instanciarlo.

Pero podemos ver si Prolog es capaz de relacionar la lista original en base a la lista transformada:

```
?- maplist(succ, Original, [2, 8, 3]).
Original = [1, 7, 2]


?- maplist(length, Original, [2, 1]).
Original = [[_G400, _G403], [_G409]]
```

Interesante, ¿eh? Prolog nos pregunta si quiere buscar más soluciones: con lo cual podemos abrir el juego de maplist para pensarlo como una relación, en lugar de una simple función de transformación.

Y qué pasa cuando queremos hacer:

```
?- maplist(abs, X, [4, 2, 1]).
ERROR: Arguments are not sufficiently instantiated
```

Ah... entonces, la inversibilidad ya no depende exclusivamente de los parámetros, sino que también necesitamos que el predicado sea inversible en los argumentos que nosotros queremos.

2.3 Múltiples soluciones

El predicado between permite dejar el último argumento sin instanciar:

```
?- between(3, 5, X).
X = 3 ;
X = 4 ;
X = 5 ;
```

Entonces podemos utilizar mapList para generar la explosión combinatoria de los números que están entre 3 y 5 y entre 3 y 6

```
?- maplist(between(3), [5, 6], X).

 X = [3, 3] ;          X = [4, 3] ;          X = [5, 3] ;
 X = [3, 4] ;          X = [4, 4] ;          X = [5, 4] ;
 X = [3, 5] ;          X = [4, 5] ;          X = [5, 5] ;
 X = [3, 6] ;          X = [4, 6] ;          X = [5, 6] ;
```

Hacer lo mismo en Haskell requiere cambiar la forma en que resuelve los cálculos el motor (necesita de conceptos no triviales como las mónadas), por lo que podemos describirla como una característica diferencial entre el map de Haskell y el maplist de Prolog.

3 Definiendo predicados de orden superior

De la misma manera que en Haskell podíamos aplicar una función recibida como parámetro

```
map f xs = [ f x | x <- xs ]
```

en Prolog contamos con el predicado call/1 o call/_ que permite evaluar un predicado recibido como parámetro:

```
? call(hijo(Padre, Hijo)).
Padre = homero
Hijo = maggie ;
...

?- call(hijo(Padre, bart)).
Padre = homero

?- call(plus(1), 2, X).     % call/_ permite separar la consulta
                            % en múltiples argumentos
X = 3 ;
```

Esto permite subir el grado de abstracción de un predicado:

```
q(X):-p(X).                 % predicado de primer orden
q(X, P):-call(P, X).        % predicado de orden superior
```

3.1 Nuestro propio maplist

Pensemos cómo codificar el predicado maplist a partir de estas definiciones:

```
maplist(_, [], []).
maplist(PredicadoTransformador, [Orig|Origs], [Transf|Transfs]):-
     call(PredicadoTransformador, Orig, Transf),
     maplist(PredicadoTransformador, Origs, Transfs).
```

3.2 Nuestro propio filter

Podemos pensar en un predicado que relacione los elementos originales de una lista con los que cumplen un determinado criterio. La primera versión la resolvemos con findall:

```
even(Numero):-0 is Numero rem 2.

filter(Criterio, ListaOriginal, ListaNueva):-
   findall(Elem,
           (member(Elem, ListaOriginal), call(Criterio, Elem)),
           ListaNueva).
```

Esto nos permite hacer la siguiente consulta:

```
?- filter(even, [1, 5, 2, 3, 4, 7], X).
X = [2, 4] ;
```

Tengamos en cuenta que el predicado Criterio debe ser de aridad 1, debido a la forma en que lo usamos con call.

Existe una versión built-in de este filter que hicimos, y es **include/3**.

El predicado filter no es inversible para los primeros dos argumentos, dado que no podemos satisfacer call(Criterio, Elem) dado member(Elem, Original) sin conocer la lista original.

De hecho, el maplist podríamos haberlo definido en función al findall

```
maplistF(PredTransf, Original, Nueva):-
   findall(Result, (member(Elem, Original),
                    call(PredTransf, Elem, Result)), Nueva)
```

Pero eso no nos permite pedir consultas como ésta:

```
?- maplist2(plus(1), X, [5, 6, 7]).
ERROR: succ/2: Arguments are not sufficiently instantiated
```

por el mismo motivo que en la definición previa de filter.

Ahora... ¿qué pasa si definimos filter en forma recursiva?

```
filter2(_, [], []).
filter2(Criterio, [X|Original], [X|Nueva]):-
   call(Criterio, X), filter2(Criterio, Original, Nueva).
filter2(Criterio, [_|Original], Nueva):-
   filter2(Criterio, Original, Nueva).
```

Es más, estamos jugando a que filter2 tenga explosión combinatoria, en cada elemento de la lista original elijo y no elijo considerarla como parte de la solución. Esto nos arma la combinatoria de posibles sublistas:

```
?- filter2(even, [1, 3, 2, 4, 5], X).
X = [2, 4] ;
X = [2] ;
X = [4] ;
X = [] ;
```

Por otra parte, si hubiéramos considerado las últimas dos cláusulas excluyentes:

```
filter3(_, [], []).
filter3(Criterio, [X|Original], [X|Nueva]):-
   call(Criterio, X), filter3(Criterio, Original, Nueva).
filter3(Criterio, [X|Original], Nueva):-
   not(call(Criterio, X)),filter3(Criterio, Original, Nueva).
```

Esto corta el árbol de soluciones posibles: cada vez que tengo un elemento de la lista, considero que sólo puede haber un curso de acción válido: o lo filtro por no cumplir el criterio, o lo incluyo. Entonces filter3 pasa a parecerse más a una función con una única solución:

```
?- filter3(even, [1, 3, 2, 4, 5], X).
X = [2, 4] ;
```

Otro ejemplo:

```
?- filter(padre(homero), [matute, chiara, bart, daisy, lisa, etc],
```

```
HijosDeHomero).
HijosDeHomero = [bart, lisa].
```

4 Efecto colateral en Lógico

Al hacer la consulta
```
?- persona(juan).
```
sobre una base de conocimientos vacía, esperamos que el motor responda `false` porque no puede inferir que Juan es una persona.

Nosotros podemos agregar conocimiento en forma dinámica, con el predicado assert/1:
```
?- assert(persona(juan)).
```

Esto trae un efecto colateral, dado que si ahora preguntamos si Juan es una persona la respuesta es sí:
```
?- persona(juan).
true
```

Esto parece bastante obvio, pero qué ocurre si agregamos una regla a la base de conocimientos:
```
?- assert(jugador(X):-persona(X)).
```

Al preguntar si Juan es un jugador, nos responde que sí.
```
?- jugador(juan).
true
```

Ahora quitamos de la base de conocimientos el hecho que afirma que Juan es una persona:
```
?- retract(persona(juan)).
```

Y nuevamente vemos que hay un efecto colateral: al remover un hecho del cual depende, el predicado jugador se ve afectado:
```
?- jugador(X).
false
```

El predicado listing permite ver el conjunto de predicados definidos para la base de conocimientos. Entonces podemos ver que al agregar un predicado esa lista se ve afectada por un cambio de estado:

```
?- listing.
```

```
:- dynamic jugador/1.

jugador(A) :-
        persona(A).

:- dynamic persona/1.

persona(seba).
persona(maxi).
```

Al hacer

```
?- assert(jugador(darth_vader)).
```

Vemos el impacto en la base de conocimientos:

```
?- listing.
:- dynamic jugador/1.

jugador(A) :-
        persona(A).
jugador(darth_vader).

:- dynamic persona/1.

persona(seba).
persona(maxi).
```

El predicado retract/1 puede usarse con variables también, podemos remover así el predicado jugador/1:

```
?- retract(jugador(X)).
X = darth_vader ;
```

Pero si queremos eliminar la regla jugador(X):-persona(X), tenemos que hacerlo explícitamente:

```
?- retract(jugador(X):-persona(X)).
```

Vemos el impacto que tuvo en la base de conocimientos:

```
?- listing.
:- dynamic persona/1.

persona(seba).
persona(maxi).
```

4.1 Conclusiones

¿Qué nos enseña este ejercicio? Que la idea naif de no tener efecto colateral se termina en algún momento, pero eso no quita que en todo sistema nosotros buscamos, en distintos momentos:

- Que haya efecto colateral
- Que no haya efecto colateral.

Ambas cosas son deseadas: si quiero saber qué clientes son incobrables, si analizo la facturación de los últimos 12 meses, o si estoy buscando la relación entre los aumentos de sueldo y el aumento o disminución de la rotación del personal, no solo puedo trabajar sin efecto colateral, quiero hacerlo sin que ningún predicado que yo escriba tenga ese efecto. Si por el contrario estoy dando de alta un cliente, cargando las notas de un alumno, o generando el recibo de sueldo de un empleado, claramente quiero que esa operación modifique el estado de los individuos. Mientras que los lenguajes orientados a objetos y los "tradicionales" me llevan a pensar en variables como posiciones o referencias que puedo pisar a través de la asignación destructiva, Prolog en cambio define

- una gran mayoría de predicados que no trabajan con efecto colateral (los que hemos visto en los módulos anteriores)
- y algunos predicados específicos para abrir la puerta a que nuestro sistema tenga un estado que pueda modificarse.

5 Integración entre Lógico y otros paradigmas

Si querés jugar un poco te dejamos el link a una pregunta hecha en Stack Overflow:
http://stackoverflow.com/questions/4303931/how-use-prolog-from-java