Universidad de Los Andes

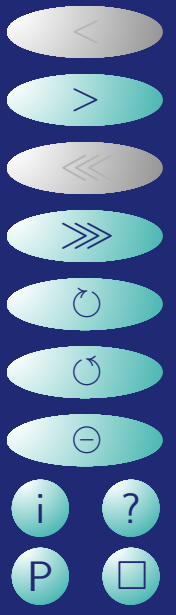
http://cesimo.ing.ula.ve

Unidad 4: La semántica de la negación

Jacinto Dávila

mailto:jacinto@ula.ve

Centro de Simulación y Modelos (CESIMO)



Negación por falla

En esta unidad tratamos el tema de la negación en lógica, discutiendo someramente los siguientes aspectos técnicos:

1. La negación por falla.
2. La suposición del mundo cerrado.
3. Completitud (*completion*) de programas lógicos.
4. Caracterización de la falla finita.

Pero, ¿Cuál es el problema con la negación?.

Sea T :

$feliz(X) \leftarrow rico(X) \wedge famoso(X).$

$rico(pedro).$

$rico(maria).$

$famoso(francisco).$

Considere el árbol SLD:

$\leftarrow feliz(X)$

$\downarrow$

$\leftarrow rico(X) \wedge famoso(X)$

$\downarrow X = pedro$ $\qquad\qquad$ $\searrow X = maria$

$\leftarrow famoso(pedro)$ $\qquad\qquad$ $\leftarrow famoso(maria)$

$\downarrow$ $\qquad\qquad$ $\downarrow$

$\bullet$ $\qquad\qquad$ $\bullet$

Nota: recuerden que el punto $\bullet$ significa que no tuvimos éxito en la prueba. Es decir, fallamos en llegar al $\square$.

La negación y los mundos cerrados

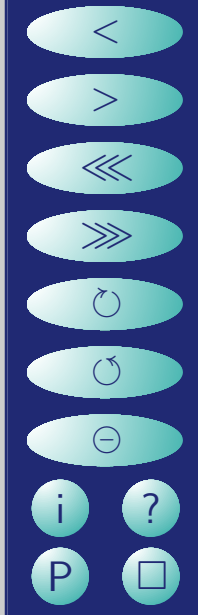
De la lámina anterior sabemos que $T \not\models \exists X(feliz(X))$

¿Qué podemos decir acerca de : $T \models \neg\exists X feliz(X)$?

Si adoptamos LA SUPOSICIÓN DEL MUNDO CERRADO (CWA) acerca de T, podemos decir:

$$CWA(T) \models \neg\exists X(feliz(X))$$

La suposición de un mundo cerrado establece que si Φ no es la consecuencia lógica de una teoría, podemos concluir $\neg\Phi$ a partir de esa teoría una vez **cerrada**.



Definiendo negación por falla

Para aplicar la negación por falla.

infiera $\neg q$ *si falla tratando de probar que* $P \models q$

$$P \vdash \neg p \text{ si } \neg P \vdash p$$

siempre que que: q sea atómica y P sea consistente.

Definición lm4.1 : Una refutación SLDNF para O_o a partir de T usando la función de selección F es una secuencia de objetivos normales O_o a O_n y una secuencia de substituciones θ_1 a θ_n donde O_n es la cláusula vacía y cada O_{i+1} se obtiene de O_i haciendo lo siguiente:

- Si $F(O_i)$ es positivo entonces O_{i+1} es un resolvente de O_i con una variante de una cláusula en T usando la substitución θ_{i+1}.
- Si $F(O_i)$ es negativo entonces existe un árbol SLDNF con falla finita para O_i a partir de T usando la función de selección F, y θ_{i+1} es vacío.

¿Para qué NF?

Con la inclusión de la negación en el cuerpo de las reglas, podemos escribir programas como:

$infeliz\ (X) \leftarrow mortal(X) \wedge no\ iluminado(X).$

$mortal\ (X) \leftarrow humano(X).$

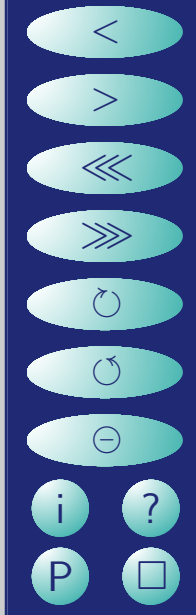
$humano(buda).$

$humano(socrates).$

$iluminado(buda).$

Pero.... ¿cúal es el significado de este programa si se supone que implica $infeliz(socrates)$?.

¿Cual es el significado del siguiente programa?



El caso de Bruno, el pingüino

Considere esta formalización que versa sobre pingüinos:

Las aves vuelan, siempre que no sean aves anormales:

$$\forall X\ (vuela(X)\ \leftarrow\ ave(X)\ \wedge \neg ave\_anormal(X).)$$

Un ave es anormal si sólo camina:

$$\forall X\ (ave\_anormal(X)\ \leftarrow\ solo\_camina(X)).$$

Un pingüino solamente camina, siempre que no sea un pingüino anormal:

$$\forall X\ (solo\_camina(X)\ \leftarrow\ pinguino(X)\ \wedge\ \neg pinguino\_anormal(X)).$$

Un pingüino es un ave:

$$\forall X\ (ave(X)\ \leftarrow\ pinguino(X)).$$

Bruno es un pingüino:

$$pinguino(bruno).$$

Actividad: Encuentre un modelo mínimo para ese programa.

Los peligros de la negación por falla

Considere el siguiente programa (que parece una base de datos):

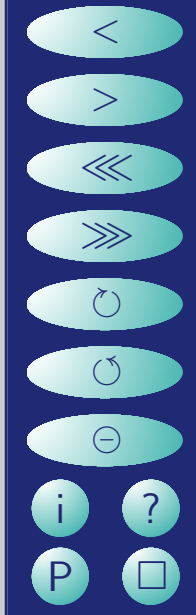
persona(adan).

persona(cain).

persona(abel).

hijo(cain, adan).

hijo(abel, adan).



Observe el siguiente árbol de refutación sobre el programa anterior:

$\leftarrow$ *persona*(X) *no hijo*$(X, adan)$

$\downarrow$ X = adan

$\leftarrow$ *no hijo*$(adan, adan)$

$\downarrow$

$\downarrow$ $\leftarrow$ *hijo*$(adan, adan)$

$\downarrow$ $\downarrow$

$\downarrow$ $\bullet$

$\downarrow$

$\square$

Observen como, al final, se abre una "caja" para probar el átomo "adentro" de la negación. Esto es diferente a lo que ocurría con SLD puro y puede ocasionar problemas.

Ahora considere este árbol sobre el mismo programa anterior:

$\leftarrow$ *no hijo(X, adan)*

$\downarrow$

$\downarrow$ $\leftarrow$ *hijo(X,adan)*

$\downarrow$ $\downarrow X = cain$ $\searrow X = abel$

$\downarrow$ $\square$ $\square$

$\bullet$

Esta respuesta no es correcta! (¿Verdad?). ¿Qué está pasando?.

Los peligros de la negación por falla: la raíz del problema

Para evitar esos "problemas y peligros" haga lo siguiente:

Levante su mano derecha.

¿Jura Ud.

1. Nunca seleccionar para resolución un literal negativo con variables (e.g. no $p(X)$)
2. Siempre evaluar los literales negativos completamente antes de evaluar los otros literales.

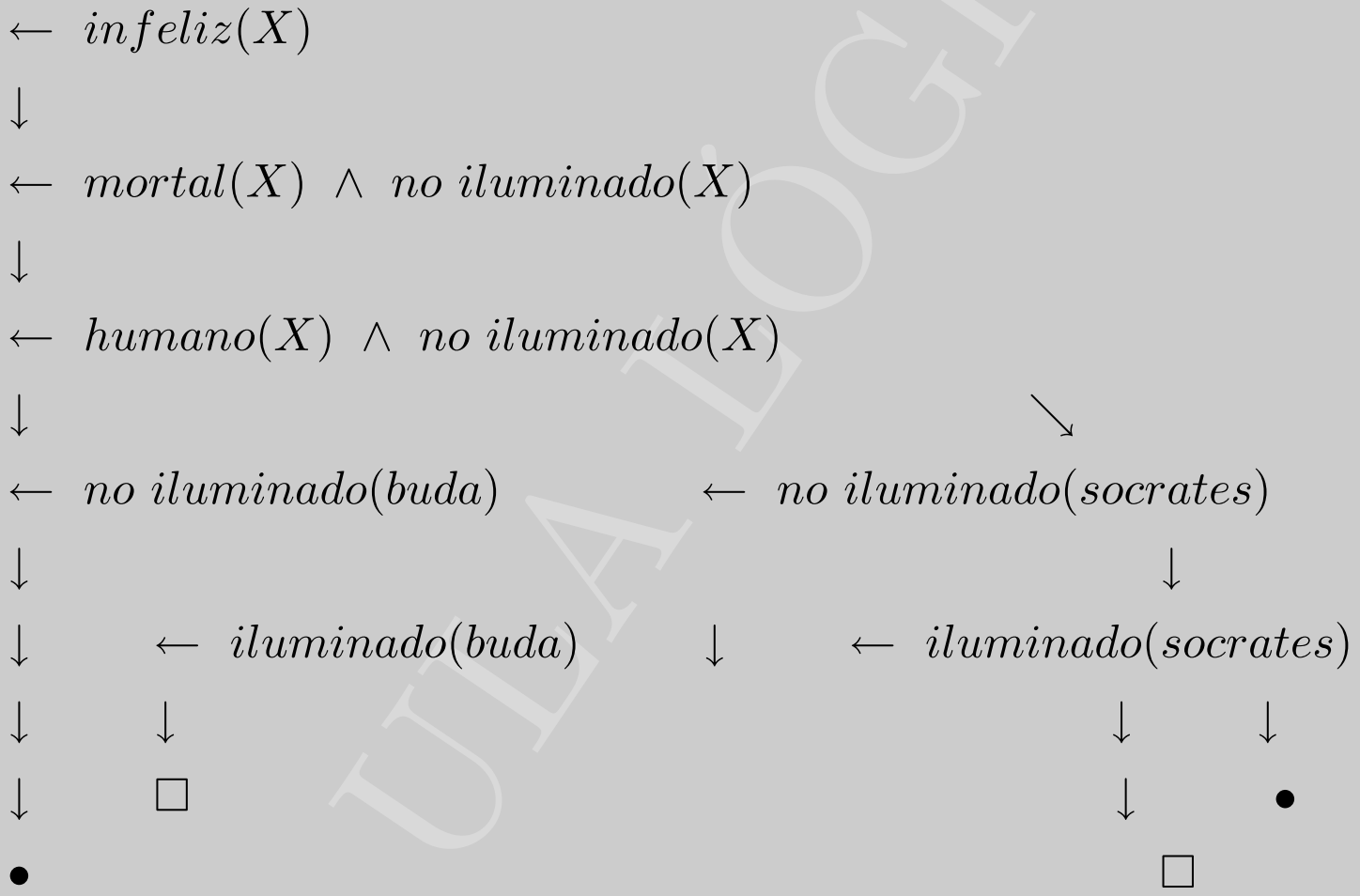
?.

Si Ud. responde "lo juro" todo está resuelto...

bueno, casi todo. Hay otras complicaciones. Observen con cuidado las siguientes láminas.

El espacio de búsqueda SLDNF

Por favor, revisen este árbol:



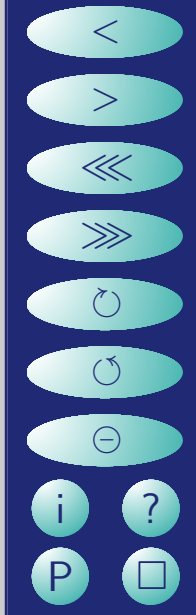
Definiciones para SLDNF

Definición lm4.2: En el contexto de SLDNF, una función de selección es *segura*, si nunca selecciona un literal negativo que contiene variables. Recuerden el juramento. Quienes juraron son ahora funciones de selección seguras.

Definición lm4.3: Una refutación SLDNF falla (con una respuesta probablemente incorrecta) con "*floundering*" si la función de selección solo puede escoger entre literales negativos con variables en ellos.

Así, para seguir siendo correcto, el procedimiento SLDNF tiene que dejar de ser completo

Actividad: Asegúrese de entender esta última afirmación/



Suposición del mundo cerrado y programas completos

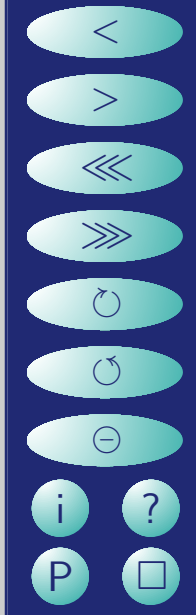
La suposición del mundo cerrado (Closed-world assumption: CWA) se puede formalizar así:

$$CWA\ (P) = P \cup \{\ \neg q \mid q \in B(P)\ y\ no\ P \models q\ \}$$

pero esta formalización, a pesar de lo razonable que parece, tiene problemas. Observe lo que ocurre con $CWA(P)$ si P es:

$$p \vee q$$

Una alternativa a esa formalización es el siguiente proceso más elaborado para *completar* (cerrar) un programa P:



Programas completos

Para construir una BASE DE DATOS COMPLETA de un programa P, $COMP(P)$:

1. Arregle P como un conjunto de cláusulas de forma : $q \leftarrow$ *cuerpo;*
2. Por cada q que se menciona en P pero no se define en P,construya $\neg q$;
3. Por cada q que tenga una definición en P con la forma:
   $q \leftarrow$ cuerpo-1
   $q \leftarrow$ cuerpo-2
   $q \leftarrow$ cuerpo-3
   ...
   $q \leftarrow$ cuerpo-n
4. construya la oración $q \leftrightarrow$ (cuerpo-1 $\vee$....$\vee$ cuerpo-n). (Observe que el cuerpo de una afirmación es el átomo *cierto*)

Ejemplos de programas completos

Una oración obtenida por los pasos anteriores se denomina la definición completa de la relación q.

A partir de: $q \leftarrow p$ se obtiene $\longrightarrow \left\{ \begin{array}{c} q \leftrightarrow p \\ \neg p \end{array} \right\}$

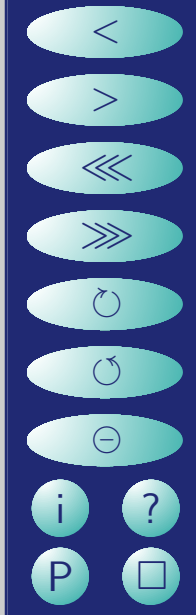
A partir de: $q \leftarrow \neg p$ se obtiene $\longrightarrow \left\{ \begin{array}{c} q \leftrightarrow \neg p \\ \neg p \end{array} \right\}$

A partir de: $p \leftarrow \neg q$ se obtiene $\longrightarrow \left\{ \begin{array}{c} p \leftrightarrow \neg q \\ \neg q \end{array} \right\}$

Programas completos en lógica de predicados

Para construir la base de datos completa de un programa P que contiene predicados.

1. GENERALICE cada cláusula en P definiendo cada predicado q.
2. En cada cláusula generalizada cuantifique existencialmente cada variable *local* a la cláusula.
3. Proceda como en el caso proposional descrito antes.
4. Incluya una definición apropiada del predicado " $=$ ".



Por ejemplo:

$$padre\ (X) \leftarrow hijo\ (Y,X)$$

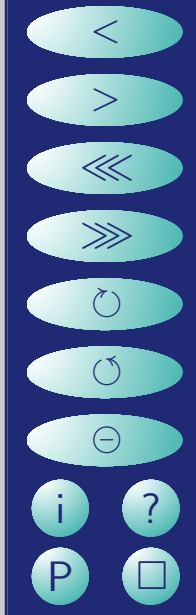
1.-.....

2.- $padre\ (X) \leftarrow \exists Y\ (hijo\ (Y,X))$

3.- $padre\ (X) \leftrightarrow \exists Y\ (hijo(Y,X))$

4.-.....

Actividad: Invente un ejemplo



Regla de computación justa y conjuntos de falla para programas definidos

Considere la ejecución de objetivos con la forma ?q usando un programa **definido** P y una regla de computación R:

La base de Herbrand de un programa P se puede particionar en tres conjuntos de átomos básicos como q. Aquellos para los cuales ?q termina con éxito, aquellos para los que ?q falla en tiempo finito y aquellos para los que ?q falla infinitamente (es decir, no termina). Uno puede indicar esos conjuntos respectivamente como SS(P,R), FF(P,R) y IF(P,R), enfatizando que su conformación depende del programa particular y de la regla de computación que se emplee.

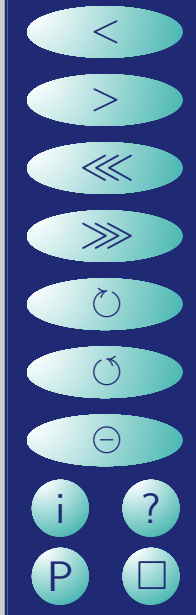
Hemos visto que para los árboles SLD, si una pregunta termina en éxito en algún árbol SLD entonces termina con éxito en cualquier árbol SLD. A eso lo llamamos "*independencia de la regla de computación*".

Desafortunadamente, eso no se cumple con las preguntas que terminan en falla (de cualquier tiempo finita ó infinita).

Regla de computación segura

Para obtener un contexto en que si se cumpla, se introduce la noción de regla de computación segura:

Definición lm4.4: Una **regla de computación justa** es una que garantiza que cualquier llamada (pregunta) introducida en el proceso de computación es seleccionada luego de un número arbitrario pero finito de pasos de ejecución.



Conjuntos de falla para programas definidos

Con lo anterior uno puede definir conjuntos de falla independientes de la regla de computación: FF(P) es el conjunto de todos los átomos q, para los cuales ?q tiene algún árbol SLD de falla justa y finita.

Lo interesante es poder decir luego cosas como:

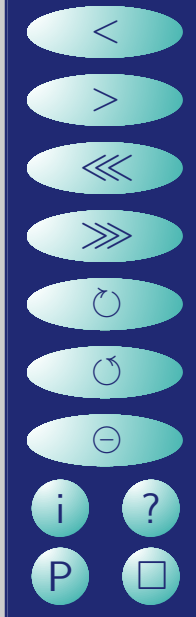
Para todo $q \in B(P)$

$q \in FF(P)$ si y solo si $\leftarrow$ no q tiene éxito con SLDNF*

$q \in SS(P)$ si y solo si $\leftarrow$ q tiene éxito con SLDNF*

SLDNF* = SLDNF con una regla de computación segura.

NOTA: Una regla de computación segura no tiene que ser justa y viceversa. Los conceptos no están conectados.



Caracterización matemática de la falla finita

Para cualquier programa definido P, hay una forma simple de construir el conjunto de éxito SS(P) usando la función T_P:

$$T_P(I) = \{q | (q \leftarrow body) \in G(P) \text{ y } body \text{ es } \textbf{cierto} \text{ en } I\}$$

Comenzando con el elemento al fondo del latis de interpretaciones, $\emptyset$, la aplicación repetida de esa función genera la cadena monotónicamente creciente:

$$\emptyset \subseteq T_P(\emptyset) \subseteq T_P^2(\emptyset)...$$

cuyo límite $T_P \uparrow \omega = mcs\{T_P^k(\emptyset) | k \in N\} = \bigcup_{k \in N} T_P^k(\emptyset)$

es precísamente SS(P) y LFP(TP).

Pues bien, algo similar podemos hacer para definir FF(P):

$$FF(P) = B(P) - \bigcap_{k \in N} T_P^k(B(P))$$

La aplicación sucesiva de la función T_P, comenzando con el elemento tope de la látis de interpretaciones, $B(P)$, genera una secuencia monotónicamente decreciente:

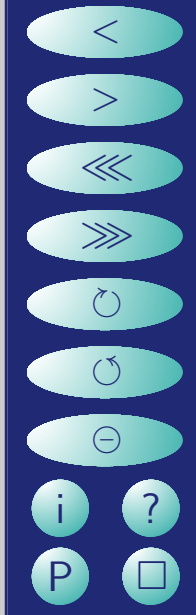
$$B(P) \supseteq TP(B(P)) \supseteq TP2(B(P))...$$

cuyo límite es (T_P abajo omega): $T_P \downarrow \omega = mci\{T_P^k(B(P))|k \in N\} = \bigcap_{k \in N} T_P^k(B(P))$

Con ello podemos escribir:

$$FF(P) = B(P) - T_P \downarrow \omega$$

y listo.. tenemos una definición matemática del conjunto sobre el cual nuestras consultas van a fallar.



Un ejemplo de la caracterización de falla finita

Considere $H = \{jacinto, gandhi\}$ y P:

$$noble(X) \leftarrow ama(X)$$

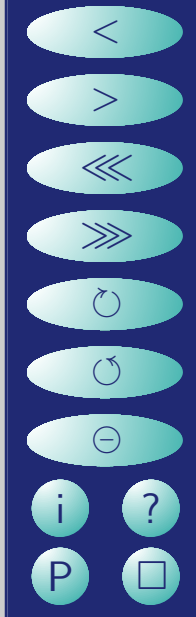
$$ama(gandhi)$$

Su $G(P)$ se puede escribir en forma compacta como:

$$ng \leftarrow ag$$

$$nj \leftarrow aj$$

$$ag$$



Sobre ese programa, este es el procedimiento iterativo para conseguir el conjunto de falla finita:

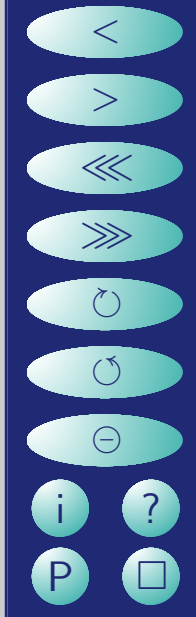
Tenemos $B(P) = \{ng, nj, ag, aj\}$. Así que:

$$FF(P,1) = B(P) - T_P(B(P)) = \{ng, nj, ag, aj\} - \{ng, nj, ag\} = \{aj\}$$

$$FF(P,2) = B(P) - T_P^2(B(P)) = \{ng, nj, ag, aj\} - \{ng, ag\} = \{nj, aj\}$$

$$FF(P,3) = B(P) - T_P^3(B(P)) = \{ng, nj, ag, aj\} - \{ng, ag\} = \{nj, aj\}$$

y como $T_P^k(B(P)) = \{ng, ag\}$ para todo $k \geq 2$ entonces $FF(P,2) = FF(P) = \{nj, aj\}$



Programas razonables

Uno puede requerir que los programas lógicos nunca fallen en forma infinita. Es decir, que cualquier pregunta siempre termine (cuando se le presenta a un interpretador SLDNF) con exito (yes) o fracaso (no). En ese caso uno tendría:

$IF(P) = \emptyset$

y en consecuencia: $T_P \uparrow \omega = T_P \downarrow \omega$

y en consecuencia: $B(P) = SS(P) \cup FF(P)$

Desafortunadamente, todo esto sólo aplica a los programas lógicos definidos.

La semántica de completitud para el "no" de falla finita

1. Interprete **no** (**fail, not**, etc) como la negación clásica ($\neg$) .
2. Vincule las preguntas a cada programa P con las consecuencias lógicas de $Comp(P)$, no de P. Así:

   Para todo $q \in B(P)$:

   $$Comp(P) \models q \text{ si } ?q \text{ tiene éxito con SLDNF*}$$

   $$Comp(P) \models \neg q \text{ si } ?\mathbf{no}\ q \text{ tiene éxito con SLDNF*}$$

* con una regla de computación segura.

Para establecer una semántica, normalmente se presentan los resultados de sanidad (correctitud) como los de la lámina anterior y los de completitud, como los que siguen:

Para todo $q \in B(P')$:

?q termina con éxito en SLDNF si $Comp(P\prime) \models q$

? **no** q termina con éxito en SLDNF si $Comp(P\prime) \models \neg q$

Desafortunadamente para que estos se cumplan en el caso de SLDNF tenemos que restringir la estructura de los programas ($P\prime$).

Las restricciones estructurales son un área de investigación muy dinámica en programación lógica. El propósito de buena parte de esa investigación es darle a SLDNF (y a otros procedimientos sobre programas no definidos) la firmeza matemática que se obtiene para SLD (y los programas definidos).

Restricciones estructurales en programas lógicos

¿Qué son consistencia por llamadas (*call-consistency*), programas estrictos (*strictness*) y programas permitidos (*allowedness*)?.

Actividad: Muestre un ejemplo de cada uno y consiga una ejemplo ó aplicación que no pueda escribirse como un programa permitido.

Considere la pregunta posiblemente compuesta $?literales$**. Si cierto programa** P **es permitido, consistente-por-llamadas y estricto con respecto a** $?literales$ **entonces:**

$?literales$ **terminará con éxito y con respuesta** θ **si y solo si :**

$$Comp(P) \models \forall(literales\theta)$$

$?literales$ **fallará en forma finita si y solo si** $Comp(P) \models \neg\exists(literales)$

Fin de Unidad 4

La negación es uno de los elementos lingüisticos más controversiales en representación del conocimiento. Lo que acabamos de hacer es apenas una introducción a este problema en el contexto de la programación lógica. Confío que puedan notar algunos de los trastornos que produce la negación que, por cierto, no son muy discutidos en otros paradigmas de computación (aunque el problema también existe en ellos).

Un problema estrechamente ligado con la negación es el de razonamiento no monótono: La posibilidad de razonar y cambiar de opinión. Sugiero que revisen la WEB y traten de averiguar quienes investigan ese problema y con que propósitos. Será muy instructivo para quienes están interesados en las llamadas "aplicaciones inteligentes" de la computación.