Prolog: unificación y recursión | MRC

Víctor Peinado

17-18 de diciembre de 2015

Referencias

- (Blackburn, *et al.*, 2006, chap. 2) [1]
- (Clocksin & Mellish, 2003, chap. 1) [2]
- Curso de Programación Lógica de la UPV/EHU [3] [4]
- Prolog: Recursive Rules [5]

[1] Blackburn, P., Bos, J., Striegnitz, K. *Learn Prolog Now!*. College Publications. Texts in Computer Science, vol 7. 2006. `http://www.learnprolognow.org/lpnpage.php?pageid=online`

[2] Clocksin, W., Mellish, C. S. *Programming in Prolog*. Springer Science & Business Media. 2003. `http://books.google.es/books?id=VjHk2Cjrti8C`

[3] Navarro, M. *Curso de Programación Lógica*. Tema 1. UPV/EHU. `http://www.sc.ehu.es/jiwhehum2/prolog/Temario/Tema1.pdf`

[4] Navarro, M. *Curso de Programación Lógica*. Tema 2. UPV/EHU. `http://www.sc.ehu.es/jiwhehum2/prolog/Temario/Tema2.pdf`

[5] `https://en.wikibooks.org/wiki/Prolog/Recursive_Rules`

Unificación

Como hemos visto antes, en Prolog hay tres tipos de términos:

- constantes, que pueden ser átomos (`socrates`, `zeus`) o números (3, 23.5).
- variables como `X` , `_Z3` o `Dioses`
- términos complejos, con la estructura `funtor(t_1..., t_n)`.

Es muy importante conocer cómo funciona el intérprete de Prolog a la hora de tratar de instanciar variables para unificar términos. Cuando Prolog unifica dos términos, lleva a cabo todas las instanciaciones necesarias para asegurar que estos dos términos son iguales. Esta funcionalidad, unida a la posibilidad de crear términos complejos con estructuras anidades, resulta un mecanismo muy potente y la principal ventaja a la hora de utilizar Prolog.

En Prolog, decimos que dos términos distintos `termino1` y `termino2` unifican en los siguientes casos:

1. si `termino1` y `termino2` son constantes unifican si y solo si son el mismo átomo o el mismo número:

Unificación de constantes

Como ves en estos dos primeros ejemplos, en Prolog existe el operador de igualdad `=/2`. Es más sencillo utilizarlo como infijo, como en el resto de ejemplos.

```
?- =(zeus, zeus).
true.

% ídem para los enteros
?- =(4, 4).
true.

?- zeus = zeus.
true.

?- 4 = 4.
```

```
true.

% por el contrario, dos contantes diferentes no unifican
?- a = 25.
false.
```

Unificación de variables

2. si `termino1` es una variable y `termino2` es cualquier tipo de término unifican, y además, `termino1` se instancia a `termino2`. En sentido contrario funciona exactamente igual. Y si ambos términos son variables, unifican porque cada uno se instancia con el otro y comparten valores.

```
?- Persona = juan.
Persona = juan.

?- Uno = 1.
Uno = 1.

?- 2 = Dos.
Dos = 2.

?- Uno = Otro.
Uno = Otro.
```

Unificación de términos complejos: definición recursiva

3. Si `termino1` y `termino2` son términos complejos, unifican si y solo si:

- tienen el mismo funtor y la misma aridad;
- todos sus argumentos unifican, y;
- las instanciaciones de las variables son compatibles

```
?- divinidad(zeus) = divinidad(zeus).
true.

?- divinidad(zeus) = divinidad(Dios).
Dios = zeus.

?- amigos(persona(pepe), Amigo1)  =  amigos(Amigo2, colega(juan)).
Amigo1 = colega(juan),
Amigo2 = persona(pepe).

/* aquí es imposible que X se instancie
con un valor válido para ambos hechos */
?- padre(cronos, X) = padre(X, zeus).
false.
```

```
?- ama(X, X) = ama(pepe, maria).
false.
```

Particularidades de la unificación en Prolog

La unificación es un procedimiento que existe en otras ramas de la informática y Prolog lo utiliza extensivamente, pero de manera distinta al que encontramos en otros ámbitos.

La consulta `padre(X) = X` fallará siempre en un sistema de unificación general. Si instanciamos la `X` con cualquier término, p. ej., `padre(padre(hermes))` la parte de la izquierda de la consulta será `padre(padre(padre(hermes)))` y la de la derecha `padre(padre(hermes))` y obviamente no unifican. Si elegimos cualquier otro término, nunca unificarán, porque la parte izquierda siempre tendrá un funtor más que la de la derecha.

Dado el funcionamiento recursivo de la unificación en Prolog, el intérprete intentará unificar siguiendo la regla número 2 que hemos visto más arriba, así que instanciará `X` con `padre(X)`. Pero como hay una variable `X` también en el término del lado izquierdo, se dará cuenta de que debería instanciarlo con `padre(padre(padre(X)))`, y después con `padre(padre(padre(padre(X))))`... entrando en un bucle.

En implementaciones antiguas de Prolog, una consulta como la anterior terminaba desbordando la memoria del ordenador y bloqueando el intérprete. En versiones modernas, sin embargo, se ha optado por modificar el comportamiento, de manera que sí unifiquen:

```
?- persona(X) = X.
X = persona(X).
```

Por el momento, para nuestros intereses, nos basta con adelantar que este mecanismo de unificación se ajusta perfectamente al modelado de conocimiento lingüístico, cuya organización natural tiene un claro carácter jerárquico. Piensa por un momento que a menudo definimos las oraciones de una lengua como secuencias de sintagmas nominales y sintagamas verbales; y un sintagma nominal puede definirse como una secuencia de un determinante y un nombre con un adjetivo opcional; un sintagma verbal es una secuencia de un verbo y un sintagma nominal, etc.

Estrategia de búsqueda de Prolog

En este apartado explicaremos el funcionamiento interno de Prolog a

Recuerda que el mecanismo de inferencia de Prolog es *modus ponens*.

la hora de realizar inferencias y deducir nueva información a partir del conocimiento explícito que aparece en las bases de conocimiento.

Dada la siguiente base de conocimiento:

```
% kb3: notas de clase
viene_a_clase(pepe).
viene_a_clase(maria).

estudia(pepe).
estudia(maria).

borda_el_examen(maria).

saca_buena_nota(X) :-
    viene_a_clase(X),
    estudia(X),
    borda_el_examen(X).
```

¿Cuál es la única respuesta posible?

```
?- saca_buena_nota(Quien).
Quien = maria.
```

¿Cómo hace Prolog para deducir precisamente `saca_buena_nota(maria)`? Prolog ha leído la base de conocimiento de arriba a abajo y la recorre sin éxito hasta que unifica `saca_buena_nota(Quien)` con la cabeza de la última regla.

Cuando Prolog unifica la variable de una consulta con otra variable que aparece en un hecho o en una regla, genera una nueva variable (las suele llamar de manera interna como `_G34`). A contiuación, tratará de comprobar la validez del cuerpo de la regla, estableciendo la siguiente lista de objetivos: `viene_a_clase(_G34)`, `estudia(_G34)`, `borda_el_examen(_G34)`.

En este momento, intentará satisfacer los objetivos por orden de izquierda a derecha. En primer lugar, instanciará `_G34` con la constante `pepe` ya que el primer hecho compatible que encuentra le permite unificar el objetivo `viene_a_clase(_G34)` con `viene_a_clase(pepe)`. Una vez hecho esto, unificará `estudia(_G34)` con `estudia(pepe)` al instanciar `_G34` de nuevo con `pepe`. A partir de aquí, sin embargo, no podrá continuar porque no hay manera de satisfacer el objetivo `borda_el_examen(pepe)`.

¿Qué ocurre ahora? Prolog se da cuenta de que ha cometido un error y vuelve atrás en sus pasos, buscando alternativas que le permita unificar `_G34`. Cade vez que Prolog toma encuentra varias alternativas crea una *señal* y la registra, de manera que puede volver atrás

cuando es necesario. Este proceso fundamental recibe el nombre de ***backtracking*** o **marcha atrás**.

Pues bien, cuando el intérprete eligió instanciar `_G34` con la constante `pepe` y unificar `viene_a_clase(_G34)` con `viene_a_clase(pepe)` dejó una señal a la que ahora puede volver. instanciará `_G34` con la constante `maria` y unificar `viene_a_clase(maria)` con `viene_a_clase(maria)`. Y seguirá adelante, satisfaciendo los objetivos `estudia(maria)` y `borda_el_examen(maria)`, e instanciando la variable `Quien` con la constante `maria`.

Recursión

Para entender la recursión primero necesitas comprender en qué consiste la recursión.
Anónimo.

En Prolog podemos tener definiciones recursivas. Un predicado está definido de manera recursiva si contiene una o más reglas que hacen referencia al propio predicado. En general, una definición recursiva es la que se hace en términos de sí misma.

Tomemos como ejemplo la siguiente base de conocimiento, con la línea sucesoras de algunos reyes de España:

```
% reyes de España
padre(carlos_I, felipe_II).
padre(felipe_II, felipe_III).
padre(felipe_III, felipe_IV).
padre(felipe_IV, carlos_II).

% mi abuelo es el padre de mi padre
abuelo(Abuelo, Nieto) :- padre(Abuelo, Padre), padre(Padre, Nieto).
```

Si lanzamos algunas consultas, vemos cómo se comporta la regla `abuelo`:

```
?- abuelo(Abuelo, carlos_II).
Abuelo = felipe_III .

?- abuelo(Abuelo, Nieto).
Abuelo = carlos_I,
Nieto = felipe_III ;
Abuelo = felipe_II,
Nieto = felipe_IV ;
Abuelo = felipe_III,
Nieto = carlos_II ;
false.
```

Imaginemos que necesitemos ampliar la base de conocimiento añadiendo una regla que defina el concepto de *ancestro*. Primer aproxi-

mación:

```
% mi padre es mi ancestro
ancestro(A, B) :- padre(A, B).
% el padre de mi padre es mi ancestro
ancestro(A, B) :- padre(A, X), padre(X, B).
% el padre del padre de mi padre es mi ancestro
ancestro(A, B) :- padre(A, X), padre(X, Y), padre(Y, B).
% ¿seguimos ad nauseam?
ancestro(A, B) :-
  padre(A, X),
  padre(X, Y),
  padre(Y, Z),
  padre(Z, B).
```

¿Hasta donde seguimos si queremos ser capaces de capturar todos los ancestros más allá de la cuarta generación? Está claro que esta estrategia no es cómoda ni es elegante.

Lo que realmente necesitamos es una definición de *ancestro* que funcione para cualquier linaje, independientemente del número de generaciones. Pensemos en otro modo de definir *ancestro*. Primero en lenguaje natural:

1. mi padre es mi ancestro; y,
2. el ancestro de mi padre es mi ancestro.

Nótese que esa segunda condición es una definición circular y puede parecer confusa, pero si combinamos los dos casos tenemos una definición completa, que en Prolog podemos implementar tal que:

```
ancestro(A, B) :- padre(A, B).
ancestro(A, B) :- padre(A, X), ancestro(X, B).
```

La primera regla recibe el nombre de caso base y es la que permite a la definición circular salir del bucle. La segunda es una definición recursiva que se llama a sí misma.

Cómo funciona esta definición, probemos a consultar cuáles son los ancestros de Felipe II:

```
?- ancestro(Ancestro, carlos_II).
Ancestro = felipe_IV ;
Ancestro = carlos_I ;
Ancestro = felipe_II ;
Ancestro = felipe_III ;
false.
```

¿Y los descendientes de Carlos I?

¿Qué ha pasado aquí? ¿Por qué no sigue buscando?

```
?- ancestro(carlos_I, Descendiente).
Descendiente = felipe_II .
```

Buenas prácticas con la recursión

Un gran poder conlleva una gran responsabilidad.
Tío Ben

Como hemos visto (y como veremos más adelante), la recursión es un mecanismo muy potente, pero no está exento de peligros. Como cualquier otro bucle en programación, necesitamos controlar cuándo va a parar. De lo contrario, podemos provocar que la memoria del ordenador se desborde.

Teniendo en cuenta cómo funciona el intérprete de Prolog (lee la base de conocimiento por orden, de arriba a abajo, y de izquierda a derecha), hay un par de reglas de oro que debemos tener en cuenta a la hora de programar con definiciones recursivas.

1. Coloca el caso base siempre al principio.

```
a :- b.
```

2. Evita la recursión por la izquierda. Cuando escribas la regla recursiva, asegúrate que tiene este formato:

```
a :- b, a.
```

Si te fijas, ambas restricciones son muy parecidas. La idea que subyace en estas recomendaciones es que evitar que Prolog entre en un bucle antes de evaluar los objetivos no recursivos. De lo contrario, entraría se quedaría atrapado en un bucle. Y esto, en el mejor de los casos, implica que el interprete trabajará innecesariamente. Y en el peor de los casos, el programa cascará por memoria insuficiente.

Ejercicios

1. Definir predicados para *descendiente*, *pariente*.
2. Base de conocimiento de digestión: ejemplo de LPN.
3. Base de conocimiento de países que son frontera y trayectos.
4. Otros ejercicios [6]

[6] http://www.learnprolognow.org/lpnpage.php?pagetype=html&pageid=lpn-htmlse11