Representación del Conocimiento

Control del razonamiento

Dr. Alejandro Guerra-Hernández

Instituto de Investigaciones en Inteligencia Artificial
Universidad Veracruzana

Campus Sur, Calle Paseo Lote II, Sección Segunda No 112,
Nuevo Xalapa, Xalapa, Ver., México 91097

mailto:aguerra@uv.mx
https://www.uv.mx/personal/aguerra/rc

Maestría en Inteligencia Artificial 2024

Índice

1 Corte

2 Negación por fallo finito

3 La compleción de un programa

4 Resolución SLDNF para programas definitivos

5 Programas Lógicos Generales

6 Resolución SLDNF para programas generales

Control

- La resolución-SLD implica cierto orden de ejecución:
  - Las metas se resuelven de izquierda a la derecha.
  - Las cláusulas se exploran de arriba a abajo.
- El control la ejecución de un programa se lleva a cabo ordenando cláusulas y metas.
- Otra opción es el operador de corte (!) que evita la reconsideración característica de Prolog.
- El corte permite la definición de una forma de negación, llamada negación por fallo finito (NAF), asociada al supuesto del mundo cerrado (CWA).
- Revisaremos estas técnicas de control de razonamiento.

Universidad Veracruzana

A la Prolog

- Prolog reconsidera automáticamente para satisfacer una meta.
- Esto es útil porque evita que el programador contienda explícitamente con la reconsideración;
- Pero si no tenemos ninguna forma de control sobre el *backtracking*, éste puede volverse la causa de programas ineficientes.

Ejemplo

- Asumamos el siguiente programa:

$$
\begin{aligned}
padre(X, Y) &\leftarrow progenitor(X, Y), hombre(X). \\
progenitor(benjamin, antonio). \\
progenitor(maria, antonio). \\
progenitor(samuel, benjamin). \\
progenitor(alicia, benjamin). \\
hombre(benjamin). \\
hombre(samuel).
\end{aligned}
$$

- Y la meta: $\leftarrow padre(X, antonio)$.

Arbol-SLD

$$\leftarrow padre(X, antonio)$$

$$\leftarrow progenitor(X, antonio), hombre(X).$$

$$\leftarrow progenitor(benjamin, antonio). \qquad \leftarrow progenitor(maria, antonio).$$

$$\leftarrow hombre(benjamin). \qquad \leftarrow hombre(maria)$$

$$\square$$

Nodo origen

- Cada nodo del árbol-SLD corresponde a una meta de una derivación y tiene un átomo asociado:

$$G_0 \overset{\alpha_0}{\rightsquigarrow} G_1 \dots G_i \overset{\alpha_i}{\rightsquigarrow} \dots G_{n-1} \overset{\alpha_{n-1}}{\rightsquigarrow} G_n$$

- Asumamos que α_i no es un caso de una submeta de la meta inicial G_0, p.ej., $hombre(benjamin)$.
- Entonces α es el caso de un átomo β_i del cuerpo de una cláusula $\beta_0 \leftarrow \beta_1, \dots, \beta_i, \dots, \beta_n$, cuya cabeza unificó con una sub-meta G_j t.q., $0 \leq j \leq i$. P. ej., $\leftarrow progenitor(X, antonio), hombre(X)$.
- Es decir un nodo entre la raíz del árbol y el nodo de la meta G.
- El nodo de la meta G_j se conoce como el origen, o padre, de α_i y se denota como $origen(\alpha_i)$.

Nodo corte

- El corte ! es un átomo que siempre tiene éxito, con la substitución vacía ϵ como resultado.
- El nodo donde ! fue seleccionado es llamado el nodo de corte.
- Un nodo de corte puede ser visitado nuevamente durante la reconsideración.
- En este caso, el curso normal del recorrido del árbol es alterado, el recorrido continua en el nodo superior a *origen*(!).
- Si el corte ocurre en la meta inicial, la ejecución simplemente termina.

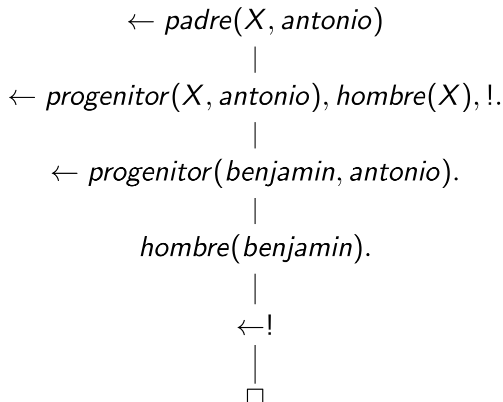
Ejemplo

- En el caso de $padre/2$, a lo más existe una solución para nuestra meta.
- Cuando la solución se encuentra, la búsqueda puede detenerse pues ninguna persona tiene más de un padre.
- Para explotar esta situación, el operador de corte se agrega al final de $padre/2$:

$$padre(X, Y) \leftarrow progenitor(X, Y), hombre(X), !.$$

Arbol de resolución con corte

- La búsqueda termina después de reconsiderar al nodo de corte.



Efectos del corte

1. Divide el cuerpo de la meta en dos partes –Después de éxito de !, no es posible reconsiderar hacía las literales a la izquierda del corte. Sin embargo, a la derecha del corte todo funciona de manera normal.
2. Poda las ramas sin explorar directamente bajo *origen*(!). En otras palabras, no habrá más intentos de unificar la submeta seleccionada de *origen*(!) con el resto de las cláusulas del programa.

Controversia

- La intención tras el corte, es poder controlar la ejecución de los programas, sin cambiar su significado lógico. Por ello la lectura lógica del corte es `true`.
- Operacionalmente, si el corte remueve sólo ramas fallidas del árbol-SLD, no tiene influencia en el significado lógico de un programa.
- Pero el corte puede remover también ramas exitosas del árbol-SLD, atentando contra la completitud de los programas definitivos, o la correctez de los programas generales.

Ejemplo

- Es bien sabido que los padres de un recién nacido están orgullosos:

$$orgulloso(X) \leftarrow padre(X, Y), recien_nacido(Y).$$

- Consideren las siguiente cláusulas adicionales:

$$\begin{aligned} padre(X, Y) &\leftarrow progenitor(X, Y), hombre(X). \\ progenitor(juan, maria). & \\ progenitor(juan, cristina). & \\ hombre(juan). & \\ recien_nacido(cristina). & \end{aligned}$$

Corridas

- La respuesta a la meta $\leftarrow orgulloso(juan)$ es `true`.
- Si remplazamos la primera cláusula, con su versión con corte:

  $$padre(X, Y) \leftarrow progenitor(X, Y), hombre(X), !.$$

  Y preguntamos nuevamente a Prolog, si

  $$\leftarrow orgulloso(juan).$$

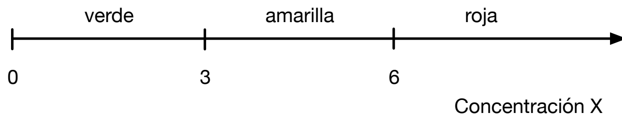
  la respuesta será `false`!
- No habrá reconsideración una vez que se ha encontrado que `maria` es hija de `juan`, pero no es recién nacida.
- Peor aún: $\leftarrow \neg orgulloso(juan)$ tendría éxito en la versión de nuestro programa que utiliza corte.

Cortes rojos y verdes

- Hasta ahora hemos distinguido dos usos del corte:
  1. Eliminar ramas fallidas en el árbol-SLD;
  2. y podar ramas exitosas.
- (1) se considera una práctica segura, porque no altera las respuestas producidas durante la ejecución de un programa: Cortes verdes.
- (2) se considera una práctica de riesgo: Cortes rojos.

Un semáforo ambiental

- Consideren una regulación sobre el nivel de alerta de la contaminación en una ciudad.



Reglas

- La relación entre X e Y se puede establecer mediante tres reglas:
  1. Si $X < 3$ entonces $Y = verde$
  2. Si $3 \leq X$ y $X < 6$ entonces $Y = amarilla$
  3. Si $6 \leq X$ entonces $Y = roja$
- En Prolog:

```
1 % Concentración X de contaminantes en el aire
2 % f(X,Y), Y estado de la alarma
3
4 f(X,verde) :- X<3.       % Regla 1
5 f(X,amarilla) :- 3=< X, X<6.  % Regla 2
6 f(X,roja) :- 6 =< X.       % Regla 3
```

Experimento 1

- Supongamos que el nivel de contaminación en la ciudad es $X = 2$.
- Asumamos que el usuario se pregunta si tal nivel de contaminación es inseguro y debiese generar una alarma amarilla:

```
1  ?- f(2,Y), Y=amarilla.
```

- ¿Cómo computa Prolog su respuesta? Al resolver la primer meta `f(2,Y)`, Y unificará con `verde`; de forma que la segunda meta toma la forma `verde=amarilla`, lo cual es falso y en consecuencia la lista completa de metas falla.
- Sin embargo, antes de rendirse, Prolog evaluará vía su reconsideración dos alternativas que resultan inútiles.

Universidad Veracruzana

Traza

```
1  [trace]  ?- f(2,Y), Y=amarilla.
2  Call: (9) f(2, _5824) ?
3  Call: (10) 2<3 ?
4  Exit: (10) 2<3 ?
5  Exit: (9) f(2, verde) ?
6  Call: (9) verde=amarilla ?
7  Fail: (9) verde=amarilla ?
8  Redo: (9) f(2, _5824) ?
9  Call: (10) 3=<2 ?
10 Fail: (10) 3=<2 ?
11 Redo: (9) f(2, _5824) ?
12 Call: (10) 6=<2 ?
13 Fail: (10) 6=<2 ?
14 Fail: (9) f(2, _5824) ?
15 false.
```

Universidad Veracruzana

Solución

- Las tres reglas de nuestro programa son mutuamente excluyentes, de forma que solo una de ellas tendrá éxito.
- Por ello, sabemos que tan pronto como una de ellas tiene éxito, no tiene caso intentar probar las otras, puesto que fallarán.
- Pero Prolog no sabe esto, será necesario indicárselo con dos cortes:

```
1  % Concentración X de contaminantes en el aire
2  % f(X,Y), Y estado de la alarma
3  % Con corte
4
5  f(X,verde) :- X<3, !.      % Regla 1
6  f(X,amarilla) :- 3=< X, X<6, !.  % Regla 2
7  f(X,roja) :- 6 =< X.       % Regla 3
```

Universidad Veracruzana

Traza

```
1  [trace]  ?- f(2,Y), Y=amarilla.
2  Call: (9) f(2, _9422) ?
3  Call: (10) 2<3 ?
4  Exit: (10) 2<3 ?
5  Exit: (9) f(2, verde) ?
6  Call: (9) verde=amarilla ?
7  Fail: (9) verde=amarilla ?
8  false.
```

Corte verde

- Observen que el comportamiento de ambos programas, con y sin corte, es el mismo.
- La única diferencia es que el primero tardará un poco más en dar su respuesta.
- En estos casos se dice que el operador corte solo cambia el significado procedural del programa.
- Como veremos más adelante, éste no es siempre el caso.

Universidad Veracruzana

Experimento 2

- Ejecutemos otro experimento con la versión con cortes de nuestro programa. Preguntémosle la siguiente meta:

```
1 ?- f(7,Y).
2 Y=roja
```

Traza

```
1  [trace]  ?- f(7,Y).
2  Call: (8) f(7, _10108) ?
3  Call: (9) 7<3 ?
4  Fail: (9) 7<3 ?
5  Redo: (8) f(7, _10108) ?
6  Call: (9) 3=<7 ?
7  Exit: (9) 3=<7 ?
8  Call: (9) 7<6 ?
9  Fail: (9) 7<6 ?
10 Redo: (8) f(7, _10108) ?
11 Call: (9) 6=<7 ?
12 Exit: (9) 6=<7 ?
13 Exit: (8) f(7, roja) ?
14 Y = roja.
```

Solución

- Las tres reglas son intentadas antes de poder computar la respuesta.
- Esto revela otra fuente de ineficiencia: algunos de los tests que se llevan a cabo son redundantes. Una tercer versión de nuestro programa sería como sigue:

```
1 % Tercer versión
2
3 f(X,verde) :- X<3, !.
4 f(X,amarilla) :- X<6, !.
5 f(_,roja).
6
```

Traza

```
1  [trace]  ?- f(7,Y).
2  Call: (8) f(7, _13718) ?
3  Call: (9) 7<3 ?
4  Fail: (9) 7<3 ?
5  Redo: (8) f(7, _13718) ?
6  Call: (9) 7<6 ?
7  Fail: (9) 7<6 ?
8  Redo: (8) f(7, _13718) ?
9  Exit: (8) f(7, roja) ?
10 Y = roja.
```

Observaciones

- La respuesta es la misma, pero esta versión del programa es más eficiente que las dos anteriores.
- Pero, ¿Qué sucede si removemos los operadores de corte? Entonces la salida de la meta sería:

```
1 Y=verde;
2 Y=amarilla;
3 Y=roja;
4 false
```

- Esto sugiere que a diferencia de nuestro segundo programa, los cortes aquí ¡cambian el resultado del programa!

Problemas

- ¿A qué se debe la respuesta de nuestro tercer programa a las siguientes metas?

```
1 ?- f(2,amarilla).
2 true
3 ?- f(2,Y), Y=amarilla.
4 false
```

Observaciones

- La traza de la primer meta explica lo sucedido:

```
[trace]  ?- f(2,amarilla).
Call: (8) f(2, amarilla) ?
Call: (9) 2<6 ?
Exit: (9) 2<6 ?
Exit: (8) f(2, amarilla) ?
true.
```

- Si no tenemos cuidado con el corte, podemos cambiar la semántica de nuestros programas inadvertidamente.

Max

```
1  % max
2
3  max(X,Y,X) :- X >= Y, !.
4  max(_,Y,Y).
```

Un sólo miembro de una lista

```
6  % single member
7
8  smember(X,[X|_]) :- !.
9  smember(X,[_|L]) :- smember(X,L).
```

Agregar sin duplicar

```
11   % agregar sin duplicar add(+X,+L,-Y)
12
13   add(X,L,L) :- smember(X,L), !.
14   add(X,L,[X|L]).
```

Clasificar en categorías

```
16  % clasificando en categorias
17
18  derrota(barcelona, real_madrid).
19  derrota(roma, barcelona).
20  derrota(cruz_azul, real_madrid).
21
22  % luchador es quien gana y pierde algunos partidos
23  % ganador es quien gana siempre
24  % deportista es quien pierde siempre
25
26  clase(X,luchador) :-
27      derrota(X,_),
28      derrota(_,X), !.
29
30  clase(X, ganador) :-
31      derrota(X,_), !.
32
33  clase(X, deportista) :-
34      derrota(_, X).
```

Ventajas del corte

1. Puede mejorar la eficiencia de los programas lógicos. La idea es decirle a Prolog explícitamente –No intentes otra alternativa, pues están condenadas al fracaso.
2. Puede usarse para definir cláusulas mutuamente exclusivas, por lo que podemos expresar reglas de la forma: Si la condición P entonces conclusión Q, y en cualquier otro caso R. Mejorando así la expresividad del lenguaje.

Universidad Veracruzana

Desventajas

- Fácilmente podemos perder la correspondencia entre el significado declarativo de nuestros programas y su significado procedural.
- Si cambiamos el orden de las cláusulas de nuestro programa, sin usar el operador de corte, afectamos la eficiencia o las condiciones de terminación de éste, sin cambiar su significado declarativo; pero si las cláusulas usan corte, un cambio en su orden si que puede afectar este significado.

Ejemplo

- Si queremos expresar en Prolog $p \Leftrightarrow (a \wedge b) \vee c$ podemos usar el siguiente programa:

```
1 p :- a,b.
2 p :- c.
```

- Podemos cambiar el orden de las cláusulas y el significado declarativo del programa sigue siendo el mismo. Introduzcamos un corte:

```
1 p :- a, !, b.
2 p :- c.
```

- El significado declarativo del programa es $p \Leftrightarrow (a \wedge b) \vee (\neg a \wedge c)$.
- Pero si invertimos el orden de las cláusulas el significado declarativo se vuele $p \Leftrightarrow c \vee (a \wedge b)$.

Problema

- ¿Cómo podemos expresar la siguiente proposición en Prolog?: A Adriana le gustan todos los animales, excepto las serpientes.
- La primer parte de la proposición es sencilla:

```
7 le_gusta(adriana,X) :-
8     animal(X).
```

- La segunda parte es más complicada:

```
1 % A Adriana le gustan todos los animales, excepto las
2 % serpientes
3
4 le_gusta(adriana,X) :-
5     serpiente(X), !, fail.
6
7 le_gusta(adriana,X) :-
8     animal(X).
```

En una sola cláusula

- De hecho, las dos cláusulas puede escribirse juntas de manera más compacta:

```
1  le_gusta(adriana,X) :-
2    serpiente(X), !, fail
3    ;
4    animal(X).
```

- Podemos usar la misma idea, para escribir un predicado *diferente*/2:

```
12 diferente(X,Y) :-
13     X=Y, !, fail
14     ;
15     true.
```

Not

- Estos ejemplos sugieren que sería conveniente tener un predicado *not*/1 con la siguiente definición:

```
1  not(P) :-
2    P, !, fail
3    ;
4    true.
```

- Esta definición descansa enteramente en la semántica de Prolog: Las sub-metas se deben resolver de izquierda a derecha, y las cláusulas se buscan en el orden en que aparecen en el texto del programa.

Ejemplos

- De forma que los ejemplos anteriores, se pueden definir ahora de la siguiente forma:

```
17 %%% Versiones con not
18
19 le_gusta2(adriana,X) :-
20     animal(X),
21     \+ serpiente(X).
22
23 diferente2(X,Y) :-
24     \+ (X=Y).
25
```

Correctez

- Ahora bien, la definición de la negación usando `fail` y corte tiene un problema serio: No se corresponde con la definición lógica de la negación.
- Ver Nilsson y Maluszynski [4], cap. 4.

Ejemplo I

- Consideren el siguiente programa de una sola cláusula:

```
1 pintor(artur_heras).
```

- Si le preguntamos al programa si `artur_heras` es un `pintor`:

```
1 ?- pintor(artur_heras).  true
```

- Lo cual es correcto pues esa respuesta se sigue del programa. Ahora, si preguntamos:

```
1 ?- pintor(juan_gris).  false
```

- Lo cual también es correcto puesto que no se sigue del programa que `juan_gris` sea un pintor.

Ejemplo II

- Ahora:

```
1  ?- \+ pintor(juan_gris).  true
```

- En este último caso, el *true* de Prolog no significa que la meta se siga lógicamente del programa.
- Este comportamiento no es robusto!

Close World Assumption (CWA)

- Un programa Prolog representa todo lo que es verdadero en el mundo que describe.
- Lo que no esté declarado en el programa, o no sea derivable de éste lógicamente, se asume como falso.
- El CWA como una pseudo-regla de inferencia:

$$\frac{\Delta \nvdash \alpha}{\neg \alpha} \quad (CWA)$$

- En los sistemas robustos y completos, la condición $\Delta \nvdash \alpha$ es equivalente a $\Delta \not\models \alpha$.
- Para la resolución-SLD la condición puede ser remplazada por $\alpha \notin M_{\Delta}$.

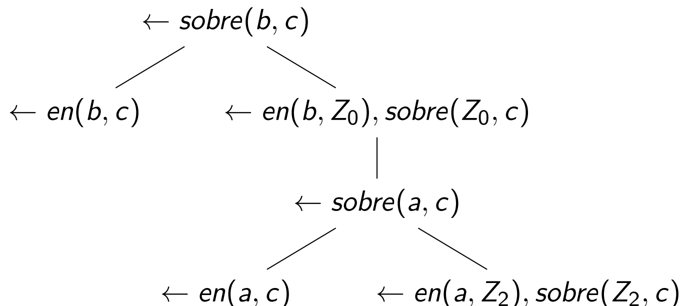
Ejemplo

- Dado el siguiente programa lógico:

$$
\begin{aligned}
sobre(X, Y) &\leftarrow en(X, Y).\\
sobre(X, Y) &\leftarrow en(X, Z), sobre(Z, Y).\\
en(c, b).\\
en(b, a).
\end{aligned}
$$

la fbf $sobre(b, c)$ no puede ser derivada por resolución-SLD a partir del programa Δ

Árbol de resolución



Problemas del CWA

- En la vida cotidiana existen muchas situaciones en donde el CWA no se puede asumir.
- Establecer que una meta no es derivable dado un programa definitivo es no decidible en el caso general, *i.e.*, no es posible determinar si la pseudo-regla asociada al CWA aplica o no.
- Una versión más débil se logra si asumimos que $\neg\alpha$ es derivable a partir del programa Δ si la meta $\leftarrow \alpha$ tiene un árbol-SLD finito que falla, *i.e.*, NAF.
- Ejemplo. El caso anterior.

CWA vs NAF

- Es necesario contrastar la NAF con el CWA, que también puede verse como una negación por falla, pero infinita.
- Ejemplo: Extendamos el programa Δ con la siguiente cláusula evidentemente verdadera $sobre(X, Y) \leftarrow sobre(X, Y)$.
- El árbol-SLD de la meta $\leftarrow sobre(b, c)$ sigue sin contener refutaciones, pero ahora es infinito.
- Por lo tanto no podemos concluir que $\neg sobre(b, c)$ usando NAF, pero si usando CWA.

Negación y cuantificadores

- Consideren el siguiente programa acerca de restaurantes:

```
29  estrellas(ricard_camarena).
30
31  caro(diverxo).
32
33  razonable(X) :-
34      \+ caro(X).
```

- Una meta de interés sería:

```
1   ?- estrellas(X), razonable(X).  X = ricard_camarena.
```

- Ahora bien, si preguntamos aparentemente la misma meta:

```
1   ?- razonable(X), estrellas(X).  false.
```

Universidad Veracruzana

Soluciones

- Aproximaciones:
  1. Los programas como resúmenes de programas más extensos que pueden derivar las literales negativas;
  2. Redefinir la noción de consecuencia lógica, *e.g.*, la Well-Founded Semantics de Gelder, Ross y Schlipf [3].
- En ambos casos, el efecto es descartar algunos modelos del programa que no son de interés.
- Usaremos (1) para la NAF.

Compleción

- El programa deseado puede formalizarse como la compleción de Δ – Clark [2].
- Consideren la siguiente definición:

$$\begin{aligned} sobre(X, Y) &\leftarrow en(X, Y). \\ sobre(X, Y) &\leftarrow en(X, Z), sobre(Z, Y). \end{aligned}$$

- Esto también puede escribirse como:

$$sobre(X, Y) \leftarrow en(X, Y) \vee (en(X, Z), sobre(Z, Y))$$

- ¿Qué sucede si remplazamos la implicación por la equivalencia?

$$sobre(X, Y) \leftrightarrow en(X, Y) \vee (en(X, Z), sobre(Z, Y))$$

Efecto

- Está fbf expresa que X está sobre Y si y sólo si una de las condiciones es verdadera.
- Si ninguna de las condiciones se cumple, ¡se sigue que X no está sobre Y!
- Esta es la intuición seguida para justificar la negación por fallo.
- Ahora, combinar cláusulas definitivas como en el ejemplo anterior, sólo es posible para cláusulas con cabezas idénticas.

Ejemplo

- Asumamos Δ:

$$en(c, b).$$
$$en(b, a).$$

- Estas cláusulas puden escribirse como:

$$en(X_1, X_2) \leftarrow X_1 = c, X_2 = b$$
$$en(X_1, X_2) \leftarrow X_1 = b, X_2 = a$$

- Y combinarse en una sola fórmula:

$$en(X_1, X_2) \leftrightarrow (X_1 = c, X_2 = b) \vee (X_1 = b, X_2 = a)$$

- La compleción de Δ.

Compleción I

- Sea Δ un programa lógico definitivo.
- $comp(\Delta)$ denota la compleción de Δ, *i.e.*, el conjunto de fórmulas obtenido a partir de las siguientes tres transformaciones:
    1. Para cada símbolo de predicado ϕ remplazar cada cláusula:

$$\phi(t_1, \dots, t_m) \leftarrow \alpha_1, \dots, \alpha_n \quad (n \geq 0)$$

por la fórmula:

$$\phi(X_1, \dots, X_m) \leftarrow \exists Y_1, \dots, Y_i(X_1 = t_1, \dots, X_m = t_m, \alpha_1, \dots, \alpha_n)$$

donde las Y_i son variables que ocurren en la cláusula y las X_i son variable únicas que no.

Compleción II

2. Para cada símbolo de predicado ϕ remplazar todas las fbf:

$$\phi(X_1, \ldots, X_m) \leftarrow \beta_1$$
$$\vdots$$
$$\phi(X_1, \ldots, X_m) \leftarrow \beta_j$$

por la fórmula:

$$\forall X_1, \ldots, X_m(\phi(X_1, \ldots, X_m)) \leftrightarrow \beta_1 \vee, \ldots, \vee \beta_j \quad \text{Si } j > 0$$
$$\forall X_1, \ldots, X_m(\neg\phi(X_1, \ldots, X_m)) \quad \text{Si } j = 0$$

Compleción III

3. Extender Δ con los siguientes axiomas de igualdad libre:

$$\forall(X = X)$$
$$\forall(X = Y \rightarrow Y = X)$$
$$\forall(X = Y \wedge Y = Z \rightarrow X = Z)$$
$$\forall(X_1 = Y_1 \wedge \cdots \wedge X_n = Y_n \rightarrow f(X_1, \ldots, X_n) = f(Y_1, \ldots, Y_n))$$
$$\forall(X_1 = Y_1 \wedge \cdots \wedge X_n = Y_n \rightarrow (\phi(X_1, \ldots, X_n) \rightarrow \phi(Y_1, \ldots, Y_n))$$
$$\forall(f(X_1, \ldots, X_n) = f(Y_1, \ldots, Y_n) \rightarrow X_1 = Y_1 \wedge \cdots \wedge X_n = Y_n)$$
$$\forall(f(X_1, \ldots, X_m) \neq g(Y_1, \ldots, Y_n)) \quad \text{Si } f/m \neq g/n$$
$$\forall(X \neq t) \quad \text{Si } X \text{ es un sub-término de } t.$$

Ejemplo

- El primer paso produce:

$$\begin{aligned} sobre(X_1, X_2) &\leftarrow \exists X, Y\ (X_1 = X, X_2 = Y, en(X, Y)) \\ sobre(X_1, X_2) &\leftarrow \exists X, Y, Z\ (X_1 = X, X_2 = Y, en(Z, Y), sobre(Z, Y)) \\ en(X_1, X_2) &\leftarrow (X_1 = c, X_2 = b) \\ en(X_1, X_2) &\leftarrow (X_1 = b, X_2 = a) \end{aligned}$$

- Luego:

$$\begin{aligned} \forall X_1, X_2(sobre(X_1, X_2) \leftrightarrow \exists X, Y(\dots) \land \exists X, Y, Z(\dots)) \\ \forall X_1, X_2(en(X_1, X_2) \leftrightarrow (X_1 = c, X_2 = b) \land (X_1 = b, X_1 = a)) \end{aligned}$$

- Y el programa se termina con las definiciones de igualdad.

Universidad Veracruzana

Observaciones

- Si $\Delta \models \alpha$, entonces *comp*$(\Delta) \models \alpha$.
- Si *comp*$(\Delta) \models \alpha$, entonces $\Delta \models \alpha$.
- Por lo tanto, al completar el programa no agregamos información positiva al mismo, solo información negativa.
- Al substituir las implicaciones en Δ por equivalencias en *comp*(Δ) es posible inferir información negativa a partir del programa completado.

Robustez y Completitud

- Sea Δ un programa definitivo y $\leftarrow \alpha$ una meta definitiva:

  Robustez. Si $\leftarrow \alpha$ tiene un árbol-SLD finito fallido, entonces $comp(\Delta) \models \forall(\neg\alpha)$.

  Completitud. Si $comp(\Delta) \models \forall(\neg\alpha)$ entonces existe un árbol finito fallido para la meta definitiva $\leftarrow \alpha$.

- La robustez se preserva aún si α no es de base.
- Ejemplo: $\leftarrow en(a, X)$ falla de manera finita y por lo tanto, se sigue que $comp(\Delta) \models \forall(\neg en(a, X))$.

Derivaciones justas y Completitud

- El teorema de completitud no es válido para Prolog.
- La completitud funciona para una subclase de derivaciones-SLD conocidas como justas (*fair*), *i.e.*, cada átomo en la derivación sea seleccionado eventualmente.
- Un árbol-SLD es justo si todas sus derivaciones son justas.
- La *NAF* es completa para árboles-SLD justos.
- Este tipo de derivaciones se pueden implementar fácilmente: selecciona la sub-meta más a la izquierda y agrega nuevas submetas al final de esta, *i.e.*, búsqueda en amplitud.
- Prolog no implementa tal estrategia por razones de eficiencia.

Metas generales

- Combinando la resolución SLD y la NAF, es posible generalizar la noción de meta definitiva para incluir literales positivas y negadas.
- Una meta general tiene la forma:

$$\leftarrow \alpha_1, \ldots \alpha_n \quad (n \geq 0)$$

donde cada α_i es una literal positiva o negada.

Resolución-SLDNF

- Sea Δ un programa definitivo, G_0 una meta general y $\mathcal{R}$ una función de selección.
- Una derivación SLDNF de G_0 usando $\mathcal{R}$, es una secuencia finita o infinita de metas generales:

$$G_0 \overset{\alpha_0}{\leadsto} G_1 \ldots G_{n-1} \overset{\alpha_{n-1}}{\leadsto} G_n$$

donde $G_i \overset{\alpha_i}{\leadsto} G_{i+1}$ puede ocurrir si:

1. La literal $\mathcal{R}$-seleccionada en G_i es positiva y G_{i+1} se deriva de G_i y α_i por un paso de resolución SLD;
2. La literal $\mathcal{R}$-seleccionada en G_i es negativa ($\neg\alpha$) y la meta $\leftarrow \alpha$ tiene un árbol SLD fallido y finito y G_{i+1} se obtiene a partir de G_i eliminando $\neg\alpha$ (en cuyo caso α_i, corresponde al marcador especial FF).

Derivaciones completas

- Además de la refutación y de la derivación infinita, existen dos clases de derivaciones SLDNF completas dada una función de selección:
  1. Una derivación se dice (finitamente) fallida si (i) la literal seleccionada es positiva y no unifica con ninguna cabeza de las cláusulas del programa, o (2) la literal seleccionada es negativa y tiene un fallo finito.
  2. Una derivación se dice plantada (*stuck*) si la sub-meta seleccionada es de la forma $\neg\alpha$ y $\leftarrow \alpha$ tiene un fallo infinito.

Ejemplo I

- Considere el siguiente programa:

$$en(c, b)$$
$$en(b, a)$$

Ejemplo II

- La meta $\leftarrow en(X, Y), \neg en(Z, X)$ tiene una refutación-SLDNF con la substitución computada $\{X/c, Y/b\}$:

$$
\begin{aligned}
G &= \leftarrow en(X, Y), \neg en(Z, X). \\
G_0 &= \leftarrow en(X, Y). \\
\alpha_0 &= en(c, b). \\
\theta_0 &= \{X/c, Y/b\} \\
G_1 &= \neg en(Z, X)\theta_0 \;= \leftarrow en(Z, c) \\
\alpha_1 &= FF \\
\theta_1 &= \epsilon \\
G_2 &= \square \\
\theta &= \theta_0\theta_1 \;=\; \{X/c, Y/b\}
\end{aligned}
$$

Robustez

- Sea Δ un programa definitivo y $\leftarrow \alpha_1, \ldots, \alpha_n$ una meta general. Si $\leftarrow \alpha_1, \ldots, \alpha_n$ tiene una refutación SLDNF con una substitución computada θ, $comp(\Delta) \models \forall(\alpha_1\theta, \ldots, \alpha_n\theta)$.

Completitud

- Sin embargo, la resolución-SLDNF no es completa.
- Esto a pesar de que la resolución-SLD y la NAF si lo son.
- Ejemplo: $\leftarrow \neg en(X, Y)$ que corresponde a la consulta "¿Hay algunos bloques X e Y, tal que X no está en Y?"
- Uno esperaría varias respuestas a esta consulta, p. ej., el bloque *a* no está encima de ningún bloque, etc.
- Pero la derivación SLDNF de $\leftarrow \neg en(X, Y)$ falla porque la meta $\leftarrow en(X, Y)$ tiene éxito (puede ser demostrada).

Cláusulas y programas generales

Cláusula general. Una cláusula general es una fbf de la forma $\alpha_0 \leftarrow \alpha_1, \ldots, \alpha_n$ donde α_0 es una fbf atómica y $\alpha_1, \ldots, \alpha_n$ son literales ($n \geq 0$).

Programa general. Un programa lógico general es un conjunto finito de cláusulas generales.

Ejemplo

- Ahora podemos extender nuestro programa del mundo de los bloques con las siguientes relaciones:

$$
\begin{aligned}
base(X) &\leftarrow en(Y, X), en_la_mesa(X). \\
en_la_mesa(X) &\leftarrow \neg no_en_la_mesa(X). \\
no_en_la_mesa(X) &\leftarrow en(X, Y). \\
en(c, b). & \\
en(b, a). &
\end{aligned}
$$

Observaciones

- Aunque el lenguaje fue enriquecido, no es posible que una literal negativa sea consecuencia lógica de un programa dado.
- La razón es la misma que para los programas definitivos, la base de Herbrand de un programa Δ, B_Δ es un modelo de Δ en el que todas las literales negativas son falsas.
- Al igual que con los programas definitivos, la pregunta es entonces como lograr inferencias negativas consistentes.

Compleción al rescate

- Afortunadamente el concepto de compleción de programa puede aplicarse también a los programas lógicos generales.
- Ejemplo: La compleción de $gana(X) \leftarrow mueve(X, Y), \neg gana(Y)$ contiene la fbf:

$$\forall X_1(gana(X_1) \leftrightarrow \exists X, Y(X_1 = X, mueve(X, Y), \neg gana(Y)))$$

- Desafortunadamente, la compleción de los programas normales puede ocasionar paradojas.

Ejemplo

- Consideren la cláusula general $p \leftarrow \neg p$, su compleción incluye $p \leftrightarrow \neg p$.
- La inconsistencia del programa completado se debe a que $p/0$ está definida en términos de su propio complemento.
- Una estrategia de programación para evitar este problema consiste en componer los programas por capas o estratos, forzando al programador a referirse a las negaciones de una relación hasta que ésta ha sido totalmente definida.

Programas estratificados

- Un programa general Δ se dice estratificado si y sólo si existe al menos una partición $\Delta_1 \cup \cdots \cup \Delta_n$ de Δ tal que:

  1. Si $p(\dots) \leftarrow q(\dots), \dots \in \Delta_i$ entonces $\Delta^q \subseteq \Delta_1 \cup \cdots \cup \Delta_i$;
  2. Si $p(\dots) \leftarrow \neg q(\dots), \dots \in \Delta_i$ entonces $\Delta^q \subseteq \Delta_1 \cup \cdots \cup \Delta_{i-1}$.

- Donde Δ^q denota el conjunto de todas las cláusulas en Δ que tienen como cabeza q.

Ejemplo

- el siguiente programa está estratificado:
  Δ_2:

$$
\begin{array}{rcl}
base(X) & \leftarrow & en(Y, X), en_la_mesa(X). \\
en_la_mesa(X) & \leftarrow & \neg no_en_la_mesa(X).
\end{array}
$$

  Δ_1:

$$
\begin{array}{rcl}
no_en_la_mesa(X) & \leftarrow & en(X, Y). \\
en(c, b). & & \\
en(b, a). & &
\end{array}
$$

Observaciones

- Apt, Blair y Walker [1] demostraron que la compleción de un programa estratificado en consistente.
- La estratificación es solo una condición suficiente para la consistencia.
- Determinar si un programa está estratificado es decidible, pero determinar si la compleción de un programa general es consistente, es indecidible.
- Hay programas generales no estratificados cuya compleción es robusta.

Ideas

- Hemos revisado el caso de la resolución-SLDNF entre programas definitivos y metas generales.
- Informalmente podemos decir que la resolución-SLDNF combina la resolución-SLD con los siguientes principios:
  1. $\neg\alpha$ tiene éxito ssi $\leftarrow \alpha$ tiene un árbol-SLD finito que falla.
  2. $\neg\alpha$ falla finitamente ssi $\leftarrow \alpha$ tiene una refutación-SLD.

Problemas

- El paso de programas definitivos a programas generales, es un poco complicado.
- Para probar $\neg\alpha$, debe de existir un árbol finito fallido para $\leftarrow \alpha$.
- Tal árbol puede contener nuevas literales negativas, las cuales a su vez deben tener éxito o fallar finitamente.
- Esto complica la definición de la resolución-SLDNF para programas generales.

Ejemplo

- Paradojas cuando los predicados están definidos en términos de sus propios complementos:

$$\alpha \leftarrow \neg\alpha$$

- Dada la meta inicial $\leftarrow \alpha$ se puede construir una derivación $\leftarrow \alpha \leadsto \leftarrow \neg\alpha$.
- La derivación puede llegar a una refutación si $\leftarrow \alpha$ falla finitamente.
- De manera alternativa, si $\leftarrow \alpha$ tiene una refutación, entonces la derivación falla.
- Helas! esto es imposible pues la meta $\leftarrow \alpha$ no puede tener una refutación y fallar finitamente al mismo tiempo.

Universidad Veracruzana

Bosque SLDNF I

- Sea Δ un programa general, G_0 una meta general, y $\mathcal{R}$ una función de selección.
- El bosque-SLDNF de G_0 es el bosque más pequeño (considerando el posible renombrado de variables), tal que:
  1. G_0 es la raíz del árbol.
  2. Si G es un nodo en el bosque cuya literal seleccionada es positiva, entonces para cada cláusula α tal que G' puede ser derivada de G y α (con UMG θ), G tiene un hijo etiquetado como G'. Si no existe tal cláusula, entonces G tiene un hijo etiquetado FF (falla finita);

Bosque SLDNF II

3. Si G es un nodo del bosque cuya literal seleccionada es de la forma $\neg\alpha$ (G es de la forma $\leftarrow \alpha_1, \dots, \alpha_{i-1}, \neg\alpha, \alpha_{i+1}, \dots, \alpha_n$), entonces:
    - El bosque contiene un árbol cuya raíz está etiquetada como $\leftarrow \alpha$;
    - Si el árbol con raíz $\leftarrow \alpha$ tiene una hoja $\Box$ con la substitución computada ϵ, entonces G tiene un sólo hijo etiquetado FF;
    - Si el árbol con raíz $\leftarrow \alpha$ es finito y tiene todas sus hojas etiquetadas FF, entonces G tiene un sólo hijo (con substitución asociada ϵ) etiquetado como $\leftarrow \alpha_{\mathbf{1}}, \dots, \alpha_{i-\mathbf{1}}, \alpha_{i+\mathbf{1}}, \dots, \alpha_n$.

Observaciones

- Observen que la literal negativa seleccionada $\neg\alpha$ falla sólo si $\leftarrow \alpha$ tiene una refutación con la substitución computada vacía ϵ.
- Como veremos más adelante, esta condición que no era necesaria cuando definimos la resolución-SLDNF para programas definitivos, es vital para la correctez de esta resolución en los programas generales.

Nomenclatura

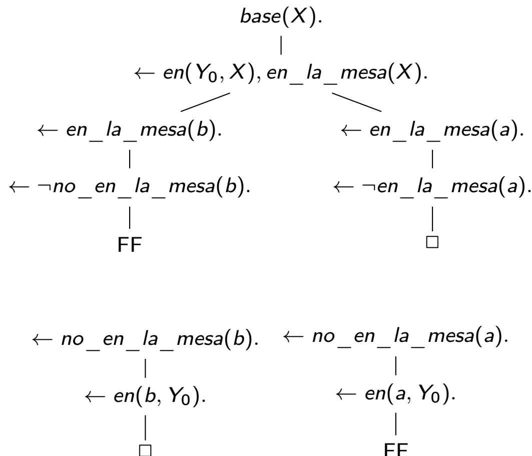
- Los arboles del bosque-SLDNF son llamados arboles-SLDNF completos;
- y la secuencia de todas las metas en una rama de un árbol-SLDNF con raíz G es llamada derivación-SLDNF completa de G (bajo un programa Δ y una función de selección $\mathcal{R}$).
- El árbol etiquetado por G_0 es llamado árbol principal.
- Un árbol con la raíz $\leftarrow \alpha$ es llamado árbol subsidiario si $\neg\alpha$ es una literal seleccionada en el bosque (el árbol principal puede ser a su vez subsidiario).

Ejemplo

- Consideren el siguiente programa general estratificado Δ:

$$
\begin{aligned}
base(X) &\leftarrow en(Y, X), en_la_mesa(X).\\
en_la_mesa(X) &\leftarrow \neg no_en_la_mesa(X).\\
no_en_la_mesa(X) &\leftarrow en(X, Y).\\
encima(X, Y) &\leftarrow en(X, Y).\\
encima(X, Y) &\leftarrow en(X, Z), encima(Z, Y).\\
en(c, b).&\\
en(b, a).&
\end{aligned}
$$

Bosque-SLDNF



Derivaciones

- Las ramas de un árbol-SLDNF completo representan todas las derivaciones-SLDNF completas de su raíz.
- Dada nuestra función de selección, hay cuatro clases de derivaciones-SLDNF completas:
  1. Derivaciones infinitas;
  2. Derivaciones finitas fallidas (terminan en FF);
  3. Refutaciones (terminan en $\Box$); y
  4. Derivaciones plantadas (si ninguno de los casos anteriores aplica).

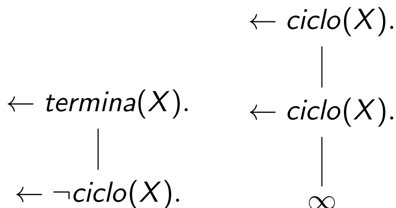
Ejemplo

- Consideren el siguiente programa:

$$\begin{aligned} termina(X) &\leftarrow \neg ciclo(X). \\ ciclo(X) &\leftarrow ciclo(X). \end{aligned}$$

Bosque para $\leftarrow termina(X)$.

- El árbol de la izquierda está plantado. El de la derecha está en una derivación infinita.



Ejemplo

- El siguiente programa también conduce a una derivación plantada (definición circular):

$$
\begin{aligned}
paradoja(X) &\leftarrow \neg ok(X). \\
ok(X) &\leftarrow \neg paradoja(X).
\end{aligned}
$$

- El bosque:

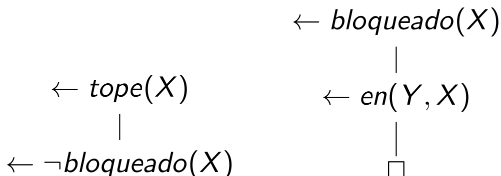
$$
\begin{array}{ccc}
\leftarrow paradoja(X) & \quad & \leftarrow ok(X) \\
| & & | \\
\leftarrow \neg ok(X) & & \leftarrow \neg paradoja(X)
\end{array}
$$

Ejemplo

- La última razón para que una derivación quede plantada es ilustrada por el siguiente programa:

$$\begin{aligned} tope(X) &\leftarrow \neg bloqueado(X). \\ bloqueado(X) &\leftarrow en(Y, X). \\ en(a, b). \end{aligned}$$

- El bosque:



Prolog no es robusto

- $tope(a)$ debería poder derivarse de este programa. Sin embargo, el árbol-SLDNF de la meta $\leftarrow tope(X)$ no contiene refutaciones.
- De hecho, esta meta se planta aún cuando $\leftarrow bloqueado(X)$ tiene una refutación.
- La razón de esto es que $\leftarrow bloqueado(X)$ no tiene ninguna derivación que termine con una substitución computada vacía.
- A la meta $\leftarrow \neg tope(X)$, Prolog no responde b, sino que ¡no todos los bloques están en el tope de la pila!
- De manera que la implementación de la resolución-SLDNF en la mayoría de los Prolog no es robusta, aunque nuestra definición si lo es.

Correctez

- Sea Δ un programa general y $\leftarrow \alpha_1, \ldots, \alpha_n$ una meta general. Entonces:
  - Si $\leftarrow \alpha_1, \ldots, \alpha_n$ tiene una substitución de respuesta computada θ, entonces $comp(\Delta) \models \forall(\alpha_1\theta \wedge \cdots \wedge \alpha_n\theta)$.
  - Si $\leftarrow \alpha_1, \ldots, \alpha_n$ tiene un árbol-SLDNF finito que falla, entonces $comp(\Delta) \models \forall(\neg(\alpha_1 \wedge \cdots \wedge \alpha_n))$.

Observaciones

- La definición de bosque-SLDNF no debe verse como una implementación de la resolución-SLDNF, sólo representa el espacio ideal donde la correctez puede ser garantizada.
- Al igual que en los demás casos, Prolog sigue una estrategia primero en profundidad: Ante una meta $\neg\alpha$, Prolog suspende la construcción del árbol de resolución en espera de la respuesta a la meta $\leftarrow \alpha$.
- El hecho de que Prolog no verifique si la substitución de respuesta de una refutación fue ϵ, al igual que la ausencia de chequeo de ocurrencias, contribuyen a que Prolog no sea robusto.

Referencias I

[1] KR Apt, HA Blair y A Walker. "Towards a theory of declarative knowledge". En: (1988), págs. 89-148.

[2] K Clark. "Negations as Failure". En: *Logic and Databases*. Ed. por H Gallaire y J Minker. New York, USA: Plenum Press, 1978, págs. 293-322.

[3] A van Gelder, KA Ross y JS Schlipf. "The Well-Founded Semantics for General Logic Programs". En: *Journal of the Association for Computing Machinery* 38.43 (1991), págs. 620-650.

[4] U Nilsson y J Maluszynski. *Logic, Programming and Prolog*. 2nd. John Wiley & Sons Ltd, 2000.