0. Introducción

0.1. Caracterización y perspectiva del agente

Conceptos a conocer:

- **Sistema inteligente**: agente inteligente/racional.
- **Agente**: entidad que percibe su entorno a través de sensores y lo modifica a través de actuadores.
- **Agente racional**: aquel que realiza lo correcto en términos de función y maximiza su rendimiento.
- **Función del agente**: lista de tareas a realizar por el agente.
- **Programa del agente**: parte programada del agente que le permite realizar acciones y percibir por sus sensores.
- **Acciones del agente**: medios que tiene para modificar su entorno o a sí mismo (moverse, limpiar, pararse…).
- **Percepción**: información recibida por un sensor del agente.
- **Entorno**: colección de elementos perceptibles. Se clasifican como:
    - **Discreto/continuo**: es continuo si tiene una serie teóricamente infinita de “estados” intermedios. Es discreto si los estados son contables (casillas de ajedrez).
    - **Estático/dinámico/semidinámico**: es dinámico si el entorno puede cambiar mientras toma una decisión. En caso contrario, es estático. Es semidinámico si el entorno no cambia pero el rendimiento cambia con el paso del tiempo.
    - **Determinista/estocástico**: es determinista si el estado actual del medio depende del anterior y de la acción ejecutada por el agente (o si el próximo estado del medio es predecible). Al contrario, si el próximo estado del medio no es predecible o no depende totalmente de las acciones del agente, es estocástico.
    - **Único agente/multiagente**: define si actúan uno o varios agentes en el mismo entorno.
    - **Episódico/secuencial**: es episódico si cada decisión es independiente del resto de decisiones. Es secuencial si, al contrario, las decisiones anteriores influyen en la decisión que debe tomar ahora o, dicho de otro modo, el agente actúa pensando también en futuro (ajedrez).
    - **Totalmente/parcialmente observable**: es parcialmente observable si el agente tiene que actuar con solo una parte del medio (información que sus sensores son capaces de recibir). Es totalmente observable si recibe toda la información que se puede adquirir del medio.

- **Rendimiento**: medida que cuantifica el grado de “éxito”. Es objetiva y externa, dado que la establece el programador y se aplica sobre el entorno que observa.

0.1.1. Tipos de agentes: estructura del agente

Dependiendo del programa utilizado por el agente, se pueden diferenciar cuatro tipos de agentes que, después, pueden convertirse en agentes que aprendan.

0.1.1.1. Agentes reactivos simples

“Reaccionan ante estímulos simples”

Realiza acciones tomando en cuenta solo las percepciones del *presente*, ignorando las percepciones que ocurriesen en el pasado o lo que pueda ocurrir en el futuro.

Un dron que tome fotografías de un terreno y lo caracterice es un ejemplo de esta estructura. Este simplemente se va a mover de terreno en terreno, tomará la foto y le dará un valor a ese terreno.

Características:

- Se basa en reglas del tipo *si esto entonces lo otro*.
- Actúan solo en medios totalmente observables.

0.1.1.2. Agentes reactivos basados en modelos

“Guardan un modelo de lo que no ven para reaccionar en torno a él”

Mantienen unos estados internos que van cambiando según los estímulos recibidos (teniendo en cuenta los pasados, por eso los estados) y que sirven para compensar los estímulos que no se pueden recibir en un medio que no sea totalmente observable.

La idea básica es que de esta manera retiene un “historial” de información. Conociendo la forma o “modelo” con el que funciona el mundo, ese historial puede ayudar al agente a tomar una decisión más eficaz o correcta.

0.1.1.3. Agentes basados en objetivos o metas

“En función de lo que quiero hacer tomará una decisión”

El agente tiene información sobre su estado actual, pero también sobre la meta deseada. De esta manera, puede realizar una búsqueda entre las diferentes acciones a tomar y seleccionar la mejor para conseguir su objetivo.

Un taxi que puede tomar diferentes caminos en un cruce es un ejemplo. Tiene dos caminos a tomar, pero uno puede llevarle a su destino y otro no. Debido a que conoce la ubicación de su meta tomará el camino que sí le lleva a su destino.

0.1.1.4. Agentes basados en utilidad

“De todas las acciones, escogerá la más útil”

En base a una función de rendimiento o “utilidad”, el agente tomará las acciones mejores para alcanzar su objetivo con el mejor rendimiento posible.

Por ejemplo, el taxi puede tomar varios caminos para llegar a su objetivo, pero escogerá siempre el camino más corto, rápido y seguro debido a su función de rendimiento.

0.1.1.5. Agentes que aprenden

“Observa y discurre cómo puede mejorar”

Posee varios elementos más que cualquiera de los agentes anteriores. Pero no viene información en las diapositivas (y creo que no cae).

0.2. Principales paradigmas de sistemas inteligentes basados en conocimientos

Los sistemas inteligentes basados en conocimientos son sistemas que explotan al máximo la información almacenada o “base de conocimientos” a través de un código llamado “motor de inferencias”, que puede actuar bajo diferentes paradigmas. Esta base de conocimientos la suele producir un usuario experto en el dominio (ingeniero del conocimiento) diferente a los programadores de la interfaz o del propio motor de inferencias.

Lo diferenciamos del sistema inteligente “a secas”, que es el que simplemente optimiza su función de rendimiento, porque este no calcula la solución (mediante un algoritmo), sino que la razona mediante el motor de inferencias.

Estos sistemas inteligentes representan su conocimiento a través de unas “reglas de producción”, que generan conclusiones (conocidos como consecuentes) a partir de hechos (conocidos como antecedentes). Estos suelen tener un formato similar a este:

```
si antecedente1 y antecedente2
entonces consecuente
```

Nosotros debemos conocer cuatro tipos de sistemas inteligentes basados en conocimientos:

0.2.1. Sistema experto

“Intenta suplantar a un experto en la materia”.

Cuando un sistema inteligente basado en conocimientos lo modela un experto en la materia, usualmente se conoce a este sistema como sistema experto.

Este experto otorgará los conocimientos para realizar las asociaciones (heurística) adecuadas.

0.2.2. Razonamiento basado en casos o CBR (Case Based Reasoning)

“Si en este caso ocurrió A, en este caso similar también ocurrirá”

En vez de almacenar todo el conocimiento y toda la heurística del dominio, este razonamiento almacena casos (caso = problema + solución) de manera que, cuando reciba un problema, pueda buscar el caso más similar y devuelva la solución de ese caso adaptada (si es necesario) al problema recibido. Si el caso nuevo es diferente y suficientemente interesante, se puede añadir a la base de conocimiento, “aprendiendo” así un caso más.

0.2.3. Métodos de aprendizaje y minería de datos

“A partir de lo que le des te clasificará información”

Es capaz de inferir nuevo conocimiento mediante la observación de los datos:

- Aprende conceptos a partir de ejemplos etiquetados (clasificación).
- Descubre conceptos a partir de ejemplos no etiquetados (clustering).

0.2.4. Razonamiento basado en modelos

“La información recibida debe pertenecer a uno de los modelos almacenados”

Estructura el conocimiento en modelos (cómo funcionan las cosas). Uno de estos sistemas razonará manipulando los modelos hasta alcanzar la solución.

0.4. Bibliografía

- Stuart J. Russell: Inteligencia Artificial. Un Enfoque Moderno.

1. Estrategias de Resolución

1.1. Conceptos

- **Cláusula de Horn**: disyunción de literales con un máximo de un solo literal positivo

  $$(a \land b \land c) \rightarrow x \equiv \neg(a \land b \land c) \lor x \equiv \neg a \lor \neg b \lor \neg c \lor x$$

  Se suele representar al revés

  $$x \leftarrow (a, b, c)$$

  - **Pregunta**, **meta**, objetivo o cláusula sin cabeza: aquellas que no tienen literal positivo:

    $$\leftarrow (a, b, c)$$

  - **Hecho**: aquellas que son solo literal positivo:

    $$x \leftarrow$$

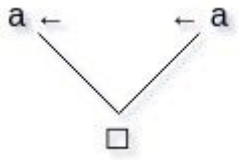
  - **Regla**: aquellas con literal positivo y negativo. Engloban a todos los demás (regla sin positivos, regla sin negativos, regla sin positivos ni negativos):

    $$x \leftarrow (a, b, c)$$

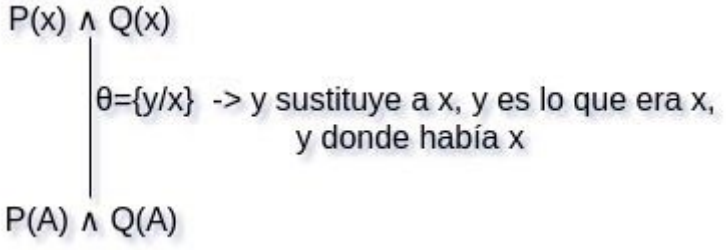
  - **Éxito**: aquellas con el cuerpo y cabeza vacíos. En ICon se interpreta como cláusula vacía:

    $$\leftarrow \equiv \square$$

- **Resolución SLD** (**S**election-rule driven **L**inear resolution for **D**efinite clauses): consiste en eliminar cláusulas positivas (izquierda de la flecha) con las negativas (derecha de la flecha).



- **Sustitución**: en lógica de primer orden, cambiar la variable por una constante u otra variable en todas las partes de una fórmula bien formada. En clase se suele representar cada sustitución con el símbolo θ.



- **FNC** (entre nosotros, **F**orma **N**o **C**ompleja): descomposición de todos los axiomas de forma que solo queden en él los símbolos $\wedge$, $\vee$ y $\neg$.
- **LPO** (**L**ógica de **P**rimer **O**rden): conjunto de fórmulas en que los símbolos son del tipo P(x). No valen ni del tipo x (a solas) ni del tipo P(P(x)).
- Una **estrategia** es **completa** si garantiza que encontrará la cláusula vacía en un conjunto de cláusulas inconsistente. Se dice que la regla de resolución es completa para la refutación.
- Una **estrategia** es **eficiente** si garantiza que encontrará la cláusula vacía en el menor número de pasos posible.
- **Resolvente de X e Y**: resultado de combinar X e Y.
- **FNP** (**F**orma **N**ormal **P**renexa): En lógica de primer orden, una fórmula bien formada tiene forma normal prenexa si está escrita encabezada por una cadena de cuantificadores existenciales o universales, seguidos por una fórmula sin cuantificadores lógicos, designada como matriz.

1.2. Refutación por resolución

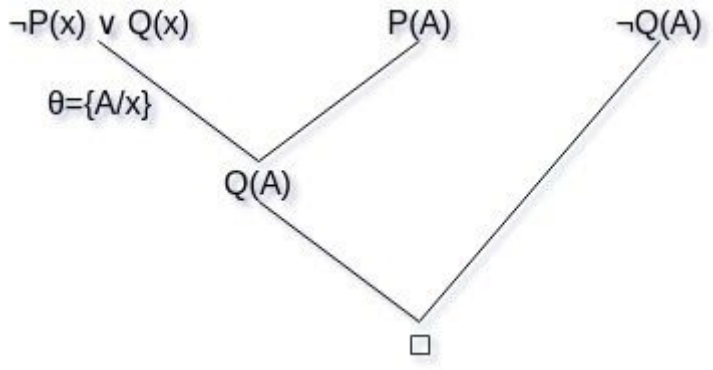
Para probar que $\boldsymbol{\Omega \Rightarrow \alpha}$, demostrar que no se cumple $\boldsymbol{\Omega \Leftarrow \alpha}$ mediante la forma

$$\boldsymbol{\Omega \vee \neg \alpha}.$$

donde Ω es nuestro conjunto de cláusulas del tipo FNC al que llamaremos “teoría sólida” (como lo llama en los apuntes).

Al negar α lo transformamos a FNC, y todas las cláusulas obtenidas las añadimos al conjunto S, en el que se encontrarán también las cláusulas del conjunto Ω.

A partir de aquí, construiremos un árbol en que se vayan eliminando elementos negativos con sus equivalentes positivos. Suponiendo que el conjunto S nos ha quedado como S={¬P(x) ∨ Q(x) , P(A) , ¬Q(A)} el árbol de resolución sería:



Aclaración: si se tiene por un lado ¬ y por otro $\neg p \vee q$, se obtienen dos cláusulas:
$p \vee \neg p$ y $\neg q \vee q$.

1.3. Estrategias de resolución

1.3.1. Estrategia de dirección

- **Saturación por niveles**: tomando como conjunto de partida el conjunto $S^0=S$, la saturación por niveles combina todas las cláusulas de los anteriores niveles para obtener las del siguiente nivel. Esto es, si *S0={a, b}* y se obtiene *S1={c}*, para obtener *S2* habrá que combinar *a* con *c* y *b* con *c* y así sucesivamente hasta obtener la cláusula vacía. Es **completa** e **ineficiente**.

1.3.2. Estrategias de simplificación

- **Eliminación de literales puros**: siendo un literal puro del conjunto S un literal (por ejemplo P(x) o p) que no tiene complementario, esta estrategia consiste simplemente en eliminar la cláusula entera. Es **completa**. Ejemplo de eliminación de literales puros:

$$S=\{a, a \vee b, \neg b, b\}$$

a es un literal puro, así que se puede eliminar las cláusulas en las que aparece:

$$S=\{\neg b, b\}$$

Conviene decir que esto solo se puede hacer si no se puede obtener dicho literal mediante una sustitución de variables (en LPO).

- **Eliminación de tautologías**: eliminación de aquellas cláusulas que den como resultado una tautología (por ejemplo, $a \vee \neg a$). Es **completa**.
- **Eliminación de cláusulas subsumidas**: eliminación de cláusulas al obtener una cláusula que sea parte de la primera o equivalente. Por ejemplo:

  $$1\text{- } p \vee q \qquad \text{y} \qquad 2\text{- } p$$

  como 2 es una parte de 1, la subsume a.k.a. la elimina. Existen dos tipos:
  - Eliminación **hacia delante**: descartar una cláusula recién obtenida por tener una más simple anteriormente obtenida.
  - Eliminación **hacia atrás**: descartar una cláusula anterior al obtener una cláusula más simple.

Completa con saturación por niveles, incompleta con alguna estrategia de restricción (¿?), **eficiente**.

- **Asociación de procedimientos**: dados unos literales básicos o unas funciones conocidas, la asociación de procedimientos consiste en evaluar (es decir, resolver) dichos elementos para simplificar las cláusulas. Por ejemplo, teniendo *+(1, 2)*, podríamos simplificarlo como *3* (solo si forma parte del dominio). En estos casos se debe aclarar con qué procedimiento hemos

asociado la función, porque dependiendo de cómo lo hayamos interpretado (interpretación parcial), puede concluir en uno u otro resultado. **No es completa**.

- **Asociación de funciones**: Asociar un símbolo de función con un procedimiento cuya evaluación (resolución) **devuelva un elemento del dominio**. Por ejemplo (de los apuntes):

  Q(suma(3,5), y)

  ≡

  Q(8, y) (con la interpretación habitual de la suma)

- **Asociación de literales**: De manera similar a la anterior, se asocia un símbolo de (esta vez) predicado cuya evaluación **devuelva verdadero o falso** en este caso. Por ejemplo (de los apuntes):

  MAYOR(3,5)

  ≡

  Falso (asumiendo que la evaluación (V) de *MAYOR(3,5) = F*)

1.3.3. Estrategias de restricción

- **Conjunto soporte**: considerando el conjunto ya mencionado S y un subconjunto (estrictamente menor, es decir, no puede ser igual que S) T diferente del conjunto vacío, denominaremos a este T como conjunto soporte de S.
  - Este conjunto soporte original lo llamaremos conjunto soporte de nivel 0 o T_0.
  - Los siguientes niveles (conjuntos i-ésimos) los obtendremos combinando las cláusulas k_T del nivel anterior (que queramos) con las cláusulas k_S de S, sin necesidad de que sean de distinto grupo ni de hacer todas las combinaciones (es decir, podemos combinar k_T+k_T, k_S+k_T (y viceversa) y k_S+k_S).
  - En ningún caso se puede, para obtener T_i, utilizar una cláusula que no pertenezca a S o a T_{i-1}.

  Dicho esto, la estrategia del conjunto soporte utiliza un conjunto soporte y sus sucesivos niveles para llegar a la cláusula vacía.

  Para formar S, hay que añadir a la teoría (las cláusulas con las que se quiere demostrar algo) el teorema negado (lo que se quiere demostrar, negado) como se ha hecho hasta ahora.

Para formar T (según los apuntes), se recomienda coger solo las cláusulas que provienen de la negación del teorema (tomado de la solución de los ejercicios de teoría, en los apuntes viene mal).

- **Resolución lineal**: en cierto modo similar al conjunto soporte. Teniendo el conjunto S, elegimos una de sus cláusulas como cláusula central (y la llamaremos C_0). Obligatoriamente combinaremos esta cláusula con otra, cuyo resultado llamaremos C_1.
    - C_i será lo que llamaremos cláusula central (de nivel) i.
    - Las cláusulas de S que no sean C_0 las llamaremos cláusulas laterales o B
    - Cualquier combinación que hagamos será necesariamente de la última cláusula central C_i con otra cláusula central $C_{j<i}$ o una cláusula lateral B hasta llegar al conjunto vacío.

  Para elegir la cláusula inicial, se escoge la cláusula del teorema negado (suponiendo que solo da una cláusula. En caso contrario, habría que usar conjunto soporte).
    - **Teorema de complitud**: si S sin la cláusula k es consistente, pero al añadir dicha cláusula se genera una inconsistencia, se garantiza que si se usa esa cláusula k como cláusula central C_0 se alcanzará la cláusula vacía.
- **Resolución por entradas**: llamando cláusulas de entrada E a aquellas cláusulas pertenecientes al conjunto S, y resolventes de entrada R a aquellas cláusulas formadas como resultado de combinar una cláusula cualquiera con una E, la resolución por entradas es aquella que se obtiene únicamente a partir de cláusulas R. Dado que una cláusula R puede tener como padres una R y una E, esta resolución engloba también a la resolución lineal. **No es completa**.
- **Resolución unitaria**: llamando resolvente unitario U a aquellas cláusulas que tienen al menos un padre unitario (es decir, solo es un elemento sin ninguna disyunción $\vee$), la resolución unitaria es aquella que solo nos permite combinar cláusulas que den como resolvente una cláusula U. **No es completa**.
    - **Teorema de equivalencia** de las resoluciones unitaria y por entradas: si con una estrategia se puede llegar al conjunto vacío, con la otra también se puede.
    - **Teorema de complitud**: si todas las cláusulas de S son cláusulas de Horn y, mediante la resolución unitaria o por entradas, se llega a la cláusula vacía, se deduce que S es inconsistente.

1.4. Procedimiento de extracción de respuesta

Proceso de encontrar los elementos del dominio que hacen cierto el teorema a demostrar, mediante una prueba de refutación por resolución. Está formado por axiomas, <<preguntas>> y <<literales respuesta>>:

- **Axiomas**: cláusulas inicialmente representadas en LPO que representan los <<conocimientos que tenemos del dominio>>.
- **Preguntas**: sentencias en FNP que usan solo cuantificadores existenciales (∃) y cuyas matrices son conjunciones (∧) de literales. Esto se hace por dos razones:
    - Al negar los prefijos existenciales, se obtiene una sola cláusula de disyunciones (∨). Ej.:

Preguntamos: $(\exists x)(Dragon(x) \wedge Vuela(x))$

Negamos el prefijo: $\neg(\exists x)(Dragon(x) \wedge Vuela(x))$

$(\forall x)\neg(Dragon(x) \wedge Vuela(x))$

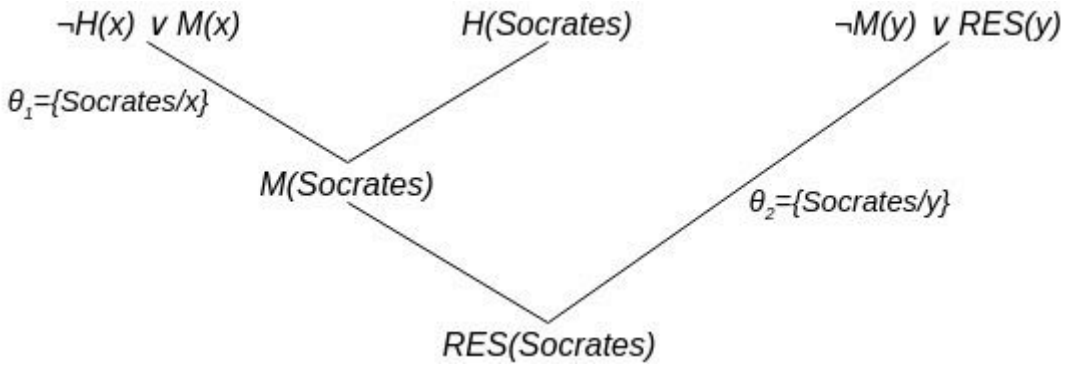
$(\forall x)(\neg Dragon(x) \vee \neg Vuela(x))$

Eliminamos el cuant.: $\neg Dragon(x) \vee \neg Vuela(x)$

    - Es más fácil de leer: ¿Existe algo que sea un dragón y vuele?

- **Literal respuesta**: literal único representado como $RESP(x_1, x_2...)$ que contiene todas las variables contenidas en la pregunta (si la pregunta no tiene variables, RESP tampoco. Las constantes no cuentan como variables). Este literal irá junto a la pregunta (por ejemplo, $(\exists x)(Dragon(x) \wedge Vuela(x) \wedge RESP(x))$) y, en este tipo de problemas, buscaremos aislar el literal respuesta (en vez de buscar el conjunto vacío). Cuando quede aislado, nos dirá si la respuesta es una constante o es una variable (es decir, cualquier respuesta).

Ejemplo de extracción de respuesta:



Se puede producir más de un literal respuesta, y se pueden producir varias respuestas (dependiendo de las sustituciones que se hagan).

1.5. Bibliografía

- Asignatura de “Lógica” de la Escuela Universitaria de Ingeniería Técnica Informática de Oviedo

2. Programación Lógica

2.1. Conceptos

- **Término**: elemento más básico de la sintaxis:
  - Constantes numéricas (1, 50, 597…) o atómicas (A, B, Algo, Alguien...)
  - Variables: x, y, a, x_1, x_2…
  - Funciones (que contienen términos): f(a, x), g(x), padre(Ana)...
- **Átomos**: expresiones de la forma $P(t_1, t_2, ... ,t_n)$, siendo P un símbolo de predicado de grado n y $t_1, t_2, ... ,t_n$ términos.
- **Cláusula**: construcción mediante operaciones de átomos.
- **Cláusulas definidas**: en programación lógica, cláusulas de Horn ($x \leftarrow (a, b, c)$, donde x es la cabeza de la cláusula y (a, b, c) es el cuerpo de la cláusula; ver conceptos de Estrategias de Resolución), donde los elementos (x, a, b, c) son llamados átomos y todas las variables se consideran cuantificadas universalmente.
- **Programa**: conjunto finito de fórmulas bien formadas. Se pueden diferenciar tres tipos:
  - Programa (al uso): cualquier tipo de FBF.
  - Programa normal: utiliza cláusulas de Horn extendidas (con literales negativos en el cuerpo de la cláusula).
  - Programa definido: utiliza cláusulas de Horn.
- **Computación**: obtención de pruebas formales. ***reformular***
- **P ∪ {G}**: aplicar la meta G al programa P.
- **θ =** ∅: sustitución vacía. No hace falta aplicar sustituciones y/o vale cualquier sustitución.
- **Derivar *a***: siendo *a* una meta, se dice que se deriva *a* si, usándolo con el programa P, se llega a la cláusula vacía.
- **Sintaxis de Edimburgo**: establece la sintaxis a utilizar en un programa definido en Prolog.
- **Programa definido**: conjunto de hechos y reglas que describen explícitamente qué es cierto, sin información explícita sobre qué es falso.
- **Rama finita**: caso en la resolución SLD en que se puede derivar un número finito de veces.
- **Rama fallo**: rama finita que no termina en la cláusula vacía
- **Rama infinita**: caso en la resolución SLD en que no existe una derivación última (siempre hay una cláusula con la que se puede derivar).

2.2. Resolución SLD

Tipo de resolución lineal que utiliza las cláusulas de Horn. Repasando:

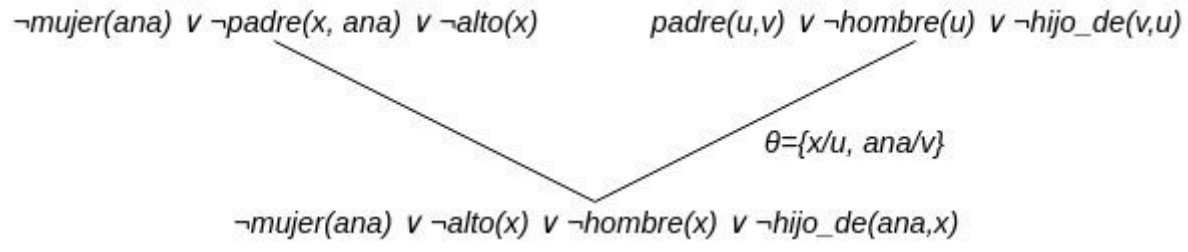
- Pregunta o meta: $\leftarrow (g_1, g_2, g_3, g_4, \ldots g_n)$
- Hecho: $t \leftarrow$
- Regla: $t \leftarrow (c_1, c_2, c_3, c_4, \ldots c_m)$

Denominaremos G a la pregunta y C a las cláusulas del programa (hay que procurar renombrar sus variables si coinciden con las de G).

La **función de selección f_S** es la decisión tomada o el algoritmo usado (regla de cómputo) para seleccionar un elemento de la meta. En un caso general, este elemento (que denominaremos g_S) será el equivalente negativo a (lo que en el ejemplo hemos llamado) *t*, de forma que al combinar G con C se irán ambos elementos. La función de selección concreta que nos devuelve g_S se indica como $f_S(G)$.

La salida de una resolución SLD se denomina **resolvente SLD** de G y C y, en un caso sencillo en que solo se usen esas dos reglas, se forma de la siguiente manera:

1. Cogemos todos los elementos de G en el mismo orden:
$$\leftarrow (g_1 , g_2 , \ldots g_{S-1} , g_S , g_{S+1} , \ldots g_n)$$
2. Eliminamos el elemento g_S:
$$\leftarrow (g_1 , g_2 , \ldots g_{S-1} , g_{S+1} , \ldots g_n)$$
3. En la ubicación donde estaba g_S, introducimos los elementos de C(omitiendo el elemento *t*) en el mismo orden:
$$\leftarrow (g_1 , g_2 , \ldots g_{S-1} , c_1 , c_2 , \ldots c_m, g_{S+1} , \ldots g_n)$$
4. Una vez tenemos esto, aplicamos el unificador general fruto de las sustituciones realizadas en el árbol de resolución (como se vio en el tema anterior):
$$\leftarrow (g_1 , g_2 , \ldots g_{S-1} , c_1 , c_2 , \ldots c_m, g_{S+1} , \ldots g_n)\theta$$
Este último paso unificaría las variables de la lógica de primer orden utilizadas, y se obtiene (al igual que la resolvente) realizando el árbol de resolución (ejemplo sacado de los apuntes):

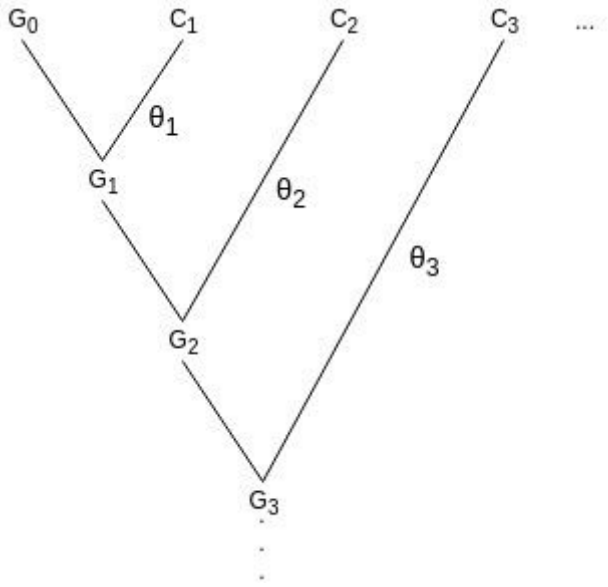


Transformando a cláusula de Horn:

← *(mujer(ana), hombre(x), hijo_de(ana, x), alto(x))*

2.2.1. Derivación SLD

Tomando todo lo anterior, supongamos que la resolvente obtenida es una meta a la que llamaremos G_1 y que tenemos más cláusulas C (C_1, C_2…), es decir, el programa tiene más cláusulas. La idea, representada de manera sencilla, de la derivación SLD es la siguiente:



La derivación SLD es una estrategia de resolución lineal y por entradas. Dicho en los mismos apuntes, G_{i+1} es el resolvente SLD de G_i y C_{i+1} usando θ_{i+1}.

Por último, se denomina **refutación SLD** a la derivación SLD que termina en el conjunto vacío. Esto es, el último G_i será igual a □.

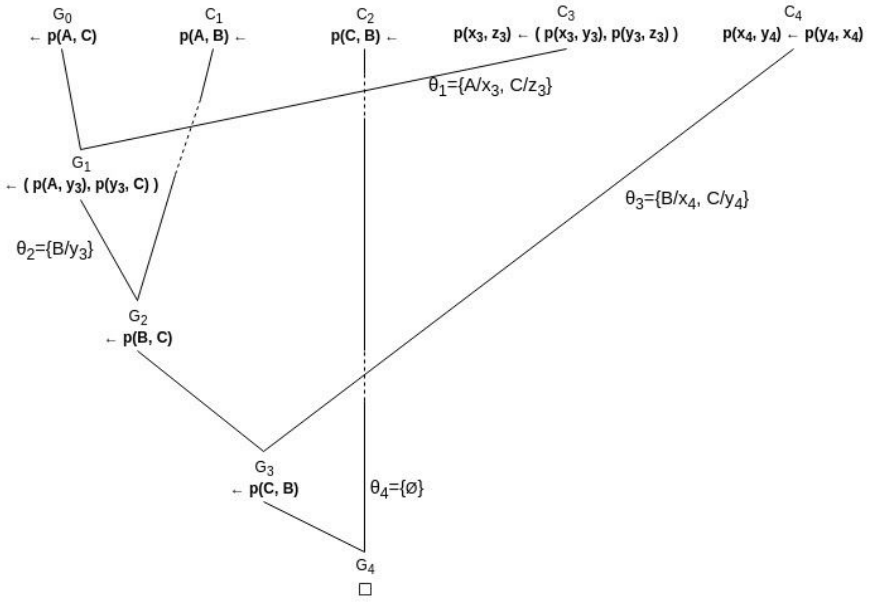
2.3. Intérprete abstracto de un programa lógico

El intérprete de un programa lógico es un conjunto de rutinas, acciones y tareas (función de programación informática) que busca las diferentes interpretaciones del programa lógico. Lo denominaremos abstracto (esto no es seguro) porque no es un método fijo, sino que hay varias formas de implementar este intérprete.

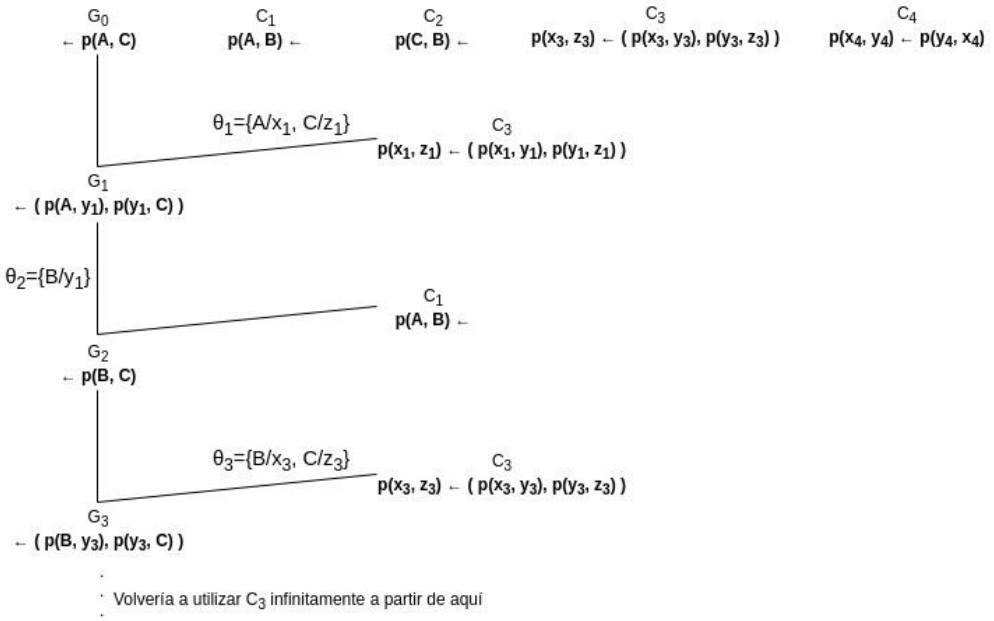
```
InterpreteAbstracto(P, G)
   resolvente ← G;
   mientras (resolvente ≠ Ø) hacer
        Q=f_s(resolvente)
        Si (existe cláusula C de P cuya cabeza unifique con Q)
        Entonces
                resolvente ← resolvente_SLD de resolvente y C
                                con umg θ
                G ← G θ
        Sino SalirMientras
        finsi
   finMientras
   Si (resolvente = Ø) Entonces (Devolver G)
   Sino (Devolver fallo)
```

En concreto, para un intérprete hay que elegir una regla de cómputo y una regla de búsqueda:

- **Regla de cómputo**: criterio para seleccionar el literal g_S sobre el que se resuelve. El uso de una u otra regla de cómputo no afecta más que al orden en que se reciben las respuestas. Un ejemplo puede ser elegir siempre el primer literal a la izquierda.
- **Regla de búsqueda**: criterio para seleccionar la cláusula C_i de programa con la que se resuelve. Dependiendo de la regla usada, puede afectar al determinismo del intérprete, es decir, que termine o no. Los dos únicos ejemplos que nos dicen los apuntes son primera cláusula no utilizada y primero en profundidad:
    - **Primera cláusula no utilizada**: regla que utiliza la primera cláusula que encaje y que lleve más tiempo sin ser usada. Si encuentra una cláusula que encaja, pero hay otras que llevan más tiempo sin ser usadas, la salta.



- **Primero en profundidad**: aplica la primera cláusula que encuentra y que puede coincidir sin contemplar otras opciones.



2.4. Concepto de respuesta

Dado un programa P y una meta G, un intérprete dará una de las siguientes respuestas a la resolución de P con G:

- Una sustitución θ para las variables de G y/o P.
- "no" (literalmente) si no se puede llegar a una cláusula vacía.

Se considera respuesta correcta si la sustitución aplicada permite simplificar la meta con alguna de las cláusulas del programa. En caso de que no existan sustituciones, la respuesta correcta será no. Por ejemplo:

P={p(x) ←} *G= ← p(x)* resp: *θ=∅*

No hace falta sustituir, ya son equivalentes.

P={p(x) ←} *G= ← p(y)* resp: *θ=∅*

No hace falta sustituir (*θ={x/y}=∅*, esto se denomina variante alfabética).

P={p(x) ←} *G= ← p(y)* resp: *θ={A/y}*

Ambos se pueden sustituir por una constante. Además, aquí se ve que no tiene por qué haber una sola respuesta correcta.

P={p(x) ←} *G= ← p(f(y))* resp: *θ=∅*

Las variables se pueden sustituir por funciones de variables (pero no al revés)

P={p(x, x) ←} *G= ← p(x, f(x))* resp: no

No se puede sustituir una función de una variable por una variable.

Diremos, además, que si una meta es consecuencia lógica del programa, existe al menos una respuesta correcta para dicha meta distinta de “no”. Y todas las respuestas correctas, o alguna generalización de las mismas, se pueden computar.

2.4.1. Respuesta computada

Unificador más general de aquellos utilizados exclusivamente sobre las metas G_i. Para obtener este, primero se recogen todas las sustituciones usadas sobre todas las G y se van unificando (de último a primero). Por ejemplo (sacado de los apuntes, ahí obtiene las sustituciones aplicando el proceso SLD):

G = ← abuelo(x, y)

θ_1={x/u, y/v} |

θ_2={luis/x, carlos/w} | -> $\theta_2\theta_3$ = {luis/x, carlos/w}{jorge/y} = {luis/x, carlos/w, jorge/y}

θ_3={jorge/y} | $\theta_1\theta_2\theta_3$ = {x/u, y/v}{luis/x, carlos/w, jorge/y} =

= {luis/u, jorge/v, luis/x, carlos/w, jorge/y}

Y, finalmente, de esa unificación nos quedamos solo con las sustituciones que sustituyen variables en la meta original:

G = ← abuelo(**x**, **y**) $\theta_1\theta_2\theta_3$ = {luis/u, jorge/v, ***luis/x***, carlos/w, ***jorge/y***}

Respuesta computada (formalmente, respuesta computada de de P ∪ {G}):

θ = {luis/x, jorge/y}

La respuesta computada es más general y engloba más valores que las respuestas correctas (y, a su vez, hay muchas más respuestas correctas que computadas). Por eso, de la respuesta computada se puede obtener cualquier respuesta correcta.

2.4.2. Propiedades

- **Teorema de solidez de resolución SLD**: Sea P un programa definido y G una meta definida, toda respuesta computada de P U {G} es una respuesta correcta de P U {G}.
- **Teorema de complitud de resolución SLD**: Sean P un programa definido, G una meta definida y θ una respuesta correcta de P U {G}, existe una respuesta computada σ y substitución γ tales que θ y σγ tienen el mismo efecto sobre las variables de G (la respuesta computada puede ser más general que la correcta).
- **Teorema de independencia de la regla de cómputo**: Sean P programa definido y G meta definida tal que P U {G} tiene una refutación SLD con respuesta computada θ; para cualquier otra regla de cómputo R, existe una

refutación SLD de P U {G} vía R con respuesta computada θ' tal que Gθ' es una variante alfabética de Gθ.

2.5. Negación

2.5.1. Suposición de mundo cerrado

Método que permite suponer que, dado un programa definido P y un átomo a, si no se cumple $P \Rightarrow a$, se puede deducir $\neg a$. En la programación lógica no se puede garantizar el cómputo de $P \Rightarrow a$.

2.5.2. Negación por fallo

Suponiendo un programa NORMAL P y una meta NORMAL G, si a la hora de hacer el cómputo vemos que no hay forma de llegar a la cláusula vacía, podemos aplicar la negación por fallo.

La negación por fallo consiste en negar la meta (a la meta negada la llamaremos G^{NF}) y utilizarla con el programa. Si aplicando esta G^{NF} al programa llegamos a la cláusula vacía, el resultado del programa P con G será "no". En caso contrario (si P con G^{NF} o, formalmente, P $\cup$ $\{G^{NF}\}$, no permite llegar a la cláusula vacía por encontrarnos con ramas infinitas o ramas fallo), el resultado será $\theta = \varnothing$ (este símbolo, sustitución vacía, significa que cualquier sustitución es válida).

2.5.3. Regla de inferencia no monotónica

Se dice que la monotonía es la propiedad que implica que el aumento de conocimientos no reduce el conjunto de las cosas conocidas. Formalmente, si tenemos una fórmula A y dos conjuntos de fórmulas Γ y Δ, la monotonía significaría que si

WIP