Para leer en casa…

Repaso de conceptos de Lógico y Funcional

Declaratividad

- No hay secuencia de pasos (casi)
- No hay asignación destructiva

Yo digo qué condiciones hay que cumplir, el motor de inferencia termina armando el algoritmo.
Qué → YO
Cómo → Motor

¿Y en Funcional? Tampoco teníamos asignación destructiva ni secuencia de pasos. ¿Quién resolvía una expresión? El reductor de expresiones de Haskell, o sea, otro motor (en este caso de expresiones).

Si tenemos declaratividad, alguien tiene que hacer la magia de generar el algoritmo. Lo importante es que me abstraigo del proceso de traducción.

Tipos

Haskell ¿es débil o fuertemente tipado?
Bueno, puedo hacer
length de distintas listas y me devuelve su longitud, independientemente de lo que cada lista constituya.

Pero no puedo hacer filter y pasarle una función que no devuelva boolean. Es más, cuando quiero hacer "hola" + " mundo" *alguien* chequea que el operador (+) no está definido con operandos Strings.

Haskell tiene inferencia automática de tipos y permite utilizar tipos genéricos (los a, b y c, con alguna posibilidad de agrupar tipos en clases). El tipo existe y además el intérprete Haskell chilla cuando no puede asignar valores a los tipos de los argumentos que espera.

Es menos estricto que C porque el reductor de expresiones le permite darse cuenta de qué tipos tienen que ser los argumentos.

Ahora bien, SWI-PROLOG ¿es fuerte o débilmente tipado?
Puedo definir:

```
bueno(2).
bueno(juan).
bueno(1.3).
```

Eso me permite preguntar

```
? bueno(X)
X=2 ;
X=juan ;
X=1.3 ;
No
```

También puedo definir

```
malo(X):-not(bueno(X)).
```

Y no tengo que decir de qué tipo son los argumentos ➔ entonces, ¿no existen los tipos?
Vemos que en realidad, si yo quiero definir malo en función de bueno, me interesa que lo que reciba sea una persona:
malo(X):-persona(X), not(bueno(X)).

Como vimos antes, además de hacer malo inversible, también fija X como un tipo persona.

Si bien el compilador no lo exige, yo sí trabajo con tipos. Los tipos existen **siempre**.

- Si el lenguaje los chequea, es fuertemente tipado.
- Si el lenguaje no los chequea (como el SWI-PROLOG), es débilmente tipado.

No todos los PROLOG son débilmente tipados, el Turbo Prolog es fuertemente tipado.

Una lista en Haskell necesitaba que todos los elementos fueran del mismo tipo.

¿Por qué? Porque Haskell es fuertemente tipado.
¿Qué pasa en un lenguaje débilmente tipado como PROLOG?
Hagamos:

```
?- findall(Bueno, bueno(Bueno), Buenos).
Buenos = [1.3, 2, juan]
```

¡Aaaaaay! ¡Poray cantaba Garay!

Ahora, ¿qué sentido tiene esto? Tiene sentido si a los elementos puedo hacerle operaciones comunes en un determinado contexto… o sea, ver lo que sigue:

Polimorfismo

En funcional

- Una cosa es decir longitud de cualquier lista (*polimorfismo paramétrico*), donde acepto un tipo genérico
- Otra cosa es decir *polimorfismo ad-hoc*, donde sumo cualquier lista de números, sean Integer, Floats, Fracciones, etc.

Esto significa que una función utiliza distintos tipos de parámetro de la misma forma: a mí me da lo mismo si son enteros o fracciones, yo los sumo y ya.

En Lógico, ¿qué concepto está relacionado con el polimorfismo? Los *functores*.
Puedo trabajar con cosas que en un determinado contexto para mí son similares.
O sea: un libro, una película y una revista para un coleccionista pueden ser cosas “parecidas” (polimórficas). A mí me interesa saber cuántas cosas valiosas tengo:

- Una foto es valiosa si la foto es de 1967 o anterior.
- Una película es valiosa si actúa Catherine Zeta Jones.
- Un libro es valioso si lo escribió Cortázar.

Yo vendo una cosa si la considero valiosa y … bla …

```
vendo(Cosa):- esValiosa(Cosa), ...

esValiosa(foto(_, Anio)):- Anio < 1968.
esValiosa(película(_,_,catherine_zeta_jones)).
esValiosa(libro(_, cortazar)).
```

Fíjense que sólo en el predicado esValiosa/1 discrimino si es foto, película o libro. Esto me resulta útil para que vendo/1 no se complique. Simplemente le delega la responsabilidad al predicado esValiosa/1 y con eso basta.

Orden superior

¿Qué representa el concepto orden superior?

En Funcional, una función de orden superior es la que devuelve o recibe funciones (ya que abstracción principal del paradigma es la función). ¿Cuál es el sentido de hacer esto? Poder dividir un problema en varias partes y delegar a cada parte una porción de la solución final.

Lo que estamos queriendo decir es que hay

- **abstracciones de primer orden**: las funciones que trabajan con números, strings, booleans, etc.
- **abstracciones de orden superior**: son funciones que tienen un nivel más de generalización. Map aplica una función a una lista de elementos, Filter filtra los elementos de una lista que cumplen un criterio; map es mucho más general que una función que multiplica por dos los elementos de una lista de números.

En Lógico, tenemos los predicados de orden superior.
not/1, findall/3, forall/2 son predicados de orden superior porque reciben predicados como argumentos. El predicado, la abstracción principal del paradigma lógico, se utiliza a sí mismo como parámetro. Entonces esas reglas se definen en base a otras reglas: aumenta el power que le podemos dar, y también aumenta la expresividad: lo que decimos queda más simple. Ejemplo:

```
looser(Flaco):-
    hombre(Flaco),
    forall(conoce(Flaco, Muchacha), rebota(Flaco, Muchacha)).
```

Comparemos este predicado para saber si un muchacho es un looser contra una solución hecha en cualquier lenguaje del paradigma imperativo…

Recursividad

En funcional y en lógico se da que hay funciones que se llaman a sí mismas y predicados que se relacionan consigo mismos. Tanto el paradigma funcional que está basado en un modelo matemático como el lógico que proviene de la lógica tienen una fuerte base inductiva para obtener soluciones.

Como hemos visto, la inducción está asociada a la recursividad, y en este tipo de paradigmas donde no hay efecto de lado, la recursividad reemplaza a la estructura de iteración de los paradigmas imperativos (sí, el for, el while, el repeat-until y las otras estructuras que vieron de C y Pascal).

Unificación y pattern matching

El pattern matching es un concepto que existe tanto en Lógico como en Funcional. No obstante recordemos dos diferencias:

- En Lógico existe la posibilidad de dejar variables sin unificar (sin asignar un valor), para que el motor de inferencia encuentre todas las soluciones posibles, mientras que en Funcional no existe esa posibilidad ➔ la inversibilidad es un concepto propio del

paradigma lógico no aplicable al funcional (no puedo hacer `bueno X` en Funcional porque no tiene sentido bajo el concepto matemático de función).

- En Lógico el motor trata de buscar todas las posibilidades para cada patrón, mientras que en Funcional se encaja siempre el primer patrón encontrado. Recordar el ejemplo aritmético de factorial:

```
factorial(0, 1).
factorial(N, FN):- N1 is N - 1, factorial(N1, FN1), FN is FN1 * N.
```

¿Qué pasa con factorial(0, …)?
Matchea las dos cláusulas por pattern matching.

¿Por qué nos pasa esto?
Y… estamos hablando de una función. Recordemos la definición de función: a un dominio sólo puede corresponderle una imagen. Si bien transformarla en una relación (inversible) está bueno, también tengo que estar prevenido de las consecuencias que trae cambiar de paradigma.

Bueno, muy rico todo…

…¿y para qué me sirve?

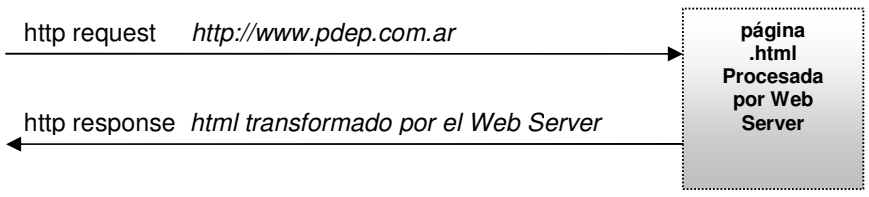
Las aplicaciones "de libro" de Lógico:

- Sistemas expertos
- Procesamiento de lenguaje natural
- Inteligencia artificial

Pero hay muchos otros campos donde un lenguaje en lógico podría ser mejor:

1) Necesitamos armar un motor de reglas de facturación (si el cliente es del Interior tiene un 5% de recargo, si el cliente es Gran Empresa tiene un 20% de descuento, etc.)
2) Tenemos la información de los libros que compró un cliente, de libros "afines" en base a los temas que tocan y queremos mandar una carta a los clientes que nos compraron ofreciéndole "otros libros que pueden ser de su interés".
3) Con la misma información de libros que compró un cliente, la inversibilidad nos sirve para conocer qué libros compró el cliente Gustavo Yalvé, o bien quiénes compraron el libro "Crimen y castigo".

En cuanto a funcional, un Web Server calza bastante bien a la metáfora de la calculadora del paradigma funcional: yo pido páginas, el servidor las procesa, las "transforma" y obtiene html como respuesta.



Limitaciones en Lógico

Ningún sistema se banca 100% ausencia de efecto colateral. En algún momento el paradigma puro se quiebra y aparecen los grises (que están fuera del alcance de la materia). Solamente

queremos decirles que existen formas de guardar información en la base de conocimientos en forma externa (assert, asserta, assertz, retract) y eso hace que el efecto de lado se pierda.

También tenemos límites a la hora de definir predicados inversibles. Predicados basados en negación de otros predicados, cláusulas findall/forall y cláusulas basadas en aritmética y comparación son algunas limitantes técnicas. En otras ocasiones, tengo limitaciones en la definición misma de la cláusula:

sumatoria(Lista, N), o
filtrarNegativos(Lista, ListaConPositivos).

En ambos casos la relación ya está pensada como una función donde el primer argumento siempre estará unificado y el segundo será el que puede estar o no ligado.

Y finalmente, cuando los predicados son no-determinísticos (no puedo saber la cantidad de soluciones de antemano), hacer la explosión combinatoria me expone a riesgos de degradar la performance o bien de traer soluciones repetidas. De todas maneras consideramos que vale el intento de conservar en esta materia la inocencia original del paradigma, para que luego al encarar desarrollos en su profesión puedan tomar los conceptos que más les sirvan.

Y esto es sólo el comienzo, amigos…