Capítulo 3

Especificaciones

XIV Conferencia Latinoamericana de Informática
17avas. Jornadas Argentinas de Informática
e Investigación Operativa.
Buenos Aires, Setiembre 1988.

Especificación de Operadores de Control y Desarrollo de Herramientas de Puesta a Punto para un Ambiente de Programación Lógica.*

Nora Szasz

ESLAI - Argentina
Escuela Superior LatinoAmericana de Informática
Casilla de Correo 3199
(1000) Buenos Aires-Argentina

Resumen. Se extiende el conjunto de operadores de control usuales de los lenguajes de Programación Lógica, especificando formalmente los operadores de disyunción, "snip" y corrutinaje, además de la conjunción y negación. La especificación se da en términos de árboles del estilo de los SLD. Se extiende la definición asociando acciones a los nodos de los árboles, lo cual permite especificar comandos de herramientas de puesta a punto de programas. Las herramientas se implementan sobre un meta-intérprete del lenguaje extendido.

*Trabajo realizado en la ESLAI en el marco del proyecto API-2 para la aprobación de la Pasantía del Quinto Semestre de la Licenciatura en Informática bajo la dirección del Dr. J. Vidart.

Capítulo 0.

Introducción.

En el ámbito de la Programación en Lógica se reconoce la carencia de ambientes integrados para el desarrollo de programas. En [CDKMV 88] se propone la construcción de un meta-ambiente para la Programación en Lógica (API-2) en el que un usuario pueda definir lenguajes junto con herramientas asociadas para distintos tipos de aplicaciones.

Un aspecto en que resulta interesante dotar de flexibilidad a los lenguajes de Programación en Lógica es el de los mecanismos de control. Una opción para implementar este objetivo es permitir la definición de lenguajes en que se combinen operadores de control seleccionados de un conjunto predefinido que incluya operadores alternativos además de los convencionales.

En particular, como una versión preliminar de API-2 se ha desarrollado un intérprete para una extensión de Prolog en la que se introducen los siguientes operadores de control:

disyunción (;)
"snip" (^)
conjunción (,)
corrutinaje (\)
negación (NOT)

La primera parte de este trabajo se concentra en la especificación de la semántica de los programas que usan estos operadores representando sus ejecuciones por árboles del tipo SLD [Lloyd 84]. Esto se desarrolla en el capítulo 1, donde la semántica de los operadores es también explicada informalmente.

Por otra parte, cualquier ambiente de programación requiere de un conjunto de herramientas de puesta a punto. Las herramientas de puesta a punto son aquellas piezas de software que permiten observar el comportamiento de un programa durante su ejecución proveyendo facilidades de depuración.

La segunda parte de este trabajo tiene que ver con la especificación e implementación de un conjunto de herramientas de puesta a punto para la extensión de Prolog mencionada arriba. Las herramientas definidas son:

Traza.

Provee un seguimiento paso a paso de la ejecución de un programa. Se exhibe la secuencia de invocaciones de objetivos, asociando a cada uno de éstos un indicador de nivel de profundidad en el *árbol and*, representado mediante numeración (lexicográfica) de las llamadas. La traza puede suspenderse durante cierto rango de la ejecución.

Puntos de espionaje.

Provee puntos de detención del programa definidos por símbolos de predicado. Para cada instancia a ser probada durante la ejecución de un predicado construído con uno de estos símbolos, la ejecución es detenida y la instancia mostrada. Luego la ejecución continúa.

Ejecuciones parciales.

Provee la facilidad de efectuar ejecuciones parciales, donde una falla se transforma en una consulta al usuario quien determinará si el objetivo que falló debe considerarse válido (y en ese caso cuáles son las instancias correspondientes de las variables).

"Debugger".

Provee una forma de detección semi-automática de errores por inconsistencia e incompletitud de programas.

En el capítulo 2 se especifican comandos que implementan las primeras dos facilidades, usando extensiones apropiadas de las definiciones del capítulo 1. Las implementaciones correspondientes a todas las facilidades mencionadas se describen en el capítulo 3, mostrando para las tres primeras cómo fueron montadas sobre el intérprete de API-2 mencionado arriba.

En el capítulo 4 se incluyen conclusiones y posibles extensiones del trabajo. Finalmente, en los apéndices figuran algunas definiciones formales que se ha preferido no incluir en el texto principal para no hacer demasiado tediosa su lectura.

Capítulo 1

Especificación de un meta-intérperte de Programas Lógicos Extendidos con Operadores de Control

Se parte de las nociones clásicas de lenguaje de primer orden, átomo, cláusulas de Horn, lenguaje Prolog, tal como están definidas en [Lloyd 84].

En este capítulo, se presentara una variación de Prolog que introduce nuevos operadores de control. Se presentará la sintaxis y una semántica formal del lenguaje extendido.

definición: *secuencia extendida*

(a) TRUE y FAIL son secuencias extendidas.
(b) todo átomo (predicado Prolog) es una secuencia extendida.
(c) Si S_1 y S_2 son secuencias extendidas, entonces:

$(S_1 ; S_2)$,
$(S_1 \wedge S_2)$,
(S_1 , S_2),
$(S_1 \setminus S_2)$ y
NOT S_1

son secuencias extendidas.

Los simbolos {; ^ , \ NOT} serán llamados *operadores*, y el conjunto formado por ellos será llamado Θ.
■

definición: *goal extendido*

Un goal extendido es

$\leftarrow S$

donde S es una secuencia extendida.
■

definición: *cláusula extendida*

Una cláusula extendida es

$A \leftarrow S.$

donde A es un átomo, llamado la *cabeza* de la cláusula, y S una secuencia extendida, llamada *resolvente* de la cláusula.
■

definición: *programa extendido*

Un programa extendido es una lista de cláusulas extendidas.
■

notación:

Los átomos se denotarán con las primeras letras del alfabeto en mayúscula (A, B, A_1, A_2, etc.).

Los goals extendidos se denotarán en la forma $\leftarrow S, \leftarrow S_1$, o bien directamente G, G_1.

Se notará *secuencia*, *goal*, *cláusula* y *programa* por sus correspondientes *extendidos*.

semántica informal de los programas:

; es el operador *or*. El significado de $A \leftarrow (S_1;S_2)$. es: *para satisfacer* A, *debe satisfacerse* S_1 *o* S_2. Es equivalente a:

$A \leftarrow S_1.$
$A \leftarrow S_2.$

^ es un operador que representa una variedad "débil" del operador *cut* (!) de Prolog. Su efecto es el de un *snip* [Arity] sobre el primer operando. Su efecto es no reintentar el primer operando durante el "backtracking" en caso de falla del segundo.

El equivalente Prolog de A^B es

A^B← P.
P← A, !, B.

, es el operador *and* de Prolog.

\ es un operador de *corrutinaje*, que causa la *ejecución paralela* de los dos goals que él conecta. Su efecto es similar al recorrido *a lo ancho* del árbol *and* generado por el goal a demostrar.

NOT es el operador *not* de Prolog.

semántica formal de los programas:

La formalización de la semántica de los programas se hará representando todas las posibles evaluaciones de cada goal mediante un árbol.

Más precisamente, para cada programa se dará una función de goals en árboles, de forma tal que toda derivación de un goal dado en el programa estará representada por una rama del árbol que corresponde a dicho goal.

definición: *árbol de secuencias*

(a) Si S es una secuencia, entonces

$\langle S,[] \rangle$

es un árbol de secuencias (donde [] denota la lista vacía).

(b) Si S es una secuencia, $t_1,\ldots,t_n$ son árboles de secuencias y $\theta_1,\ldots,\theta_n$ son sustituciones para variables de S tales que, para todo i, las variables de θ_i no ocurren en t_i entonces

$\langle S,[\langle t_1,\theta_1\rangle,\ldots,\langle t_n\theta_n\rangle] \rangle$

es un árbol de secuencias.

■

En esta definición de árbol los subárboles que componen a cada árbol están ordenados [Knuth 68]. Además figura una sustitución asociada a cada subárbol, cuya intención es representar una etiqueta en la arista.

El orden de los subárboles se debe a que en Prolog las cláusulas de un programa están ordenadas, lo que debe respetarse en la elección de las mismas. Por medio de listas es muy simple hacer explícito este hecho.

En adelante se notará $\mathbb{S}$ al conjunto de las secuencias, $\mathbb{G}$ al de los goals, $\mathbb{P}$ al de los programas y $\mathbb{T}$ al de los árboles.

La sustitución identidad se notará id.

definición: *semántica de programas*

Para cada programa P se define la función $\tau(P)$, que a cada goal asocia un *árbol de secuencias* o sea:

$$\tau : \mathbb{P} \longrightarrow (\mathbb{G} \longrightarrow \mathbb{T})$$

Sea P un programa, entonces:

(a) $\tau(P)(\leftarrow \text{TRUE}) = \langle \text{TRUE}, [] \rangle$

(b) $\tau(P)(\leftarrow \text{FAIL}) = \langle \text{FAIL}, [] \rangle$

(c) $\tau(P)(\leftarrow A) = \langle A, \mathscr{L} \rangle$
donde
$$\mathscr{L} = [\langle \tau(P)(\leftarrow S_1\theta_1), \theta_1 \rangle, \ldots, \langle \tau(P)(\leftarrow S_n\theta_n), \theta_n \rangle]$$
si la lista $[B_1 \leftarrow S_1, \ldots, B_n \leftarrow S_n]$ es la sublista no vacía de P tal que A y B_i unifican con mgu θ_i y
$$\mathscr{L} = [\langle \langle \text{FAIL}, [] \rangle, id \rangle]$$
si no hay cláusulas que cumplan esto.

(d) $\tau(P)(\leftarrow (A;B)) = \langle (A;B), [\langle \tau(P)(\leftarrow A), id \rangle, \langle \tau(P)(\leftarrow B), id \rangle] \rangle$

(e) $\tau(P)(\leftarrow (A \wedge B)) = \langle (A \wedge B), [\langle \text{SNP}(\tau(P)(\leftarrow A), B, P), id \rangle] \rangle$

(f) $\tau(P)(\leftarrow (A,B)) = \langle (A,B), [\langle \text{CNJ}(\tau(P)(\leftarrow A), B, P), id \rangle] \rangle$

(g) $\tau(P)(\leftarrow (A \setminus B)) = \text{COR}(\tau(P)(\leftarrow A), \tau(P)(\leftarrow B))$

(h) $\tau(P)(\leftarrow \text{NOT } A) = \text{NEG}(\tau(P)(\leftarrow A))$

■

Una demostración de $\leftarrow S$ en P corresponde a una *rama exitosa*, esto es una rama que termine con el átomo TRUE. Las ramas que terminan con el átomo FAIL son *ramas de falla finita*, y aquellas que son de longitud infinita son las correspondientes a *derivaciones infinitas*. A continuación se define la función booleana scs, la cual, aplicada a un árbol, devuelve *true* si haciendo un recorrido del mismo en profundidad se encuentra una rama exitosa.

$$scs : \mathbb{T} \longrightarrow \{true, false\}$$

$$scs(S, []) = (S = \text{TRUE})$$

$$scs(S, [\langle t_1, \theta_1 \rangle | \mathscr{L}]) = scs(t_1) \text{ or } scs(\langle S, \mathscr{L} \rangle)$$

■

La interpretación de los casos (a) y (b) en la definición de la función τ, corresponde a la interpretación usual de los goals TRUE y FAIL: el goal $\leftarrow$ TRUE siempre es exitoso y el goal $\leftarrow$ FAIL siempre falla.

Para la evaluación de un átomo, se consideran todas las cláusulas del programa cuyas cabezas unifican con dicho átomo y para cada una de estas, el árbol correspondiente a su resolvente con la sustitución adecuada. En el caso en que no haya cláusulas que cumplan con esta propiedad, habrá una rama de fallo.

La sustitución asociada a un subárbol es la unificación de salida entre el goal y la cabeza de cla cláusula que se utiliza en el paso de inferencia.

Se han usado las funciones CNJ, SNP , COR y NEG para definir los árboles. Estas son funciones que "combinan" los árboles de los goals conectados por estos operadores de forma tal que el nuevo árbol refleja la semántica del goal resultante. Se definen, entonces, naturalmente, por inducción sobre los árboles.

Para evaluar la *conjunción* , de dos goals, es necesario evaluar inicialmente al primero y si esta evaluación fue exitosa, evaluar al segundo. En términos de Prolog, esto significa *por cada demostración del primer goal, intentar todas las posibles demostraciones del segundo con las unificaciones heredadas de la demostración del primero.*

La función CNJ hace exactamente eso: dado el árbol correspondiente al primer operando, *reemplaza* cada hoja TRUE por el árbol correspondiente al segundo operando, al que previamente se le han aplicado todas las substituciones que se hicieron en esa (rama->demostracion.

La definición de la función CNJ es la siguiente:

$$CNJ:\mathbb{T} \times \mathbb{S} \times \mathbb{P} \longrightarrow \mathbb{T}$$

$$CNJ(\langle TRUE,[]\rangle,S,P) = \tau(P)(\leftarrow S)$$

$$CNJ(\langle FAIL,[]\rangle,S,P) = \langle FAIL,[]\rangle$$

$$CNJ(\langle R,[\langle t_1,\theta_1\rangle,\ldots,\langle t_n,\theta_n\rangle]\rangle,S,P) = \langle R,[\langle CNJ(t_1,S\theta_1,P),\theta_1\rangle,\ldots,\langle CNJ(t_n,S\theta_n,P),\theta_n\rangle]\rangle$$

■

El operador ^ corresponde al operador ! en forma local entre sus dos operandos. En términos de Prolog, A^B significa *demuestre A y luego demuestre B, sin reintentar otra demostración alternativa de A*. Corresponde a la "fijación" de la primera demostración (con su correspondiente unificación asociada) e implica (en términos de árboles) la *poda* del primer árbol, una vez obtenido el primer éxito.

Para la definición del operador ^, deberá proveerse una función similar a CNJ, pero tal que, una vez obtenida la primera demostración de su primer operando, *pode* al árbol correspondiente a éste, *eliminando al resto de sus ramas*, y *reemplace* la hoja TRUE por el árbol correspondiente a su segundo operando, luego de la sustitución adecuada.

Formalmente

$$SNP:\mathbb{T} \times \mathbb{S} \times \mathbb{P} \longrightarrow \mathbb{T}$$

$$SNP(\langle R,[]\rangle,S,P) = \text{IF } (R = TRUE) \text{ THEN } \tau(P)(\leftarrow S) \text{ ELSE } \langle FAIL,[]\rangle$$

$$SNP(\langle R,[\langle t_1,\theta_1\rangle|\mathcal{L}]\rangle,S,P) = \text{IF } scs(t_1) \text{ THEN } \langle R,[\langle SNP(t_1,S\theta_1,P),\theta_1\rangle]\rangle \text{ ELSE } ADD(\langle t_1,\theta_1\rangle,SNP(\langle R,\mathcal{L}\rangle,S,P))$$

■

El operador de *corrutinaje* \ corresponde a la ejecución como corrutinas de los goals que él conecta. Esto es: para demostrar A\B se ejecutarán en forma simultánea pasos de derivación de A y B hasta que uno de ellos o bien se demuestre (en cuyo caso se continúa con el otro) o bien falle (en cuyo caso se reintenta con otra derivación).

La función COR *combina* los árboles de sus operandos de manera tal de proveer todas las posibles combinaciones de derivaciones simultáneas de ambos, de la siguiente forma:

$$\text{COR}: \mathbb{U} \times \mathbb{U} \rightarrow \mathbb{U}$$

$$\text{COR}(\langle \text{FAIL},[]\rangle,\langle R,\mathscr{L}\rangle) = \langle \text{FAIL} \backslash R,[\langle\langle \text{FAIL},[]\rangle,\text{id}\rangle]\rangle$$

$$\text{COR}(\langle R,\mathscr{L}\rangle,\langle \text{FAIL},[]\rangle) = \langle R \backslash \text{FAIL},[\langle\langle \text{FAIL},[]\rangle,\text{id}\rangle]\rangle$$

$$\text{COR}(\langle \text{TRUE},[]\rangle,\langle R,\mathscr{L}\rangle) = \text{IF } (R \neq \text{FAIL}) \text{ THEN } \langle \text{TRUE} \backslash R,[\langle R,\mathscr{L}\rangle]\rangle$$

$$\text{COR}(\langle R,\mathscr{L}\rangle,\langle \text{TRUE},[]\rangle) = \text{IF } (R \neq \text{FAIL}) \text{ THEN } \langle R \backslash \text{TRUE},[\langle R,\mathscr{L}\rangle]\rangle$$

$$\text{COR}(\langle A,\mathscr{L}_A\rangle,\langle B,\mathscr{L}_B\rangle) = \text{IF } \text{UNIF}(\mathscr{L}_A,\mathscr{L}_B) \neq [] \text{ THEN } \langle A \backslash B,\text{UNIF}(\mathscr{L}_A,\mathscr{L}_B)\rangle \text{ ELSE } \langle A \backslash B,[\langle\langle \text{FAIL},[]\rangle,\text{id}\rangle]\rangle$$

donde se define UNIF de la siguiente forma:

$$\text{UNIF}([],\mathscr{L}) = []$$

$$\text{UNIF}([\langle t,\theta\rangle|\mathscr{L}_1],\mathscr{L}_2) = \text{APPEND}(\text{UNIF}_1(\langle t,\theta\rangle,\mathscr{L}_2),\text{UNIF}(\mathscr{L}_1,\mathscr{L}_2))$$

$$\text{UNIF}_1(\langle t,\theta\rangle,[]) = []$$

$$\text{UNIF}_1(\langle t,\theta\rangle,[\langle u,\sigma\rangle|\mathscr{L}]) = \text{IF } \textit{compatibles?}(\theta,\sigma) \text{ THEN } [\langle t\delta \backslash u\delta,\theta\sigma\rangle,|\text{UNIF}_1(\langle t,\theta\rangle,\mathscr{L})] \text{ ELSE } \text{UNIF}_1(\langle t,\theta\rangle,\mathscr{L})$$

donde la función *compatibles?* es booleana e indica si existe una única sustitución δ que tenga el efecto de sus dos argumentos.
■

Para la evaluación de la *negación* de un goal, es preciso tratar de demostrar el goal no negado y, sólo en el caso de no poder hacerlo se habrá demostrado la tal negación. En términos de Prolog, el resultado de la evaluación de NOT S es: *si puede demostrar S, entonces* NOT S *falla, si no* NOT S *tiene éxito*.

La función NEG, entonces deberá buscar la primera hoja TRUE en el árbol de su operando y, de encontrarla, *sustituir* esta por FAIL y *podar* el resto del árbol (no es necesario buscar demostraciones alternativas del goal que se negó), si tal hoja TRUE no existe en el árbol, se colgará directamente de la raíz del árbol dado, una hoja TRUE.

$$\text{NEG}: \mathbb{U} \rightarrow \mathbb{U}$$

$$\text{NEG}(\langle S,\mathscr{L}\rangle) = \text{IF } \text{SCS}(S) \text{ THEN } \langle \text{NOT } S,[\langle \text{CNJ}(\langle S,\mathscr{L}\rangle,\text{FAIL},P),\text{id}\rangle]\rangle \text{ ELSE } \langle \text{NOT } S,[\langle \text{TRUE},\text{id}\rangle]\rangle$$
■

La función ADD construye un árbol a partir de dos dados, colocando al primero como primera rama del segundo.

$$\text{ADD}: (\mathbb{U} \times \mathbb{S}) \times \mathbb{U} \rightarrow \mathbb{U}$$

$$\text{ADD}(\langle t,\theta\rangle,\langle S,\mathscr{L}\rangle) = \langle S,[\langle t,\theta\rangle|\mathscr{L}]\rangle$$
■

De esta forma, queda completa la definición de la semántica de un programa en términos de árboles *or*, donde cada rama de un árbol correspondiente a un goal es una derivación de dicho goal en el tal programa.

Además, se puede asociar una sustitución a cada rama del árbol, que será la composición de las sustituciones asociadas a todos los arcos que componen esa rama. Así cada *respuesta* exitosa del sistema al goal será la aplicación de la sustitución asociada a cada rama exitosa al goal dado.

El recorrido del arbol se hará *en profundidad*, de la misma forma que se hace en Prolog, con lo que, un programa y un goal que sólo usen los operadores originales de Prolog obtendrán el mismo comportamiento que tendrían si se usara un intérprete Prolog tradicional. Es más, puede demostrarse que el árbol $\tau(P)(G)$ para un tal programa P y un tal goal G es el mismo que el árbol *SLD* para $PU\{G\}$ (Lloyd 84), a menos del etiquetado de los arcos con sustituciones.

No se incluye hasta el momento la semántica del operador ! (*cut*) ya que la variacion con que se trabajo no lo proveia.

De todas formas su definición puede darse con este formalismo, reformulando la regla (c) de la definición de la función τ.

(c) $\tau(P)(\leftarrow A) = \langle A,\mathscr{E}\rangle$
donde
Sea $\mathbb{C} = [B_1 \leftarrow S_1,\ldots,B_n \leftarrow S_n]$ la sublista de P tal que A y B_i unifican con mgu θ_i.
entonces,
Si $\mathbb{C} = []$ entonces $\mathscr{E} = [\langle\langle \text{FAIL},[]\rangle,id\rangle]$.
Si $\mathbb{C} \neq []$ y $\exists k\ (1 \leq k \leq n)$ tal que S_k es el primero que contiene una secuencia extendida "S_{k1},!,S_{k2}" con $scs(S_{k1}) = true$, entonces
$\mathscr{E} = [\langle\tau(P)(\leftarrow S_1\theta_1),\theta_1\rangle,\ldots,\langle snp(\tau(P)(\leftarrow S_{k1}\theta_k),S_{k2}\theta_k,P),\theta_k\rangle]$
En otro caso
$\mathscr{E} = [\langle\tau(P)(\leftarrow S_1\theta_1),\theta_1\rangle,\ldots,\langle\tau(P)(\leftarrow S_n\theta_n),\theta_n\rangle]$

■

capítulo 2.

Especificación de herramientas de puesta a punto para programas lógicos extendidos.

2.1 Introducción.

La especificación del capítulo anterior es suficiente para representar cualquier ejecución de un programa lógico extendido.

Las herramientas de puesta a punto esencialmente proveen información sobre el desarrollo de la ejecución de un programa. La información exhibida debe poder ser seleccionada por el usuario. Eventualmente, el usuario podrá también alterar el curso de la ejecución.

Para cumplir este propósito se debe asociar a la representación de la ejecución la información disponible para ser exhibida. Esto se hace en la sección 2.2 donde se extiende la definición del árbol de ejecución, dando una numeración de los goals que corresponde al orden en que serán invocados.

Por otro lado, deben definirse comandos cuyo efecto sea seleccionar la información a ser exhibida. Para ello, en la sección 2.3 se extiende nuevamente la definición del árbol de ejecución asociando acciones a los nodos. En 2.4 se definen comandos cuyo efecto es dado por funciones que modifican los árboles con acciones.

2.2 Arboles numerados.

definición: *secuencia numerada* y *p-número*

Una secuencia numerada es una secuencia donde cada componente de la misma está acompañado de una cadena de caracteres de la forma $n_1.n_2...n_m$, $(m \geq 1)$, donde los n_i son numeros naturales.

Tales cadenas seran llamadas *p-números*.

■

definición: *goal numerado*

Un goal numerado es aquel que se obtiene a partir de una secuencia numerada.

■

ejemplos: ← A(a,c)-1.9.2
←(A(y)-2.4;(TRUE^B(a,c))-9.1)
son goals numerados.

convenciones:

A partir de aquí, a los goals numerados, se les agregará el sufijo num, quedando de la forma ← S_{num},← S_{num_1}, o directamente G_{num}, G_{num_1}, etc.

definición: *árbol de secuencias numeradas*

(a) Si Snum es un secuencia numerada, entonces
$\langle Snum,[]\rangle$
es un árbol de secuencias numeradas.

(b) Si Snum es un secuencia $,t_1,\ldots,t_n$ son árboles de secuencias numeradas y $\theta_1,\ldots,\theta_n$ son sustituciones para variables de Snum que no ocurren en t_i entonces
$\langle Snum,[\langle t_1,\theta_1\rangle,\ldots,\langle t_n\theta_n\rangle]\rangle$
es un árbol de secuencias numeradas.

■

notación:

Se notará *n-secuencia*, *n-goal* y *n-àrbol* por sus correspondientes *numerados*, y los conjuntos formados por ellos se notarán $\mathbb{S}_n$, $\mathbb{G}_n$ y $\mathbb{T}_n$ respectivamente.

El conjunto de los p-números será notado $\mathbb{N}_p$.

definición: *numeración de goals*

Hay dos funciones, NUMINIT y NUM, las cuales sirven para numerar secuencias y se definen de la siguiente forma:

NUMINIT: $\mathbb{S} \longrightarrow \mathbb{S}_n$

NUMINIT(TRUE) = TRUE-1
NUMINIT(FAIL) = FAIL-1
NUMINIT(A) = A-1
NUMINIT((S1;S2)) = S1-1;S2-2
NUMINIT((S1^S2)) = S1-1^S2-2
NUMINIT((S1,S2)) = S1-1,S2-2
NUMINIT((S1;S2)) = S1-1;S2-2
NUMINIT((S1\S2)) = S1-1\S2-2
NUMINIT(NOT S) = (NOT S)-1

■

NUM: $\mathbb{S} \times \mathbb{N}_p \longrightarrow \mathbb{S}_n$

NUM(TRUE,K) = TRUE-K.1
NUM(FAIL,K) = FAIL-K.1
NUMINIT(A,K) = A-K.1
NUMINIT((S1;S2),K) = S1-K.1;S2-K.2
NUM((S1^S2),K) = S1-K.1^S2-K.2
NUM((S1,S2),K) = S1-K.1,S2-K.2
NUM((S1;S2),K) = S1-K.1;S2-K.2
NUM((S1\S2),K) = S1-K.1\S2-K.2
NUM(NOT S,K) = (NOT S)-K.1

■

La numeración de los goals surge de la necesidad de poder representar la relación existente entre los átomos que van apareciendo para ser demostrados en la ejecución de un programa extendido.

La función NUMINIT se aplicará al goal a ser demostrado y, como puede verse, lo que hace es: si el goal es un átomo, se numera con el número 1, y si es un goal compuesto otros dos, éstos se numeran con los números 1 y 2.

La función NUM recibe como parámetros a un p-número K y a una secuencia y devuelve la secuencia numerada obtenida de numerar a los componentes de la primera con los p-números K.1 y K.2, si ésta es compuesta o K.1 si nó .

La relación que quiere representarse mediante esta numeración de los goals es el orden de invocación que surge en la demostración de un goal determinado. Naturalmente, el goal a ser demostrado estará numerado con el número 1 (o 1 y 2 si es compuesto), y cada vez que un átomo numerado A-K se derive en otros B⟨op⟩C, éstos se numerarán de la forma B-K.1⟨op⟩C-K.2.

Por otro lado, se quiere representar en forma explícita en los árboles de secuencias la relación inversa de la anterior, esto es: el átomo del que es derivado cada goal que se demuestra. Para esto, cada vez que se deriva un goal G a partir de un átomo A, se agrega al nodo correspondiente a esta derivación el átomo A-o.K al final de G. Así, cada nodo que comience con un átomo de la forma A-o.K, corresponderá a la finalización de la prueba de A.

Para formalizar estos conceptos, se define la función τ_n, que es una extensión de la función τ definida en el capítulo anterior.

definición: *semántica "numerada" de programas*

Para cada programa P se define la función $n\tau(P)$, la cual aplicada a un goal devuelve un árbol de secuencias numerado de la siguiente forma:

$\tau_n : \mathbb{P} \longrightarrow (\mathbb{G}_n \longrightarrow \mathbb{T}_n)$

Sea P un programa, entonces:

(a) $\tau_n(P)(\leftarrow \text{TRUE-K}) = \langle \text{TRUE-K}, [] \rangle$

(b) $\tau_n(P)(\leftarrow \text{FAIL-K}) = \langle \text{FAIL-K}, [] \rangle$

(c) $\tau_n(P)(\leftarrow \text{A-K}) =$ IF K = 0.J
THEN ⟨A-K,[⟨⟨TRUE-J,[]⟩,id⟩]⟩
ELSE ⟨A,$\mathcal{E}$⟩

donde
$\mathcal{E} = [\langle cn_J n(\tau_n(P)(\leftarrow \text{NUM}(S_1\theta_1,K)),A\text{-}o.K,P),\theta_1\rangle, \ldots, \langle cn_J n(\tau_n(P)(\leftarrow \text{NUM}(S_n\theta_n,K)),A\text{-}o.K,P),\theta_n\rangle]$
si la lista $[B_1 \leftarrow S_1, \ldots, B_n \leftarrow S_n]$ es la sublista no vacía de P tal A y B_i unifican con mgu θ_i.
$\mathcal{E} = [\langle\langle \text{FAIL-K}, [] \rangle, id\rangle]$
en otro caso.

(d) $\tau_n(P)(\leftarrow (A;B)\text{-K}) =$
$\langle (A;B)\text{-K}, [\langle \tau_n(P)(\leftarrow A\text{-K}.1), id\rangle, \langle \tau_n(P)(\leftarrow B\text{-K}.2), id\rangle]\rangle$

(e) $\tau_n(P)(\leftarrow (A\hat{}B)\text{-K}) =$
$\langle (A\hat{}B)\text{-K}, [snp(\tau_n(P)(\leftarrow A\text{-K}.1), B\text{-K}.2, P), id\rangle]\rangle$

(f) $\tau_n(P)(\leftarrow (A,B)\text{-K}) =$
$\langle (A,B)\text{-K}, [\langle cn_J n(\tau_n(P)(\leftarrow A\text{-K}.1), B\text{-K}.2, P), id\rangle]\rangle$

(g) $\tau_n(P)(\leftarrow (A \backslash B)\text{-K}) = cor_n(\tau_n(P)(\leftarrow A\text{-K}.1), \tau_n(P)(\leftarrow B\text{-K}.2), K)$

(h) $\tau_n(P)(\leftarrow (\text{NOT } A)\text{-K}) = neg_n(\tau_n(P)(\leftarrow A\text{-K}))$

■

Se define la función scs_n del mismo modo que la función scs de la sección anterior, de la forma:

$scs_n : \mathbb{T}_n \longrightarrow \langle true, false \rangle$

$scs_n(S\text{-k},[]) = (S = \text{TRUE})$

$scs_n(S\text{-K},[\langle t_1,\theta_1\rangle!\mathcal{E}]) = scs_n(t_1)$ *or* $scs_n(\langle S\text{-K},\mathcal{E}\rangle)$

Para construir el árbol numerado correspondiente a un goal ← S en un programa P dado, debe computarse la función $\tau_n(P)(\leftarrow \text{NUMINIT}(S))$.
■

Se notará que la definición de la función τ_n es muy similar a la de la función τ definida en el capítulo anterior. De hecho, hay sólo dos diferencias fundamentales entre los árboles generados por τ y, por τ_n, aparte del hecho obvio de que una trabaja con secuencias sin numerar y la otra con secuencias numeradas. Tales diferencias están en el caso (c), y corresponden a las relaciones que se quisieron hacer explícitas en el árbol de derivaciones de un goal en un programa.

La regla (c) es la encargada de las derivaciones de programa propiamente dicho, pues es éste el único caso en que se *consulta* al programa para extraer de allí los resolventes de las cláusulas que unifican con un átomo dado. Es aquí entonces donde se introduce la relación antecedente-consecuente que se quería hacer explícita en el árbol generado por un goal: se agrega al final de cada rama correspondiente a las derivaciones del tal átomo, el mismo átomo *cerodo* para indicar que éste es consecuencia de lo recién demostrado.

El caso atómico debe también considerar ahora que el átomo a derivar corresponda a la finalización de la demostración de un goal. Para ello, según se definió, sólo basta con ver si el p-número que acompaña al átomo comienza con o.

Las funciones que se utilizan para definir la conjunción, snip, corrutinaje y negación son muy parecidas a las definidas anteriormente para los mismos propósitos. Sus definiciones se dan a continuación.

$$\text{CNJn}:\mathbb{T}n \times \mathbb{S}n \times \mathbb{P} \longrightarrow \mathbb{T}n$$

$$\text{CNJn}(\langle \text{TRUE-K},[]\rangle,\text{Snum},P) = \tau(P)(\leftarrow \text{Snum})$$

$$\text{CNJn}(\langle \text{FAIL-K},[]\rangle,\text{Snum},P) = \langle \text{FAIL-K},[]\rangle$$

$$\text{CNJn}(\langle R\text{-K},[\langle t_1,\theta_1\rangle,\ldots,\langle t_n,\theta_n\rangle]\rangle,\text{Snum},P) =$$
$$\langle R\text{-K},[\langle \text{CNJn}(t_1,\text{Snum}\theta_1,P),\theta_1\rangle,\ldots,\langle \text{CNJn}(t_n,\text{Snum}\theta_n,P),\theta_n\rangle]\rangle$$
■

$$\text{SNPn}:\mathbb{T}n \times \mathbb{T}n \longrightarrow \mathbb{T}n$$

$$\text{SNPn}(\langle R\text{-K},[]\rangle,\text{Snum},P) = \text{IF } (R = \text{TRUE})$$
$$\text{THEN } \tau(P)(\leftarrow \text{Snum})$$
$$\text{ELSE } \langle \text{FAIL-K},[]\rangle$$

$$\text{SNPn}(\langle R\text{-K},[\langle t_1,\theta_1\rangle!\mathscr{L}]\rangle,\text{Snum},P) =$$
$$\text{IF SCSn}(t_1)$$
$$\text{THEN } \langle R\text{-K},[\langle \text{SNPn}(t_1,\text{Snum},P),\theta_1\rangle]\rangle$$
$$\text{ELSE ADD}(\langle t_1,\theta_1\rangle,\text{SNPn}(\langle R\text{-K},\mathscr{L}\rangle,\text{Snum},P))$$
■

$$\text{COR}_n : \mathbb{T}_n \times \mathbb{T}_n \longrightarrow \mathbb{T}_n$$

$$\text{COR}_n(\langle \text{FAIL-K}, [] \rangle, \langle R\text{-J}, \mathscr{L} \rangle) = \langle \text{FAIL-K} \backslash R\text{-J}, [\langle \langle \text{FAIL-K}, [] \rangle, id \rangle] \rangle$$

$$\text{COR}_n(\langle R\text{-K}, \mathscr{L} \rangle, \langle \text{FAIL-J}, [] \rangle) = \langle R\text{-K} \backslash \text{FAIL-J}, [\langle \langle \text{FAIL-J}, [] \rangle, id \rangle] \rangle$$

$$\text{COR}_n(\langle \text{TRUE-K}, [] \rangle, \langle R\text{-J}, \mathscr{L} \rangle) = \text{IF } (R \neq \text{FAIL}) \text{ THEN } \langle \text{TRUE-K} \backslash R\text{-J}, [\langle R\text{-J}, \mathscr{L} \rangle] \rangle$$

$$\text{COR}_n(\langle R\text{-K}, \mathscr{L} \rangle, \langle \text{TRUE-J}, [] \rangle) = \text{IF } (R \neq \text{FAIL}) \text{ THEN } \langle R\text{-K} \backslash \text{TRUE-J}, [\langle R\text{-K}, \mathscr{L} \rangle] \rangle$$

$$\text{COR}_n(\langle A, \mathscr{L}_A \rangle, \langle B, \mathscr{L}_B \rangle, K) = \text{IF } \text{UNIF}(\mathscr{L}_A, \mathscr{L}_B) \neq \emptyset \text{ THEN } \langle (A \backslash B)\text{-K}, \text{UNIF}(\mathscr{L}_1, \mathscr{L}_2) \rangle \text{ ELSE } \langle (A \backslash B)\text{-K}, [\langle \langle \text{FAIL-K}, [] \rangle, id \rangle] \rangle$$

■

Donde la función UNIF es la misma que la que se definió en el capítulo anterior, junto con la definición de COR.

$$\text{NEG}_n : \mathbb{T}_n \longrightarrow \mathbb{T}_n$$

$$\text{NEG}_n(\langle R\text{-K}, \mathscr{L} \rangle) = \text{IF } \text{SCS}_n(R\text{-K}) \text{ THEN } \langle \text{NOT } R\text{-k}, [\langle \text{SNP}_n(\langle R\text{-K}.1, \mathscr{L} \rangle, \text{FAIL-K}, P), id \rangle] \rangle \text{ ELSE } \langle \text{NOT } R\text{-k}, [\langle \text{TRUE-K}, id \rangle] \rangle$$

■

Así se completa la definición de la función τ_n, que de una forma casi idéntica a τ, define la semántica de los programas haciendo explícito el orden de invocación de los objetivos a demostrar en cada derivación de un goal dado en un programa. Es más, esta numeración de los goals que van apareciendo en el árbol corresponde exactamente a la representación del *árbol and* de una ejecución mediante una numeracion lexicográfica de sus nodos.

2.3 Arboles con acciones.

Se introducen en esta sección nuevas modificaciones a los árboles definidos anteriormente y a la función τ_n. Tales modificaciones corresponden a la introducción de acciones asociadas a los nodos de los árboles, con el fin de poder definir algunas de las herramientas de puesta a punto sobre el árbol generado por la función τ_n.

Las acciones que se asociarán a los nodos serán las encargadas de mostrar algún tipo de información sobre el estado de la ejecución en ese momento, la cual dependerá de la herramienta que se esté usando.

Inicialmente, todos los nodos de un árbol tendrán asociada la acción vacía (NOP), y los distintos comandos invocados durante la ejecución, irán modificando las tales acciones y generando nuevos árboles. Para poder formalizar estos conceptos, se introducen las siguientes definiciones:

definición: *acciones*

Se define por extensión el conjunto de acciones
$\hat{A} = \langle \text{NOP}, \text{EXH}, \text{EXH-FAIL}, \text{EXH-SLV}, \text{EXH-NOSLV} \rangle$

definición: *árbol con acciones*

(a) Si Snum es un secuencia numerada y acc ∈ $\mathbb{A}$, entonces
$$\langle Snum,[],acc\rangle$$
es un árbol de acciones.

(b) Si Snum es un secuencia $,t_1,\ldots,t_n$ son árboles de acciones, $\theta_1,\ldots,\theta_n$ son sustituciones para variables de Snum tales que, para todo i, las variables de θ_i no ocurren en t_i y acc ∈ $\mathbb{A}$, entonces
$$\langle Snum,[\langle t_1,\theta_1\rangle,\ldots,\langle t_n\theta_n\rangle],acc\rangle$$
es un árbol de acciones.

El conjunto de los árboles de acciones se notará $\mathbb{T}_a$.

■

Se define la función $\tau_{na}:\mathbb{P} \longrightarrow (\mathbb{G} \longrightarrow \mathbb{T}_a)$ en forma similar a la función τ_n asociando a cada nodo del árbol la acción NOP. La definición completa de la misma se halla en el Apéndice I.

2.4 Comandos de puesta a punto.

A continuación se da una lista con los comandos que se quieren proveer. Algunos de estos comandos se corresponden con los mencionados en la introducción del trabajo. Tales comandos son:

(a) TRACE
(b) SKIP(nc)
(c) JUMP(nc,p-num)
(d) GOPRED(nc,pred)
(e) DEMON(nc,cond)
(f) SPY(pred).

Cada posible comando será una función de árboles con acciones en árboles con acciones. Cada una de estas funciones modificará al árbol, asociando diferentes acciones a los nodos, que dependerán del comando elegido. Una sesión estará compuesta por un goal a demostrar y una lista de comandos. Las funciones asociadas a estos comandos se compondrán sucesivamente, y el árbol resultante de la sesión será el obtenido de aplicar esta composición al árbol generado por el goal a demostrar. Esto es, para un goal G y una lista de comandos $[c_1,..,c_n]$ el árbol obtenido será $c_n(c_{n-1}(\ldots(c_1(\tau_{na}(G)))\ldots))$.

Los comandos no podrán ser invocados en cualquier orden: por ejemplo SKIP(n) (suspensión de la traza durante la demostración del goal asociado al nodo n) sólo podrá ser invocado después de la invocación de un TRACE. Para poder imponer esta noción de secuenciamiento cada función de comando tiene asociada una precondición, que impone requisitos sobre las acciones asociadas al árbol al cual la función se aplica.

Las especificaciones formales de las funciones asociadas a los comandos consisten básicamente en recorridos apropiados de los árboles de goals. A continuación, se describen los comandos y las funciones de manera informal, figurando las especificaciones formales en el Apéndice II.

TRACE: este comando provee un seguimiento paso a paso de la ejecución de un programa. Para poder representar tal facilidad en el árbol de la ejecución, debe asociarse a todos los nodos del mismo una acción que muestre el estado actual del programa. La información asociada no es la misma para todos los nodos: por ejemplo, hay nodos de falla, nodos de culminación de una prueba, nodos de reemplazo de un átomo por alguno de sus resolventes, etc. Todos estos casos son considerados en la especificación incluida en el Apéndice II, donde se explican en detalle.

SKIP(n): este comando provoca la suspensión de la traza durante la demostración del goal asociado al nodo identificado con el p-número n.

De acuerdo a la definición de τ_n, la demostración de un goal G-K comienza en el nodo que tiene tal goal asociado y termina con la primera aparición de un nodo cuyo goal sea G-0.K. Puede darse el hecho, sin embargo, de que no se pueda demostrar al tal goal, lo cual se pone en evidencia si no existe ningún nodo con goal asociado G-0.K en los subárboles del árbol cuya raíz es G-K.

Lo que debe hacerse entonces es un recorrido en profundidad del árbol al cual se aplica la función SKIP, buscar al primer nodo con p-número asociado n, el cual debe existir para que el comando tenga efecto, buscar en el árbol cuya raíz es ese nodo al primer nodo con p-número 0.n y asociar a todos los nodos que se recorren desde el nodo n al nodo 0.n la acción NOP. En el caso de no encontrar un nodo 0.n en ese árbol, se asociará a todos los nodos del árbol la acción NOP y se mostrará la falla de la demostración del nodo n.

JMP(n,m): provoca la suspensión de la traza entre el nodo n y el nodo con p-número m. Para que la invocación de este comando tenga efecto debe existir el nodo n en el árbol. Se asocia a los nodos entre n y m (en orden de profundidad primero) la acción NOP. Si no existe un nodo con p-número m, la operación descripta se aplica a todo nodo del árbol que siga a n.

GPRED(n,p): provoca la suspensión de la traza entre el nodo n y el comienzo de la demostración de alguna instancia del predicado p. Para que la invocación de este comando tenga efecto debe existir el nodo n en el árbol. Se asocia a los nodos entre n y el primero siguiente a éste con nombre de goal p en orden de profundidad primero la acción NOP. Si no existe un nodo con goal asociado p, la operación descripta se aplica a todo nodo del árbol que siga a n.

DEMON(n,cond): La activación de este comando provoca la suspensión de la traza entre el nodo n y el primer nodo cuyo goal asociado cumpla la condición cond. El nodo n debe existir en el árbol. Se asocia a los nodos entre n y el primero siguiente a éste cuyo goal cumpla cond (en orden de profundidad primero) la acción NOP. Si no existe un nodo cuyo goal cumpla cond, la operación descripta se aplica a todo nodo del árbol que siga a n.

SPY(p): se muestran todos los nodos del árbol que contengan al predicado p en su goal asociado o sean resolventes de una instancia del predicado p. Esto último equivale a que el nodo sea raíz de un subárbol de un árbol cuya raíz contiene a p.

Capitulo 3.

Montaje de herramientas de puesta a punto sobre un meta-intérprete de programas lógicos extendidos.

3.1 El meta-intérprete.

Un meta-intérprete para un lenguaje es un intérprete para dicho lenguaje escrito en ese lenguaje.

La forma más simple de un meta-intérprete para Prolog es, obviamente:

```
SOLVE(G) :- G.
```

La semántica de los operadores usados en el lenguaje puede hacerse explícita como en el caso de la conjunción, representada por

```
SOLVE(true).
SOLVE(G) :- CLAUSE(G,C),SOLVE(C).
SOLVE((A,B)) :- SOLVE(A),SOLVE(B).
```

donde se utiliza la primitiva CLAUSE para indicar que existe una cláusula cuya cabeza unifica con G, y cuyo resolvente es C.

Los dos primeros casos resuelven los goals atómicos mientras que el tercero se ocupa de definir al operador de conjunción dándole el significado usual.

Una forma más general de resolver el caso anterior está basada en la reescritura del objetivo hasta llegar a la cláusula *true*, por ejemplo:

```
SOLVE(true).
SOLVE(G) :- REWRITE(G,NG),SOLVE(NG).

REWRITE(A,true) :- SYSTEM(A),A.
REWRITE(G,C) :- CLAUSE(G,C).
REWRITE((true,B),B).
REWRITE((A,B),(RA,B)) :- REWRITE(A,RA).
```

De esta manera pueden tratarse predicados del sistema, para los cuales no existen cláusulas que los tengan como cabeza.

Mediante este mecanismo pueden escribirse meta-intérpretes para lenguajes provistos con un conjunto más amplio de operadores. Precisamente esta idea es la seguida en la construcción del meta-intérprete de API-2. Por ejemplo, para el operador de corrutinaje se tienen las reglas:

```
REWRITE((A\B),(RA\RB)) :- REWRITE(A,RA),REWRITE(B,RB).
REWRITE((true\B),B).
REWRITE((A\true),A).
```

3.2 Montaje de acciones sobre el meta-intérprete.

En las reglas del meta-intérprete pueden introducirse acciones. De esta manera, tales acciones tendrán efecto a medida que se desarrolla cada paso de la ejecución del programa.

Por ejemplo, para mostrar la ejecución paso a paso de un programa, bastará con introducir en algunos pasos de la demostración una acción que exhiba el goal que se pretende demostrar. Por ejemplo:

```
SOLVE(true).
SOLVE(G) :- EXH(G),REWRITE(G,NG),SOLVE(NG).

REWRITE(A,true) :- SYSTEM(A),EXH-SYS(A),A.
REWRITE(G,C) :- CLAUSE(G,C),EXH-SLV(C).
REWRITE((true,B),B).
REWRITE((A,B),(RA,B)) :- REWRITE(A,RA).
```

De esta forma, cada goal a ser demostrado será exhibido antes de proceder a su reescritura. Cada reescritura posible de un goal, será exhibida si ésta es un resolvente de una cláusula. Si el predicado es del sistema, esto será indicado.

Esta idea fue la usada para implementar las herramientas de puesta a punto sobre el meta-intérprete de API-2.

3.3 Implementación de las herramientas de puesta a punto.

Para implementar las funciones descriptas en el capítulo anterior fue necesario definir primero la numeración dada a los goals. Cuando se da un goal para ser evaluado, se le aplica la función NUMINIT. Esta se implementó con la siguiente variante: no se considera a los goals como expresiones binarias, sino que se tiene en cuenta la asociatividad de los operadores numerándose los sub-goals en forma correlativa respecto del operador de menor jerarquía, lo cual resulta más grato a la intuición. Esta variante no causa conflictos con lo especificado.

Al goal numerado le es aplicado el ciclo del meta-interprete. Cada vez que un átomo se reescribe como un resolvente de una cláusula, se numera el resolvente con NUM, implementada con igual variante que NUMINIT.

Lo anterior se refleja en las cláusulas:

```
SOLVE(G) :- NUMINIT(G,Gn),SOLVEN(Gn).
SOLVEN(Gn) :- EXH-FST(Gn),REWRITE(Gn,RGn),SOLVEN(RGn).

REWRITE(A-n,RAn) :- L-CLAUSE(A,L),MEMBER(C,L),
                    EXH-SLV(C),NUM(C,n,RAn).
```

El predicado L-CLAUSE devuelve una lista con las cláusulas del programa que resuelven A. Cada reintento de demostración de A se reflejará en una re-invocación al predicado MEMBER. Asociando a esta re-invocación una acción adecuada se pueden exhibir los reintentos de demostraciones de A. Por otra parte, una falla en MEMBER implica que no hay más cláusulas que resuelvan A. Otra vez, una acción adecuada asociada a esta situación permite exhibir la falla de A.

Las cláusulas de reescritura correspondientes a los demás operadores son las del meta-intérprete, aplicadas a goals numerados.

Para implementar las herramientas que suspenden la traza en un intervalo de la ejecución (JMP, SKIP, GOPRED, DEMON), se modifica el predicado EXH-FST de manera tal que éste permita al usuario introducir los comandos correspondientes a tales herramientas. Los comandos introducidos modifican una variable de estado que es global al sistema y determina si debe mostrarse o no información en cada paso de demostración (la consulta sobre si debe o no mostrarse la información, la hace EXH-FST).
Mayor detalle puede encontrarse en el código Prolog que se anexa, y que no es excesivamente engorroso.

En cuanto a la facilidad de SPY, ésta se considera independiente de la traza, ya que no es necesario mostrar el contexto (numeración) asociado a cada goal en tal caso. De todas formas, la idea utilizada para su implementación es la misma que la anterior, usando los predicados EXH-FST y EXH-SLV en los casos adecuados.

3.4 Otras herramientas de puesta a punto implementadas.

3.4.1 Ejecuciones parciales ("Query the user").

La facilidad de "query the user" habilita al usuario a introducir instancias que se asumirán como válidas de predicados que han fallado. Usando la estructura del meta-intérprete de 3.1 esto puede escribirse de la siguiente forma:

```
REWRITE(A,RA) :- CLAUSE(A,RA).
REWRITE(A,RA) :- QUERY(A,RA).
```

es decir: la falla de CLAUSE provoca la consulta al usuario mediante el predicado QUERY, donde RA debe ser una instancia de A que será tomada como válida.
Esto se implementa usando listas de cláusulas como en la sección anterior. QUERY tiene lugar cuando se tienen listas vacías de resolventes de A.

En términos de la semántica formal definida en los capítulos 1 y 2, tales consultas provocarían una modificación del árbol de ejecución, pues en este caso la modificación no incumbiría solamente a la acciones asociadas a los nodos, sino que implicaría una variación en la estructura misma del árbol, esto es, el agregado de nuevos nodos correspondientes a las nuevas instancias válidas de los predicados introducidas y a partir de allí, nuevas ramas para la derivación de goals que los incluyan.

Este tema no fue abordado en la especificación de las herramientas del capítulo anterior, y queda como una posible extensión de este trabajo.

3.4.2 Debugger declarativo.

Un programa puede ser considerado como una colección de hipótesis arbitrariamente complejas y su conducta como una consecuencia de estas hipótesis. Debido a que en un momento determinado, no es posible conocer todas las consecuencias de un tal conjunto de supuestos, no se puodrán anticipar todas las posibles conductas de un programa dado. Es por esta razón que el problema de "debugging" debe ser considerado como de particular importancia en cualquier ambiente de programación [Shapiro 82].

Es importante la existencia de debuggers "declarativos", en el sentido de que puedan ser utilizados sin tener un conocimiento total del comportamiento del sistema. En [Lloyd 87] se propone un sistema declarativo de diagnóstico de errores para programas lógicos que usen sintaxis extendida y facilidades de control adicionales.

Siguiendo la propuesta recién mencionada, se implementó la parte del debugger declarativo que detecta los errores de inconsistencia de programas, sin considerar operadores de control alternativos ni errores de incompletitud en programas.

El código del debugger se anexa a la documentación entregada, y como podrá observarse, éste es muy sencillo y fácil de comprender.

Capítulo 4.

Conclusiones.

Se ha especificado formalmente un lenguaje de programación lógica extendido con un conjunto significativo de operadores de control. Aunque los aspectos de control de los lenguajes lógicos son de formalización frecuentemente engorrosa, la conseguida en este trabajo resulta clara aún cuando mantiene rigor formal.

El formalismo se ha revelado útil para especificar también acciones asociadas a cada paso de ejecución de un programa, lo cual se usó para definir también formalmente comandos para herramientas de puesta a punto. En este caso, la especificación es, sin embargo, algo tediosa por tratarse esencialmente de recorridos de árboles n-arios. Sin embargo la estructura subyacente fue útil para comprender el significado de cada comando. Igualmente, nuevas herramientas de puesta a punto pueden definirse con facilidad especificando su efecto sobre los árboles de ejecución con acciones.

La estructura de la especificación se refleja naturalmente en la de la implementación, lo cual hace a ésta también fácil de comprender.

Las extensiones del trabajo deberían contemplar la especificación e implementación de nuevas herramientas de puesta a punto con acciones que permitan modificar el árbol de ejecución del programa y de un mecanismo amigable que permita al usuario definir sus propias herramientas de puesta a punto.

Agradecimientos.

A Tato, que me ayudó como nadie con esto.

A Juan, por la cantidad de ideas que me dió y porque es lo más.

A Jorge Vidart, por su paternal dirección durante este semestre.

A Raúl Carnota.

Y un "gracias" muy especial para Daniel Pambrejo, ya sabe él por qué.

Apéndice I

Definición de la función τ_{na}

$\tau_{na}: \mathbb{P} \longrightarrow (\mathbb{G} \longrightarrow \mathbb{T}_a)$

(a) $\tau_{na}(P)(\leftarrow \text{TRUE-K}) = \langle \text{TRUE-K}, [] \rangle$

(b) $\tau_{na}(P)(\leftarrow \text{FAIL-K}) = \langle \text{FAIL-K}, [], \text{NOP} \rangle$

(c) $\tau_{na}(P)(\leftarrow \text{A-K}) =$ IF K = 0.J
THEN $\langle \text{TRUE-J}, [], \text{NOP} \rangle$
ELSE $\langle A, \mathcal{L}, \text{NOP} \rangle$

donde

$$\mathcal{L} = [\langle \text{CNJa}(\tau_{na}(P)(\leftarrow \text{NUM}(S_1\theta_1, K)), A\text{-0.K}, P), \theta_1 \rangle, \ldots, \langle \text{CNJa}(\tau_{na}(P)(\leftarrow \text{NUM}(S_n\theta_n, K)), A\text{-0.K}, P), \theta_n \rangle]$$

si el conjunto $\langle B_1 \leftarrow S_1, \ldots, B_n \leftarrow S_n \rangle$ (ordenado según su orden en P) de las cláusulas de P tales que A y B_i unifican con mgu θ_i es no vacío, y

$$\mathcal{L} = [\langle \langle \text{FAIL-K}, [], \text{NOP} \rangle, id \rangle]$$

en otro caso.

(d) $\tau_{na}(P)(\leftarrow (A;B)\text{-K}) =$
$\langle (A;B)\text{-K}, [\langle \tau_{na}(P)(\leftarrow A\text{-K.1}), id, \text{NOP} \rangle, \langle \tau_{na}(P)(\leftarrow B\text{-K.2}), id, \text{NOP} \rangle] \rangle$

(e) $\tau_{na}(P)(\leftarrow (A \hat{} B)\text{-K}) =$
$\langle (A \hat{} B)\text{-K}, [\langle \text{SNPa}(\tau_{na}(P)(\leftarrow A\text{-K.1}), B\text{-K.2}, P), id, \text{NOP} \rangle] \rangle$

(f) $\tau_{na}(P)(\leftarrow (A,B)\text{-K}) =$
$\langle (A,B)\text{-K}, [\langle \text{CNJa}(\tau_{na}(P)(\leftarrow A\text{-K.1}), B\text{-K.2}, P), id, \text{NOP} \rangle] \rangle$

(g) $\tau_{na}(P)(\leftarrow (A \backslash B)\text{-K}) =$
$\text{CORa}(\tau_{na}(P)(\leftarrow A\text{-K.1}), \tau_{na}(P)(\leftarrow B\text{-K.2}), K)$

(h) $\tau_{na}(P)(\leftarrow (\text{NOT } A)\text{-K}) = \text{NEGa}(\tau_{na}(P)(\leftarrow A\text{-K}))$

■

La definición de las funciones CNJa, SNPa, CORa y NEGa no se dará aquí por ser ésta sólo una simple extensión de las respectivas CNJn, SNPn, CORn y NEGn

Apéndice II

Definición de las funciones asociadas a los comandos de puesta a punto.

TRACE: $\mathbb{T}_\alpha \longrightarrow \mathbb{T}_\alpha$

TRACE(A)
precondición: todos los nodos del árbol deben tener asociada la función NOP.

(a) TRACE (<TRUE,[],acc>) = <TRUE,[],acc>

Los átomos TRUE se "filtrarán" durante la traza del programa, sin mostrar ningún tipo de información asociada en estos casos,ya que un TRUE implica la finalización de una demostración el cual se da a conocer al encontrar a un objetivo cerado (regla (c)).

(b) TRACE(FAIL,[],acc>) = <FAIL,[],acc>

Los átomos FAIL también se filtrarán durante la ejecución de la traza, ya que la falla en la prueba de un objetivo se muestra con la regla (d)

(c) TRACE(<A-o.K.$\mathcal{L}$,acc>) = <A-o.K,$\mathcal{L}$,(acc;EXH-SCS(A-K))>

Como se explicó en la sección anterior, un nodo de la forma A-o.K corresponde a la finalización de la prueba de A (ver punto (c) en la definición de τ_n)

(d) TRACE(<A-K,[<<FAIL,[]>,id>],acc>) =
<A-K,[<<FAIL,[]>,id>],(acc;EXH-NOSLV(A-K))>

Este es el caso en que el átomo A no tiene resolventes en el programa. Por lo tanto debe mostrarse que no hay cláusulas en el programa cuyas cabezas unifiquen con A (ver punto (c) en la definición de τ_n).

(e) TRACE(<R,[<<R_1,$\mathcal{L}_1$,acc$_1$>θ_1>,...,<<R_n,$\mathcal{L}_n$,acc$_n$>,θ_n>],acc> =
<R,[<TRACE(<R_1,$\mathcal{L}_1$,(EXH-SLV(R_1);acc$_1$)>),θ_1>,
...<TRACE(<R_n,$\mathcal{L}_n$,(EXH-FAIL(R_{n-1});EXH-SLV(R_n);acc$_n$)>),θ_n>],
(acc;EXH(R))>

Este es el caso mas complejo en la definición de la función TRACE: para cada subárbol se muestra su raiz como resolvente (de su padre) y la falla del anterior. Esto queda claro si se tiene en cuenta que los árboles con los que se está tratando son árboles or y por lo tanto cualquier subárbol de uno dado es una derivación de la raiz del mismo. entonces, si el recorrido del árbol se hace en profundidad, el inicio del recorrido de una rama significa la falla de la rama precedente.

SKIP : $\mathbb{T}_\alpha \times \mathbb{N}_p \longrightarrow \mathbb{T}_\alpha$

SKIP(A,n)
precondición: existe un nodo con p-número asociado n en el árbol y todos los nodos del árbol cuya raíz es ese nodo tienen asociada la acción EXH.

```
SKIP(<R-K,ℒ,acc>,n) =
        IF (K = n)
        THEN IF MEMBER (0.n, ℒ)
                THEN <R-K,IGN(ℒ,0.n),EXH(R)>
                ELSE <R-K,NOEXH(ℒ),(EXH(R);EXH-FAIL(R))>
        ELSE <R-K,SKP(ℒ,n),acc>
```
■

SKP: $\mathbb{L} \times \mathbb{N}_p \longrightarrow \mathbb{L}$

```
SKP([],n) = []
SKP([<<R-K,ℒ,acc>,θ>!ℒ1],n) =
                        IF (K = n)
                        THEN [SKIP(<R-K,ℒ,EXH(R-K)>,θ>,n)!ℒ1]
                        ELSE [<<R-K,ℒ.acc>,θ>!SKP(ℒ1,n)]
```
■

JMP:$\mathbb{T}_\alpha \times \mathbb{N}_p \times \mathbb{N}_p \longrightarrow \mathbb{T}_\alpha$

JMP(A,n1,n2)
precondición: existe un nodo con p-número asociado n1 en el árbol y todos los nodos del árbol a partir de tal nodo y el nodo con p-número n2 (si existe) tienen asociada la acción EXH, en caso de no existir un nodo con p-número n2 después del tal nodo, todos los nodos del árbol posteriores a n1 tienen asociada la acción EXH.

```
JMP(<R-K,[<t,θ>!ℒ],acc>,n1,n2) =
     IF (K = n1)
     THEN <R-K,IGN(ℒ,n2),EXH(R-K)>
     ELSE IF MEMBER(n1,t)
          THEN IF MEMBER(n2,t)
               THEN <R-K,[<JMP(t),θ>!ℒ],EXH(R-K)>
               ELSE ADD(<JMP(t),θ>,(OP-ELIM(<R-K,ℒ,EXH(R-K)>,n2)
          ELSE ADD(<t,θ>,JMP(<R-K,ℒ,acc>,n1,n2)
```
■

OP-ELIM:$\mathbb{T}_\alpha \times \mathbb{N}_p \longrightarrow \mathbb{T}_\alpha$

```
OP-ELIM(<R-K,[],acc>,n) = IF (K = n)
                              THEN <R-K,[],EXH(R-K)>
                              ELSE <R-K,[],NOP>

OP-ELIM(<R-K,[<t,θ>!ℒ],acc>,n) =
      IF (K = n)
      THEN (<R-K,[<t,θ>!ℒ],EXH(R-K)>
      ELSE ADD(<NOEXH(t),θ>,OP-ELIM(<R-K,ℒ,NOP>,n))
```
■

GOPRED:$\mathbb{T}_a \times \mathbb{N}_p \times \mathbb{S} \longrightarrow \mathbb{T}_a$

GOPRED(A,n,p)
precondición: existe un nodo con p-número asociado n en el árbol y todos los nodos del árbol a partir de tal nodo y el nodo con goal asociado p (si existe) tienen asociada la acción EXH, en caso de no existir un nodo con goal p después del tal nodo, todos los nodos del árbol posteriores a n tienen asociada la acción EXH.

```
GOPRED(<R-K,[<t,θ>|E],acc>,n,p) =
      IF (K = n)
      THEN <R-K,IGNP(E,p),EXH(R-K)>
      ELSE IF MEMBER(n,t)
            THEN IF MEMBERP(p,t)
                 THEN <R-K,[<GOPRED(t,n,p),θ>|E],EXH(R-K)>
                 ELSE
                   ADD(<GOPRED(t,n,p),θ>,(OP-ELIMP(<R-K,E,EXH(R-K)>,p)
            ELSE ADD(<t,θ>,GOPRED(<R-K,E,acc>,n,p)
■
```

OP-ELIMP:$\mathbb{T}_a \times \mathbb{S} \longrightarrow \mathbb{T}_a$

```
OP-ELIMP(<R-K,[],acc>,p) = IF (R = p)
                           THEN <R-K,[],EXH(R-K)>
                           ELSE <R-K,[],NOP>

OP-ELIMP(<R-K,[<t,θ>|E],acc>,p) =
      IF (R = p)
      THEN (<R-K,[<t,θ>|E],EXH(R-K)>
      ELSE ADD(<NOEXH(t),θ>,OP-ELIMP(<R-K,E,NOP>,p))
■
```

DEMON:$\mathbb{T}_a \times \mathbb{N}_p \times \mathbb{C} \longrightarrow \mathbb{T}_a$

DEMON(A,n,c)
precondición: existe un nodo con p-número asociado n en el árbol y todos los nodos del árbol a partir de tal nodo y el primer nodo tal que su goal asociado cumple c (si existe) tienen asociada la acción EXH, en caso de no existir un nodo cuyo goal cumpla c después del tal nodo, todos los nodos del árbol posteriores a ntienen asociada la acción EXH.

```
DEMON(<R-K,[<t,θ>|E],acc>,n,p) =
      IF (K = n)
      THEN <R-K,IGNC(E,c),EXH(R-K)>
      ELSE IF MEMBER(n,t)
            THEN IF MEMBERC(c,t)
                 THEN <R-K,[<DEMON(t,n,c),θ>|E],EXH(R-K)>
                 ELSE
                   ADD(<DEMON(t,n,c),θ>,(OP-ELIMC(<R-K,E,EXH(R-K)>,c)
            ELSE ADD(<t,θ>,DEMON(<R-K,E,acc>,n,c)
■
```

```
OP-ELIMP:𝕋σ × ℂ ⟶ 𝕋σ

      OP-ELIMC(⟨R-K,[],acc⟩,c) = IF c(R)
                                 THEN ⟨R-K,[],EXH(R-K)⟩
                                 ELSE ⟨R-K,[],NOP⟩

      OP-ELIMC(⟨R-K,[⟨t,θ⟩|ℒ],acc⟩,c) =
            IF c(R)
            THEN (⟨R-K,[⟨t,θ⟩|ℒ],EXH(R-K)⟩
            ELSE ADD(⟨NOEXH(t),θ⟩,OP-ELIMC(⟨R-K,ℒ,NOP⟩,c))
■
```

```
SPY:𝕋σ × ℙ ⟶ 𝕋σ

SPY(A,p)

      SPY(⟨R-K,[],acc⟩,p) = IF INST(R,p)
                            THEN ⟨R-K,[],EXH(R-K)⟩
                            ELSE ⟨R-K,[],acc⟩

      SPY(⟨R-K,[⟨⟨R1,ℒ1,acc1⟩θ1⟩,...,⟨⟨Rn,ℒn,accn⟩,θn⟩],acc⟩,p) =
            IF (INST(R,p) and K ≠ 0..J)
            THEN ⟨R-K,[⟨SPY(⟨R1,ℒ1,(EXH-SLV(R1);acc1),p)⟩θ1⟩,
                  ...,⟨SPY(⟨Rn,ℒn,(EXH-SLV(Rn);accn)⟩,p),θn⟩],
                  (EXH(R);acc)⟩
■
```

donde INST es una función booleana que devuelve *true* si el primer parámetro es instancia del segundo.

---

funciones auxiliares

```
IGN:𝕋σ × ℕp ⟶ 𝕋σ

      IGN    ([⟨⟨R-K,ℒ,acc⟩,θ⟩|ℒ1],n)    =
                  IF (K = n)
                  THEN [⟨⟨R-K,ℒ,EXH-SCS(R-K)⟩,θ⟩|ℒ1]
                  ELSE IF MEMBER (n, ℒ)
                        THEN [⟨⟨R-K,IGN(ℒ,n),NOP⟩,θ⟩|ℒ1]
                        ELSE [⟨⟨R-K,NOEXH(ℒ),NOP⟩,θ⟩|IGN(ℒ1,n)]
■

IGNP:𝕋σ × ℙ ⟶ 𝕋σ

      IGN    ([⟨⟨R-K,ℒ,acc⟩,θ⟩|ℒ1],p)    =
                  IF (R = p)
                  THEN [⟨⟨R-K,ℒ,EXH-SCS(R-K)⟩,θ⟩|ℒ1]
                  ELSE IF MEMBERp (p, ℒ)
                        THEN [⟨⟨R-K,IGNP(ℒ,p),NOP⟩,θ⟩|ℒ1]
                        ELSE [⟨⟨R-K,NOEXH(ℒ),NOP⟩,θ⟩|IGNP(ℒ1,p)]
■

IGNC:𝕋σ × ℂ ⟶ 𝕋σ

      IGNC   ([⟨⟨R-K,ℒ,acc⟩,θ⟩|ℒ1],c)    =
                  IF c(R)
                  THEN [⟨⟨R-K,ℒ,EXH-SCS(R-K)⟩,θ⟩|ℒ1]
                  ELSE IF MEMBERC (c, ℒ)
                        THEN [⟨⟨R-K,IGNC(ℒ,c),NOP⟩,θ⟩|ℒ1]
                        ELSE [⟨⟨R-K,NOEXH(ℒ),NOP⟩,θ⟩|IGNC(ℒ1,c)]
■
```

```
MEMBER:ℕp × 𝕃 ⟶ 𝕃

      MEMBER(n,[]) = false

      MEMBER(n,[<<R-K,ℒ,acc>,θ>¦ℒ1]) = (n = K) or
                                        MEMBER (n,ℒ) or MEMBER(n,ℒ1)
■

MEMBERP:ℙ × 𝕃 ⟶ 𝕃

      MEMBER(p,[]) = false

      MEMBER(p,[<<R-K,ℒ,acc>,θ>¦ℒ1]) = (p = R) or
                                        MEMBERP (p,ℒ) or MEMBERP(p,ℒ1)
■

MEMBERC:ℂ × 𝕃 ⟶ 𝕃

      MEMBERC(c,[]) = false

      MEMBER(c,[<<R-K,ℒ,acc>,θ>¦ℒ1]) = c(R) or
                                        MEMBERC (c,ℒ) or MEMBERC(c,ℒ1)
■

NOEXH:𝕃 ⟶ 𝕃

      NOEXH([]) = []
      NOEXH([<<R,ℒ,acc>,θ>¦ℒ1])      =
                [<<R,NOEXH(ℒ),NOP>,θ>¦NOEXH(ℒ1)]
■
```

---

notación:

𝕃 denota el conjunto de las listas.
ℂ denota el conjunto de las condiciones.
ℙ denota el conjunto de los predicados.

Referencias.

[CDKMV 88]
Api-2: Un Meta-Ambiente para Programación Lógica.
R. Carnota, J. Diaz, A. Kvitca, A. Monteiro, J. Vidart.
aceptado para publicación, 1988.

[Knuth 68]
The Art of Computer Programming: Fundamental Algorithms.
D. Knuth
Addison Wesley, 1968.

[Lloyd 84]
Foundations of Logic Programming.
J. W. Lloyd.
Springer Verlag, 1984.

[Lloyd 87]
Declarative Error Diagnosis.
J.W. Lloyd.
New Generation Computing 5, 1987.

[Shapiro 82]
Algorithmic Program Debugging.
E. Shapiro.
PhD Thesis, Yale University, 1982.