Lenguajes de Programación I
Programación Lógica

Ernesto Hernández-Novich
<emhn@usb.ve>

Universidad "Simón Bolívar"

Programación Lógica

- Estilo declarativo de programación basado en:
  1. Definir la colección de **axiomas** y **reglas de inferencia** (*facts* y *rules*).
  2. Presentar un **teorema objetivo** (*goal*).
  3. Implantación explora el espacio de soluciones hasta encontrar una colección de axiomas y pasos de inferencia que impliquen el objetivo.
- Los programas son colecciones de *predicados* que describen el problema, expresando las relaciones entre sus partes.
- Prolog es el lenguaje más utilizado.

¿Cómo se explora el espacio de soluciones?

- **Cláusulas de Horn**

$$H \leftarrow B_1, B_2, \ldots, B_n$$

  H es la conclusión a partir de la conjunción de los B_i –
  si H no requiere antecedentes, se considera un axioma (*fact*).
- El lenguaje procura derivaciones a partir de los axiomas existentes utilizando el proceso de **resolución**.
  - Si se tienen las reglas...

$$C \leftarrow A, B$$
$$D \leftarrow C$$

  - ...y se dan A y B como ciertos.
  - Entonces el lenguaje puede concluir que D es "demostrable".

Prolog

- Resolución sobre **base de datos de cláusulas** que se asumen ciertas.
- Cláusulas compuestas por **términos**
  - **Números** enteros o reales.
  - **Atomos**
    - Identificadores que comienzan con minúscula.
    - Cadena entre comillas.
    - Secuencia de símbolos de puntuación.

```
foo  un_atomo  'El atomo'  =:=
```

  - **Variables** nombradas que comienzan con mayúscula, o anónimas que comienzan con *underscore*.

```
Var1  Esta_es_una_variable  Z  _obey
```

  - **Estructuras** construidas con un átomo **functor** y sus argumentos

```
foo(bar,baz) un_functor(unario)
```

¡No son funciones sino predicados multivariable!

Variables y Functores

Son todas locales

- Variables se **instancian** a tiempo de ejecución con valores concretos – esto ocurre durante el proceso de **unificación**.
- Alcance limitado a la cláusula en la cual aparece.
- Verificación dinámica de tipos.
- Anidamiento arbitrario de estructuras.

  ```
  foo(bar(X,Y),baz(qux),grok(A,42,69,_))
  ```

  permite simular registros o tipos de datos recursivos arbóreos.
- Se denota **Predicado** a la combinación de un functor y su aridad – `foo/3`, `bar/2`, `baz/1`, `qux/0` y `grok/4`.

Cláusulas

Punto final

- En la base de datos de cláusulas puede haber
  - **Hechos** – cláusulas de Horn *sin* lado derecho.

    ```
    esto(es,un,hecho).
    otro(hecho).
    estoEsUnHecho.
    ```
  - **Reglas** – cláusulas de Horn *con* lado derecho, en las cuales se usa `:-` (el "cuello") como símbolo de implicación ($\Leftarrow$).

    ```
    abuelo(X,Y) :- padre(X,Z), padre(Z,Y).
    foo(X) :- bar, baz(Y), grok(qux,X,Y).
    ```
- **Consulta** – cláusula de Horn *sin* lado izquierdo.
  - Usualmente escritas en el interpretador.
  - Ocasionalmente incluidas como "inicialización" o "programa principal" en el texto del programa.

  ```
  ?- mofo(You), do(You,speak(english)).
  ```

Principio de Resolución

- Si C_1 y C_2 son cláusulas de Horn, y el lado izquierdo de C_1 coincide con uno de los términos del lado derecho de C_2, entonces podemos reemplazar el término C_2 por el cuerpo de C_1.
- Esto puede usarse para concluir nuevos resultados en base a hechos de la base de datos, i.e. si tenemos

  ```
  padre(miguel,ernesto).
  padre(ernesto,santiago).
  abuelo(X,Y) :- padre(X,Z), padre(Z,Y).
  ```

  podemos instanciar `X` con `miguel` y `Z` con `ernesto` para deducir

  ```
  abuelo(miguel,Y) :- padre(ernesto,Y).
  ```

Unificación

- Algoritmo para **instanciar** variables con valores concretos.
- Las reglas de unificación para Prolog son:
  1. Una constante unifica sólo consigo misma.
  2. Dos estructuras se unifican solamente si tienen el mismo functor, la misma aridad y los argumentos unifican entre sí.
  3. Una variable unifica con cualquier cosa
     - Si lo otro es un valor, se instancia con ese valor.
     - Si lo otro es otra variable sin instanciar, quedan ligadas para compartir el mismo valor una vez que alguna de ellas sea instanciada.
- Se descarta el *occurs check* por razones de eficiencia.

Listas heterogéneas

- El punto (.) es el constructor – `[]` denota la lista vacía.
- Pueden expresarse directamente

```
[ foo, 42, bar, f(X,grok) ]
```

que es equivalente a

```
.(foo, .(42, .(bar, .(f(X,grok) , []))))
```

- Para unificar, la barra vertical separa la cola de la lista.

```
member(X, [X|T]).
member(X, [H|T]) :- member(X,T).
```

Los parámetros y la unificación

- No aplica el concepto de parámetros ni resultados – porque no son funciones, sino predicados a satisfacer.
- En el clásico ejemplo...

```
append([],A,A).
append([H|T],A,[H|L]) :- append(T,A,L).
```

...se pueden preguntar cosas como

```
?- append([a, b, c], [d, e], L)
L = [a, b, c, d, e]
?- append(X, [d, e], [a, b, c, d, e])
X = [a, b, c]
?- append([a, b, c], Y, [a, b, c, d, e])
Y = [d, e]
```

Aritmética

Una cosa es la expresión y otra su valor

- Operadores disponibles como *predicados*.
- `18 + 3*8` es lo mismo que `+(18,*(3,8))` – no es más que una *estructura* con predicados `+/2` y `*/2`.
- ¡Y **no** "vale" 42!
- El predicado `is/2` permite evaluar si se unifica con alguna variable.

```
?- is(X,18+24).
X = 42
?- X is 18+24.
X = 42
?- 42 is 18+24.
yes
```

- No resuelve ecuaciones – `42 is 18+X` no tiene sentido.

Orden de Ejecución

- Ante un objetivo es necesario:
  - Comprobarlo encontrando al menos una secuencia de pasos que deduzca el objetivo a partir de las cláusulas.
  - Refutarlo demostrando que no existe tal secuencia.
- En la Lógica Formal existen dos estrategias:
  - ***Forward Chaining*** – partir de las cláusulas existentes e intentar avanzar hasta derivar el objetivo.
  - ***Backward Chaining*** – partir del objetivo e intentar retroceder hacia las cláusulas existentes.
- Prolog busca soluciones con *backward chaining* recorriendo un árbol de sub-objetivos en DFS de izquierda a derecha según el orden de aparición de las cláusulas.
- *Backtracking* cuando es necesario regresar en el árbol.

Implantación

- Pila con registros de activación se simula en *heap*.
- Cada registro de activación contiene las asociaciones que permiten unificar un sub-objetivo del lado derecho con algún lado izquierdo.
  - Aparecen cuando se intenta resolver un sub-objetivo.
  - Desaparecen cuando no hay más unificaciones posibles y debe regresarse al contexto anterior.
- Cuando se encuentra una unificación para todos los sub-objetivos de la consulta, se reporta la unificación y el éxito al llamador – pero la pila se preserva por si es necesario hacer *backtracking*.

Flujo de Control Imperativo

Implantando selectores

- Orden de las cláusulas establece el orden de exploración – el resultado de un programa es completamente determinístico...
- ...y potencialmente ineficiente!
- Posible cortar con `!` (`cut`) el *backtracking* a partir de cierto objetivo.
- Puede usarse para lograr el efecto de un selector imperativo

```
if :- predicate, !, then.
if :- else.
```

Flujo de Control Imperativo

Implantando ciclos

- Puede utilizarse un *generador* de soluciones posibles...

```
append([],A,A).
append([H|T],A,[H|L]) :- append(T,A,L).
```

- ...y obligar a un fallo para probar todas las combinaciones...

```
particiones(L) :- append(A,B,L),
                  write(A), write(' '),
                  write(B), nl,
                  fail.
particiones(L) :- write('The End').
```

Estrategia *Generate and Test*.

Flujo de Control Imperativo

Implantando contadores

- Puede utilizarse un *generador* de números...

```
n(1).
n(N) :- n(M), N is M+1.
```

- ...y obligar a un fallo para contar, pero cortar el *backtracking* hasta llegar al valor deseado

```
loop(TIMES) :- n(I), I <= TIMES,
               write(I), nl,
               I = TIMES, !, fail.
loop(TIMES) :- write('The End').
```

Meta-programación
Manipulación de la Base de Datos

- Posible alterar la base de datos de cláusulas a tiempo de ejecución.
  - `asserta/1` – agregando cláusulas al principio
  - `assertz/1` – agregando cláusulas al final
  - `retract/1` – eliminando cláusulas *exactas*
- Implantaciones modernas manejan de forma diferente y más eficiente las cláusulas estáticas (inmutables) que las dinámicas – la flexibilidad de cláusulas dinámicas debe indicarse explícitamente.

¿Pero cómo construimos una cláusula?

Meta-programación
Estudiando la estructura de functores

- Analizar predicados – `functor/3`.

```
?- functor(foo(a, X), Arg2, Arg3).
Arg2 = foo
Arg3 = 2
?- functor(Arg1, foo, 2).
Arg1 = foo(_A,_B)
```

- Analizar argumentos de un predicado – `arg/3`.

```
?- arg(2, foo(bar,baz), X).
X = baz
?- arg(1, foo(X,baz), Y).
X = Y
```

Meta-programación
Descomponiendo y recomponiendo functores

- Pasar de lista a predicado y viceversa – `=..` (univ).

```
?- Pred =.. [foo, bar, baz(Qux), Grok].
Pred = foo(bar,baz(Quz),Grok)

?- foo(Bar,baz(42),Grok) =.. L.
L = [foo,Bar,baz(42),Grok]
```

Consultar la base de datos – `ndfa.prlg`
Meta-programación

- ¿Existe una cláusula como ésta? – `clause/2`.

```
?- clause(nfa(E,I),C).

C = estado_final(E)
I = [] ? ;

C = transicion(E,A,B),nfa(B,D)
I = [A|D]
```

- Suministrar un lado izquierdo – obtener todos los lados derechos correspondientes por *backtracking*.
  - Si la cláusula es un hecho, lado derecho es `true`.
  - El resto son functores coma (,) anidados.
- El predicado debe ser **dinámico**.

Meta-programación

Prolog en Prolog

```
prove(true).
prove((Goal1, Goal2)) :- prove(Goal1),
                         prove(Goal2).
prove(Goal) :- clause(Goal, Body),
               prove(Body).
```

- ¿Cómo le agregarían la disyunción?
- ¿Cómo le agregarían traza de ejecución?

Interacción con el mundo real

- Control rudimentario de archivos
  - `see(File)` y `seen` – Abrir y cerrar entrada.
  - `tell(File)` y `told` – Abrir y cerrar salida.
- *Parser* y *Pretty-printer* incluidos
  - `read(X)` lee un término desde la entrada.
  - `write(X)` escribe un término a la salida.
  - `nl` y `tab(N)` – espaciado discreto.
- Algunas implantaciones ofrecen mecanismos para operar byte por byte, incluso sobre *sockets*.

Interacción con el mundo real

El *parser* es extensible

- Se declaran operadores...

```
:- op(500,xfx,'es_de_color').
```

- ...y después se usan libremente.

```
b es_de_color azul.
?- b es_de_color C.
C = azul
?- Que es_de_color azul.
Que = b
```

- Primer argumento – precedencia.
- Segundo argumento – asociatividad.
  - Prefijos – `fx` anida y `fy` no anida.
  - Postfijos – `xf` anida y `yf` no anida.
  - Infijos – `xfy` y `yfx` asocian hacia la `y`, pero `xfx` no asocia.

Lenguajes en Prolog

Procesamiento de lenguajes – gramáticas generales

- DCG (*Definite Clause Grammar*)

```
expr --> term.
expr --> term, [+], expr.
expr --> term, [-], expr.
term --> num.
term --> num, [*], term.
term --> num, [/], term.
num  --> [D], { number(D) }.
parse(E) :- expr(E,[]).
```

- Parámetros en no-terminales — construir estructuras.

```
num(D) --> [D], { number(D) }.
```

- Reconocedor por *backtracking*.

Bibliografía

- [*Scott*]
  - Capítulo 11
  - Ejercicios y Exploraciones