META-CONTROL y PROGRAMACION LOGICA

Adolfo Kvitca
Universidad de Buenos Aires (UBA)
Escuela Superior Latinoamericana de Informática (ESLAI)
Casilla de Correo 3193 - 1000: Buenos Aires - Argentina

Resumen:

En el presente trabajo se describe un formalismo que permite expresar el componente de control de un programa lógico en forma declarativa. Esta representación explícita de la información de control, independiza al sistema del orden parcial de las cláusulas y permite lograr eficiencia en la ejecución, sin sacrificar la inteligibilidad del programa.

La propuesta ilustra principios generales según los cuales diferentes reglas de control puedan ser usadas en forma conjunta, y está inspirada en las ideas sugeridas por R.Davis sobre meta-reglas, pero aplicadas al ámbito de la demostración de teoremas. El sistema permite, a diferencia de otras propuestas, introducir un componente heurístico breadth-first en la búsqueda y expresar diferentes reglas de inferencia de manera natural.

(Nota: La lectura del trabajo presupone conocimientos sobre aspectos relativos al control en programación lógica [KOW/81]).

Este trabajo se ha realizado en el marco del proyecto "Ambiente de Programación Inferencial (API)", financiado por el Programa Nacional de Informática y Electrónica (537-001) SECYT-Argentina

1. Introducción:

En la programación tradicional un programa se realiza especificando las operaciones necesarias para resolver el problema, es decir, diciendo cómo debe ser resuelto, pero quedando usualmente implícitos los supuestos en que el programa está basado. En cambio, en la programación lógica, los supuestos son explícitos, y la secuencia de uso de las operaciones pasa a ser implícita.

La idea clave de la programación lógica es programar mediante descripciones, cambiando de esta manera la perspectiva prescriptiva de la programación hacia una perspectiva declarativa, donde es más importante especificar "qué" debe hacer el programa y no "cómo" debe hacerlo. Por lo tanto, los algoritmos se pueden considerar integrados por un componente lógico, (que especifica "qué" es lo que se debe hacer) y un componente de control (que determina "cómo" debe ser hecho).

La eficiencia de un algoritmo puede ser mejorada modificando el componente de control sin cambiar la lógica y por lo tanto sin cambiar el significado del algoritmo. Un programa puede incluir en el componente lógico lo que otro incluye en el componente de control, pero en general, son preferibles los algoritmos que, para aumentar la eficiencia, ponen el énfasis en el componente de control. Esto permite que el componente lógico sea más claro y que represente, en forma obviamente correcta, el problema y el conocimiento que puede ser usado para su solución. La hipótesis es que, cuando se desarrollan programas bien estructurados mediante refinamientos sucesivos y sistemáticos, el componente lógico debe especificarse antes que el componente de control. Es decir que, primero se debe representar y describir el problema y luego, controlar la búsqueda para que sea resuelto en un lapso razonable.[KOW/79]

2. El Problema del Control en Prolog:

Prolog ha ganado popularidad como lenguaje de programación lógica, [CLO/84],[STE/86] pero sin embargo, no cumple con las altamente deseables características enunciadas. La causa es su rígida estrategia de control para realizar las deducciones ("backtracking"), lo que origina, entre otros inconvenientes, serialidad, ineficiencia y ciclos.

La noción serial impuesta por el backtracking, obliga a tener en cuenta el orden de las cláusulas y de los literales dentro de ellas. Además, el predicado "cut", brindado al programador para que pueda controlar el desarrollo del árbol, es un mecanismo de muy bajo nivel. (Se dice que el "cut" es a la programación lógica como el "go to" a la programación imperativa). Esto no significa que no deben existir mecanismos para decidir qué partes del árbol desechar o probar primero, pero nunca a expensas de la inteligibilidad del programa.

Cuando los programadores se independicen de estas deficiencias, sus intuiciones se moverán en un nivel superior, cambiando la perspectiva empírica (que fomenta "local-patching": cómo hago que esto se haga a continuación), a una perspectiva más general, que considere la estrategia de control como información acerca del nivel objeto.

3. Diferentes Facilidades de Control:

Para solucionar estos inconvenientes se han implementado diversos sistemas con distintas facilidades de control, que corresponden en mayor o menor grado a diferentes propuestas. Las mismas se pueden clasificar en:

.a)Control Incorporado al Programa: implementado mediante anotación de variables (Ej:IC-PROLOG [CLA/80]), metapredicados en cláusulas (Ej: freezer, PROLOG II [COL/82]), distintas conectivas para conjunciones (Ej: EPILOG [POR/84],[PER/84]).

.b)Control Específico: Enfoque seguido principalmente por Bundy [BUN/81], que consiste básicamente en la construcción de intérpretes específicos para el dominio de aplicación.

.c)Control Separado del Programa: utilización de aserciones independientes de las cláusulas del programa para expresar distintos aspectos de control (Ej:MU-PROLOG [NAI/84c], METALOG [DIN/84], CERT [GAL/80]. Naish muestra cómo, para algunos casos, este control puede ser generado automáticamente [NAI/83]).

Estos sistemas proveen algunas soluciones, pero: a) no ofrecen un tratamiento uniforme del problema del control, b) se basan únicamente en un mecanismo depth-first (que no es completo), y c) realizan backtracking "inteligente" utilizando solamente propiedades sintácticas (METALOG provee un mecanismo más general (aunque no lo suficiente), donde se puede especificar a qué resolvente volver en caso de falla).

4. Representación Explícita del Control:

La representación explícita del control en forma separada del componente lógico satisface las caracteristicas deseadas.
La ventaja de especificar el control en forma separada del programa consiste en que el control no está restringido a una cláusula, sino que está relacionado con todo el espacio de búsqueda y permite la expresión de estrategias más generales. El componente lógico del programa es más claro y se puede experimentar con distintos tipos de control sin cambiarlo.

Siguiendo la línea expuesta anteriormente, los programas deben desarrollarse en forma declarativa pero, desde un punto de vista práctico, se puede observar que un estilo puramente declarativo es inadecuado para la demanda de muchas aplicaciones: los procedimientos de entrada/salida y otras acciones que contienen efectos colaterales, se deben realizar eficientemente y en secuencias cuidadosamente diseñadas.

Esta aparente contradicción está originada en el hecho de que hay información que el programador conoce, pero que en el programa está sólo en forma implícita. La hipótesis es que la ineficiencia no depende de una estrategia de deducción particular, sino que es intrínseca al requerimiento de que toda la lógica del programa sea expresada de una manera uniforme y no diferenciada.

5. Propuesta para la Representación Explícita del Control:

La propuesta consiste, entonces, en que el programador especifique explícitamente los siguientes aspectos:

a) Información correspondiente a la selección de la regla:
Las condiciones, relacionadas con la selección de una regla, deben explicitarse para poder evaluarlas (cuando sea posible) estrictamente antes de las consecuencias de la misma:

selector:predicado <- sub-objetivos.

(donde selector y sub-objetivos consisten en una conjunción -posiblemente nula- de predicados). La regla PROLOG equivalente sería: predicado <- selector,sub-objetivos, pero con la diferencia fundamental que, si bien el selector se evalua preferentemente antes de los sub-objetivos, los literales en cada uno de ellos se evaluan mediante una estrategia mas flexible.

b) Las reglas excluyentes:
El conocimiento del programador acerca de la imposibilidad de invocar más de una regla para un predicado particular se expresa de la siguiente manera:

selector:predicado <-> sub-objetivos.

Esta información puede ser usada para controlar que realmente las distintas reglas sean excluyentes, para lograr eficiencia -en caso de satisfacerse una regla no es necesario intentar probar el resto de las alternativas- y finalmente, para expresar claramente la utilización de reglas "default", en el sentido que se utilizan sólo si no hay otra alternativa. La regla PROLOG equivalente sería: predicado:selector,!,sub-objetivos.

Ej: Mediante un selector "else" que tiene la propiedad de tener prioridad mínima para la selección de reglas (es decir que la regla que contiene el selector "else" se utilizará sólo después de haber intentado el resto), se puede implementar naturalmente, una estructura "if-then-else":

```
if-1(x):p(x) <-> then-1(x).
if-n(x):p(x) <-> then-n(x).
else   :p(x) <-> then(x)  .
```

(if-# una serie de condiciones y then-# una serie de acciones).

Ej: La definición de factorial sería entonces:

```
                fact(0 1) <->.
      mayor(X 0):fact(X Y) <->
                +(X1 1 X),fact(X1 Y1),*(Y1 X Y).
```

(En particular, a pesar de la exigencia de evaluar el predicado "mayor" antes del resto, es posible -mediante reglas de control adecuadas-, posponer su evaluación hasta que las variables esten instanciadas).

c) Las reglas que corresponden a acciones:

Consisten en una secuencia ordenada de acciones a tomar cuando se satisface el predicado, o el mismo es requerido en algun sub-objetivo:

```
selector:predicado -> acciones.
```

La utilización de "acciones" también esta gobernada por la regla de exclusión, pero tiene la propiedad adicional de siempre satisfacerse la invocación a la misma, eventualmente mediante una acción nula. (Se asume implícito que en toda acción "p", existe una regla de la forma: else:p(x) ->.). Esto permite implementar acciones "transparentes" ("demonds"), por ejemplo:

```
trace:resolvente(X) -> imprimir("resolvente:",X)
```

d) Los aspectos referentes al control:

La idea consiste en evitar la explosión combinatoria mediante un demostrador de teoremas que piensa acerca de qué está intentando hacer. Para esto los objetivos, acciones y razonamientos deben estar explícitamente representados en una forma manipulable por el proceso de deducción. En rigor, este problema interno de control, puede ser formalizado usando afirmaciones y reglas de la misma manera que se hace en el dominio externo, considerando el problema de deducción desde un punto de vista "meta".

A este aspecto está dedicado el resto del artículo, y consiste en la definición de un formalismo para tratar distintos aspectos referentes al control de una manera explícita y uniforme.

(Nota: En los ejemplos se usará X1.X para denotar la lista con primer elemento X1 y resto X, y X..Y para la lista resultado de concatenar la lista X con la lista Y).

6. Una Visión "Meta" del Problema de Control:

La posibilidad del sistema de manipular aspectos relativos a la dualidad conocimiento / meta-conocimiento, teoría / meta-teoría, lenguaje / meta-lenguaje, permite expresar conocimiento extensivamente y utilizarlo eficientemente [BOW/82], pudiéndose expresar aspectos de gran interés en diversas áreas de la Inteligencia Artificial: en Deducción Automática se implementan estrategias que utilizan meta-teoremas para acortar las pruebas en el nivel objeto (Ej:paramodulación); en Representación del Conocimiento, para aumentar el poder expresivo de los lenguajes de representación, (Ej: puede servir para formalizar creencias, razonamiento "default", no monotonía, etc); en Resolución de Problemas para controlar el espacio de búsqueda. [AIE/86]

A continuación se expondrá un formalismo para expresar el componente de control en forma declarativa, que está inspirado en las ideas de Davis sobre meta-reglas [DAV/80], pero aplicadas al ámbito de la demostración de teoremas. El mismo, pretende introducir principios generales donde diferentes reglas de control puedan ser usadas en forma conjunta. (El análisis se restringe a estrategias lineales (SLD) que son eficientes y, en particular, completas para cláusulas de Horn [LLO/83], pero, es aplicable a estrategias de resolución mas generales).

La idea fundamental del mismo, consiste en identificar en el proceso de deducción, tres fases distintas, y proveer mecanismos para manipular cada una de ellas:

a) recuperación: alguna propiedad de las cláusulas es usada para elegir un subconjunto (RECUPERAR-cláusulas).

b) refinamiento: las ramas, reglas y literales se reordenan y filtran (ORDENAR-ramas,reglas,literales).

c) ejecución: se obtiene un nuevo resolvente (obtener-RESOLVENTE).

El algoritmo corresponde básicamente al siguiente esquema:

```
Demo(teoría,nil.árboles)      <- .
Demo(teoría,árbol1.árboles) <-
        ORDENAR-literales(árbol1,árbol1´),
        RECUPERAR-cláusulas(teoria,árbol1´,cláusulas),
        ORDENAR-cláusulas(cláusulas,cláusula1.cláusulas´),
        obtener-RESOLVENTE(árbol1´,cláusula1,resolvente),
        armar-arbol(árbol1´,cláusulas´,árbol1´´),
        ORDENAR-ramas(resolvente.árbol1´´,árboles,árboles´),
        Demo(teoria,árboles´).
```

Observar que siempre se selecciona el primer elemento de cada estructura, pues ésta se encuentra constantemente ordenada y que la utilización de los predicados de las distintas fases no deben instanciar variables de los predicados del nivel objeto.

Los sistemas existentes sólo atacan la fase de refinamiento, y lo hacen en forma parcial pues, están desarrollados, en mayor o menor medida, alrededor de las siguientes ideas:

a) durante la ejecución "forward" debe decidirse qué literal resolver (elección AND) y qué cláusula usar para resolverlo (elección OR).

b) durante el "backtracking", en caso de falla, debe decidirse qué pasos deshacer y cómo seguir.

(En particular, PROLOG, elige el literal más a la izquierda y la primera cláusula que unifique (en orden de escritura) y, en caso de falla, realiza "backtracking" al resolvente precedente en orden cronológico, eligiendo la próxima cláusula que unifique con el mismo literal)

Siendo las principales deficiencias las siguientes:

a) no ofrece un tratamiento uniforme del problema del control (los distintos mecanismos se implementan en formas diferentes, (especialmente el componente "forward" y "backward"), lo que hace difícil, y en muchos casos imposible, la cooperación entre ellos).

b) se basa únicamente en un mecanismo depth-first (que además de no ser completo, utiliza la información heurística en forma parcial, lo que muchas veces determina una complejidad exponencial [PEA/84])

c) la verdadera inteligencia (heurística dependiente del dominio), es puesta en la componente "forward" y realiza "backtracking inteligente" utilizando solamente propiedades sintácticas. [BRU/84],[PIE/82]

d) una vez realizada la elección (AND) de un literal, ésta no tiene posibilidad de ser revisada

6.1. Refinamiento:

La presente propuesta soluciona esas deficiencias y consiste básicamente en considerar al proceso de refinamiento únicamente como depuración y reordenamiento (de un árbol de busqueda particular), basados en aspectos heurísticos provistos por el usuario o generados automáticamente. Este conocimiento heurístico se expresa mediante meta-reglas, que pueden tener un alcance global o local y, que pueden incluir invocación dirigida mediante "content-reference" [DAV/79]. La heurística utilizada puede introducir un componente "breadth-first" en el recorrido del árbol (aunque "depth-first" se asume por "default").

A continuación se ilustrará, mediante una serie de ejemplos, las características de organización del árbol de busqueda, diseñado con el propósito de permitir la revisión de las elecciones "AND" y la incorporación de un componente "breadth-first", características distintivas con respecto a otros formalismos y que permite obtener demostraciones más cortas.

En cada nivel del árbol, se elige un literal y la cláusula correspondiente, dejando asociado al árbol las cláusulas no elegidas que se utilizarán posteriormente.

Ej: <- P(X),Q(Y),R(X)

P(a) <-. Q(c) <-. R(e) <-. S(g) <-.
P(b) <- Q(d) <-. R(f) <-.
P(Z) <- S(a).

Utilizando la "estrategia PROLOG" el árbol quedaría con 8 nodos:

(0) P(X),Q(Y),R(X)!X,Y
(1) Q(Y),R(X)!X/a,Y (2)P(X){P(b),P(Z)<-S(a)},Q(Y),R(X)!X,Y
(11) R(X)!X/a,Y/c (12)Q(Y){Q(d)},R(X)!X/a,Y
falla
(121) R(X)!X/a,Y/d
falla
(21) Q(Y),R(X)!X/b,Y (22)P(X){P(Z)<-S(a)},Q(Y),R(X)!X,Y
(211) R(X)!X/b,Y/c (212)Q(Y){Q(d)},R(X)!X/b,Y
falla
(2121) R(X)!X/b,Y/d
falla
(221) S(a),Q(Y),R(X)!X,Y
falla

Mediante "Backtracking Inteligente" el árbol sería de 6 nodos:

(0) P(X),Q(Y),R(X)!X,Y
(1) Q(Y),R(X)!X/a,Y (2)P(X){P(b),P(Z)<-S(a)},Q(Y),R(X)!X,Y
(11) R(X)!X/a,Y/c (12)Q(Y){Q(d)},R(X)!X/a,Y
falla
(21) Q(Y),R(X)!X/b,Y (22)P(X){P(Z)<-S(a)},Q(Y),R(X)!X,Y
(211) R(X)!X/b,Y/c (212)Q(Y){Q(d)},R(X)!X/b,Y
falla
(221) S(a),Q(Y),R(X)!X,Y
falla

En cambio, mediante la mejor elección permitida en formalismos tradicionales (no permiten revisión de la elección AND), sería de 5 nodos:

(0) P(X),Q(Y),R(X)!X,Y
(1) P(X),Q(Y)!X/e,Y (2) R(X){R(f)},P(X),Q(Y)!X,Y
(11) S(a),Q(Y)!X/e,Y
falla
(21) P(X),Q(Y)!X/f,Y
(211) S(a),Q(Y)!X/f,Y
falla

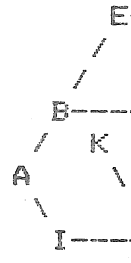
Pero el esquema propuesto permite, eligiendo adecuadamente, reducir el árbol a solo 4 nodos:

(0) P(X),Q(Y),R(X)!X,Y
(1) S(a),Q(Y),R(X)!X,Y (2)P(X){P(a),P(b)},Q(Y),R(X)!X,Y
falla
(21) P(X){P(a),P(b)},Q(Y)!X/e,Y
(22)R(X){R(f)},P(X){P(a),P(b)},Q(Y)
falla
(221) P(X){P(a),P(b)},Q(Y)!X/f,Y
falla

En el ejemplo se puede notar una semejanza del árbol con el modelo sugerido en [NAI/84a] "Heterogenous SLD Resolution (HSLD)", sin embargo el presente sistema fue concebido independientemente. Aunque desde el punto de vista teórico ambas propuestas son equivalentes (únicamente en lo referente a la organización del árbol: en HSLD sólo se contempla la etapa de refinamiento), y por lo tanto, todos los resultados sobre consistencia y completitud se mantienen [LLO/83], la presente formulación conduce a una implementación mucho más eficiente que la permitida por HSLD, pues restringe la invocación de predicados y podría mantener las unificaciones realizadas en el "Máximo Subconjunto Unificable (MUS)" [CHE/86],[COX/84].

A continuación se muestra la conveniencia de la posibilidad de utilización de un componente "breadth-first".

Supongamos el siguiente mapa, representado con las cláusulas correspondientes, donde se desea hallar si es posible ir de A hasta K:




```
ir(a,b)<-.  ir(b,c)<-.  ir(b,e)<-.
ir(a,i)<-.  ir(e,f)<-.  ir(c,d)<-.
ir(e,g)<-.  ir(j,k)<-.  ir(i,j)<-.
ir(g,h)<-.

camino(X,X)<-.
camino(X,Y)<-ir(X,Z),camino(Z,Y).
     <-camino(A,K)
```

Se considera como información heurística la distancia aerea entre las ciudades, es decir, en cada momento se elige la ciudad que este más cerca del objetivo K:

```
dist-k(a,2)<-.  dist-k(b,1)<-.  dist-k(c,3)<-.  dist-k(d,6)<-.
dist-k(e,4)<-.  dist-k(f,2)<-.  dist-k(g,3)<-.  dist-k(h,6)<-.
dist-k(i,2)<-.  dist-k(j,2)<-.  dist-k(k,0)<-.
```

Si esta información se utiliza en forma local, es fácil verificar que las ciudades seran "visitadas" en el siguiente orden: A,B,C,D,E,F,G,H,I,J,K

En cambio, introduciendo una componente "breadth-first" en el uso de la heurística, es decir utilizándola en forma global, se obtiene un comportamiento más eficiente, pues el orden sería: A,B,I,J,K

Un ejemplo más drástico, donde utilizando una estrategia "depth-first" no es posible hallar las dos soluciónes para ningún ordenamiento (inclusive dinámico) de las cláusulas, es el siguiente:

```
<-C(X).
C(X)<-P(X).     C(X)<-Q(X).
P(0).           P(X)<-P(s(X)).
Q(1).           Q(X)<-Q(s(X)).
```

Pues una vez encontrada una solución, el sistema "depth-first" entra en una rama infinita.

A continuación se muestra cómo expresar los meta-predicados sugeridos principalmente por Dincbas [DIN/84] y Gallaire [GAL/80].

```
Selección de literales:
ACTIVATE(R,L) <- condición
                 [activar el literal subsumido por L en el
                 resolvente R]
FREZZE(R,L) <- condición
                 [postergar la utilización de L en R]
BEFORE(L1,L2) <- condición
                 [seleccionar el literal L1 antes del L2]

ORDENAR-literales(nil,nil) <->.
ACTIVATE(X,X1),borrar(X1,X,X'):ORDENAR-literales(X,X1.Y) <->
    ORDENAR-literales(X',Y).
FREEZE(X,X1),borrar(X1,X,X'):ORDENAR-literales(X,Y..(X1)) <->
    ORDENAR-literales(X',Y).
BEFORE(X1,X2):ORDENAR-literales(X1.X,X1.Y) <->
    ORDENAR-literales(X,Y).
BEFORE(X1,X2),borrar(X1,X,X'):ORDENAR-literales(X2.X,X1.X2.Y)<->
    ORDENAR-literales(X',Y).
else:ORDENAR-literales(X1.X,X1.Y) <->
    ORDENAR-literales(X,Y).

Selección de cláusulas:
CHOOSECLAUSE(L,B) <- condición
                 [dar prioridad a la cláusula con cuerpo B para el
                 literal L]
INHIBCLAUSE(L,B) <- condición
                 [evitar la cláusula con cuerpo B para el literal
                 L]
OPBEFORE(C1,C2) <- condición
                 [utilizar la cláusula C1 antes de la C2]

ORDENAR-cláusulas(nil,nil) <->.
CHOOSECLAUSE(L,B),BORRAR(L.B,X,X'):ORDENAR-cláusulas(X,(L.B).Y).
          ORDENAR-cláusulas(X',Y).
INHIBCLAUSE(L,B):ORDENAR-cláusulas((L.B).X,Y) <->
          ORDENAR-cláusulas(X,Y).
OPBEFORE(X1,X2):ORDENAR-cláusulas(X1.X,C1.Y) <->
          ORDENAR-cláusulas(X,Y).
OPBEFORE(X1,X2),borrar(X1,X,X'):ORDENAR-cláusulas(X2.X,X1.X2.Y).
          ORDENAR-cláusulas(X',Y).
else:ORDENAR-cláusulas(X1.X,X1.Y) <->
          ORDENAR-cláusulas(X,Y).
```

(Obviamente, para lograr que "else" sea elegido siempre último sólo hay que escribir: BEFORE(X,else:Y) <-)

```
Selección del punto de "Backtracking":
INHIBACK(L,R) <- condición
                [evitar la elección del resolvente R en caso de
                falla de L]
BACKFAIL(L,R) <- condición
                [utilizar el resolvente R en caso de falla de L]
REJECTGOAL(N,R) <- condición
                [no intentar probar el resolvente R de nivel N]

INHIBACK(L,X1),falla(L):ORDENAR-ramas(nil,X1.X,Z) <->
        ORDENAR-ramas(nil,X,Z).
BACKFAIL(L,X1),falla(L),borrar(X1,X,X'):ORDENAR-rama(nil,X,X1.Z)
        ORDENAR-ramas(nil,X',Y).
else:ORDENAR-ramas(nil,X,X) <->.
REJECTGOAL(N,X1),profundidad(X1,N):ORDENAR-ramas(X1.X,Y,Z) <->
        ORDENAR-ramas(X,Y,Z).

Implementando "Depth-First":
ORDENAR-ramas(X,Y,Z),DIF(X,nil) <->
        FILTRAR-ramas(X..Y,Z).
Implementando "Breadth-First":
ORDENAR-ramas(X,Y,Z),DIF(X,nil) <->
        FILTRAR-ramas(Y..X,Z).
Implementando "Heuristic Search":
ORDENAR-ramas(X,Y,Z),DIF(X,nil) <->
        ordenar-juntar(X,Y,Z'),FILTRAR-ramas(Z',Z).
```

El mecanismo de "backtracking inteligente" sugerido en [BRU/84], que consiste en volver directamente al "literal culpable" cuando se detecta una falla, también se puede realizar mediante este enfoque.

En el mismo, es necesario guardar información durante la ejecución "forward", con la consiguiente sobrecarga, e impidiendo muchas veces, por aspectos de implementación, la interacción con otras reglas de control. En el sistema propuesto, una regla que puede ser vista como "backtracking inteligente (BI)", consiste en, cuando ocurre una falla, depurar todos los resolventes que tengan literales subsumidos por el que falló y aquellos en los que las variables ya hayan tomado su valor (notar que en el árbol están representadas las substituciones en forma explícita y que sería posible agregar información semántica acerca de qué variables son las que habría que tener en consideración para cada predicado). La ejecución "forward" será menos costosa en tiempo y en espacio, no así la "backward", pero al estar el componente "forward" guiado mediante heurística, se necesitaría realizar "backtracking" en menor cantidad de oportunidades. En caso contrario, y si se implementa adecuadamente el árbol de búsqueda, la sobrecarga seguiría siendo lineal.
Ademas, esta regla puede interactuar con otras, en formas que el "BI" no puede, pues este sólo reacciona "inteligentemente" ante los errores cuando estos son descubiertos, en vez de evitarlos en primer lugar [NAI/84a].

"Ordenar una lista" o "el problema de las 8 reinas", son ejemplos de esta situación. Una vez generada una permutación, el "BI" regresa directamente al resolvente culpable de la falla, pero es necesario generar una nueva permutación antes que pueda ser detectada otra falla. Un comportamiento más eficiente estaría dado por la posibilidad de co-rutinar el "test" con la permutación, en forma conjunta con el BI. Es más, en el caso de "ordenar una lista", se podría instrumentar que en el momento de "backtracking" se tenga en cuenta el resolvente donde se instanció la primera variable, pues esta debe ser cambiada de lugar. Actitud que no se podría realizar con el "BI" tradicional.

Es posible también, implementar mecanismos más generales que "aprendan" de los errores, cambiando el orden de los tests, de las cláusulas, etc.

Los métodos de detección de ciclos sugeridos en [BRO/84] también son fácilmente implementables en forma similar al predicado "REJECTGOAL".

6.2. Ejecución:

Expresando explícitamente la fase de ejecución, se puede obtener resultados sumamente interesantes, pues es posible cambiar o ampliar las reglas de inferencia. La regla provista por un sistema típico sería:

a) RESOLVENTE(<- X1.X, X1 <- W, <- W..X) <->.

Pero supongamos que queremos agregar información sobre como actuar con predicados particulares, por ejemplo el predicado "igual": estaremos agregando a la teoría aspectos correspondientes a paramodulación [ROB/69]

b) RESOLVENTE(<- X1=Y1.X, Y2=X1 <- W , <- Y2=Y1.W..X) <->.

Supongamos que se desea demostrar la unicidad del elemento neutro:

<- e=e´ a partir de los siguientes axiomas:
0) X = X <-
1) X.e = X <-
2) e.X = X <-
3) X.e´= X <-
4) e´.X= X <-

utilizando estos axiomas junto con la definición de "RESOLVENTE" de la meta-teoría, quedarían los siguientes pasos:
tomando (3/b): <- e.e´=e´ ; tomando (2/b): <- e´=e´ ; y con (0/a): <-
De esta forma, no solo se obtiene una prueba más corta y más clara, sino que, además se evitan los cíclos debido a la conmutatividad de la suma.

Definir la conmutatividad en la meta-teoría, para cualquier predicado "OP" y de esta manera evitar ciclos, se haría de la siguiente manera:

```
c)  RESOLVENTE(<-X1 OP X2.X,X1 OP X2<-W,<-W..X) <->.
    RESOLVENTE(<-X1 OP X2.X,X2 OP X1<-W,<-W..X) <-> CONMUT(OP).
```

También se puede formalizar el método de resolución para cláusulas generales (con más de una conclusión). Para lograr una estrategia completa, cada vez que se produce un resolvente "merge" éste puede ser agregado a la base de datos.[NIL/80] (Suponer las cláusulas formadas por un conjunto de literales positivos y negativos).

```
d)    RESOLVENTE(  X1.X, Y, Z) <- CANCELAR(X1,Y,Y'),MERGE(X.Y',Z)
<->.
                CANCELAR(+X,-X.Y,Y) <->.
    +X <> Y1:CANCELAR(+X,Y1.Y,Z) <-> CANCELAR(+X,Y,Z).
                CANCELAR(-X,+X.Y,Y) <->.
    -X <> Y1:CANCELAR(-X,Y1.Y,Y1.Z) <-> CANCELAR(-X,Y,Z).
    remueve-repetidos(X,Y):MERGE(X,Y) <-> Agregar(Y).
                          else:MERGE(X,X) <->.
```

Se puede utilizar un parámetro para expresar la probabilidad y computarla mediante meta-cláusulas:

```
e)    RESOLVENTE(  <-  X1.X  / P1, X1 <- Y / P2, <- X..Y.(* P2 P1
P)/P ) <->.
```

Se puede expresar otras reglas de computación, como las sugeridas para "Distributed Logic" [MON/84]:

```
f)  RESOLVENTE(<-(X1)..Y1..(X2)..Y2,X1'X2<-Y,<-Y..Y1..Y2) <->.
```

Los ejemplos muestran como es posible incorporar teorías en el método de resolución, haciendo innecesario resolver directamente a partir de los axiomas de la misma. Esto reduce el largo de las demostraciones y el espacio de prueba, evitando muchas veces ciclos de difícil detección. Esta posibilidad permite considerar al sistema como una implementación del método "Theory Resolution" [STI/85].

6.3. Recuperación:

También se puede explicitar criterios para la fase de recuperación, que están fuertemente ligados con los de la fase de refinamiento:

```
RECUPERAR( X, S) <-> conjunto(clausulas(X <- Y),S).
RECUPERAR( X1=X2, S) <-> conjunto(clausulas(Y1=X1,S)).
```

Inclusive se podría implementar mecanismos de recuperación que utilizaran apareo (matching) aproximado.

También, mediante la implementación de un esquema que mantenga dos sistemas: uno para el objetivo y sus sucesores (sistema "top-down") y otro para los hechos y sus consecuencias (sistema "bottom-up"), es posible lograr estratégias flexibl

es de recuperación. De esta manera, en cambio de existir un único mecanismo predefinido ("goal-directed") el sistema podria cambiar su estructura de control dinámicamente a medida que el programa progresa, utilizando "backward" para algunas demostraciones y "forward" para otras. Por ejemplo:

la recuperación bi-direccional podría expresarse mediante:

use reglas que mencionen el objetivo actual o la conclusión actual

o mediante el criterio sugerido por [POH/72]:

en cada paso elegir la dirección de inferencia que da lugar al menor número de alternativas

la invocación que determina pruebas de "a un paso" mediante:

use reglas que mencionen el objetivo actual y la conclusión actual

Una forma de acelerar algunas deducciones podría consistir en "deducirlas" en el sistema "top-down" y "memorizarlas" en el "bottom-up" para su posterior uso.

7. Limitaciones:

El presente trabajo no abarca la definición de un meta-lenguaje que provea un conjunto de conceptos predefinidos para expresar más facilmente las distintas estrategias. Considero que para el desarrollo del mismo habría que tener en cuenta la clasificación propuesta en [NAI/84b].

Otro aspecto no contemplado concierne al uso de notación funcional y relacional. De otro modo, la siguiente invocación "f(g(x),h(r(y)))" debe escribirse "g(X,U),r(Y,V),h(U,W), f(U,W,Z)", siendo necesario el uso de variables intermedias, e imponiendo nuevos problemas de control. Una posibilidad de solución consiste en "compilar", a partir de la expresión funcional, la forma relacional con el correspondiente control asociado.

Hay que notar que el uso de cláusulas de control no siempre reducirá el tiempo de ejecución, pues hay que agregar el ciclo de interpretación de las cláusulas de control. Sin embargo el crecimiento es lineal y la mejora puede ser exponencial!!. Al respecto, un tema que requiere futura investigación consiste en la posibilidad de utilizar el conocimiento acerca del control para la transformación automática del programa, permitiendo un control más elemental durante la ejecución.

8. Conclusiones:

Se presentó un formalismo que permite expresar el componente de control de un programa lógico en forma declarativa. El mismo tiene un interesante grado de uniformidad, al tratar al problema de control desde el punto de vista "meta", permitiendo la codificación de distintas estrategias específicas que pueden interactuar eficientemente. A diferencia de otras propuestas, ésta posibilita la codificación de estrategias completas, gracias a la introducción de un componente "breadth-first" en la búsqueda.

La hipótesis es que la ineficiencia no depende de una estrategia de deducción particular, sino que es intrínseca al requerimiento de que toda la lógica del programa sea expresada de una manera uniforme y no diferenciada. Esto lleva a explicitar: a)la selección de reglas, b)las reglas excluyentes, c)las acciones secuenciales, y d)la separación de las distintas etapas del proceso de invocación. Estas características brindan al sistema una gran flexibilidad permitiendo expresar fácilmente: a)aspectos de control (refinamiento), b)distintas reglas de inferencia (ejecución), y c)distintas estrategias de búsqueda (recuperación).

Bibliografía:

[AIE/86] L.Aiello,C.Cecchi,D.Sartini,"Representation and Use of Metaknowledge", Proceedings of the IEEE, Vol.74,No 10, 1986

[BOW/82] K.Bowen,R.A.Kowalski,"Amalgamating Language and Metalanguage in Logic Programming", Logic Programming, Clark and Tarnlund(eds), Academic Press, 1982.

[BRO/84] D.R.Brough and A.Walker, "Some Practical Properties of Lógic Programming Interpreters", Proceedings of the 1984 Conference on Fifth Generation Computer Systems, Tokyo.

[BRU/84] M.Bruynooghe, L.M.Pereira, "Deduction Revision by Intelligent Backtracking", Implementations of Prolog, J.A.Campbell (ed), Ellis Horwood, 1984.

[BUN/81] A.Bundy,L.S.Sterling, "Meta-level Inference in Algebra", Workshop on logic programing for intelligent systems, Los Angeles,1981.

[CHE/86] T.Y.Chen,J.L.Lassez,G.S.Port, "Maximal Unifiable Subsets and Minimal Non-Unifiable Subsets", New Generation Computing, 4/86,133-152.

[CLA/80] K.L.Clark and F.G.McCabe, "IC-Prolog - Language Features", in Logic Programming, Clark and Tarnlund(eds), Academic Press, 1982.

[CLO/84] W.F.Clocksin and C.S.Mellish, "Programming in Prolog", Springer Verlag, 1984.

[COL/82] A.Colmerauer,"Prolog-II Manuel de Reference", GIA, Universite d'Aix-Marseille II, 1982.

[COX/84] P.T.Cox, "Finding Backtrack Points for Intelligent Backtracking", Implementations of Prolog, J.A.Campbell (ed), Ellis Horwood, 1984.

[DAV/77] R.Davis, "Generalized Procedure Calling Content Directed Invocation", Proc.Symposium on AI and Programming Languages,Rochester,1977.

[DAV/80] R.Davis, "Meta-Rules: Reasoning about Control", MIT AI Memo 576, 1980.

[DIN/84] M.Dincbas, "Metacontrol of Logic Programs in METALOG", Proceedings of the 1984 Conference on Fifth Generation Computer Systems, Tokyo.

[GAL/80] H.Gallaire and C.Lasserre, "A control Metalanguage for Logic Programming", Logic Programming, Clark and Tarnlund(eds), Academic Press, 1982.

[KOW/79] R.A.Kowalski, "Algorithm=Logic+Control", Communications of the ACM 22, 7/79, 424-436.

[KOW/81] R.A.Kowalski, "Logic for Problem-Solving", North-Holland, 1981.

[LLO/83] J.W.Lloyd, "Foundations of Logic Programming", Springer, 1984.
[MON/84] L.Monteiro, "A Proposal for Distributed Programming in Logic", Implementations of Prolog, J.A.Campbell (ed), Ellis Horwood, 1984.
[PIE/82] T.Pietrzykowski and S.Matwin,"Exponential Improvement of Efficient Backtracking", Proc.6th Conference on Automated Deduction,1982.
[NAI/83] L.Naish, "Automatic Generation of Control for Logic Programs", Negation and Control in Prolog, Springer, 1985.
[NAI/84a] L.Naish, "Heterogeneous SLD Resolution", Negation and Control in Prolog, Springer Verlag, 1985.
[NAI/84b] L.Naish, "PROLOG Control Rules", Negation and Control in Prolog, Springer Verlag, 1985.
[NAI/84c] L.Naish, "MU-PROLOG 3.2db Reference Manual", Negation and Control in Prolog, Springer Verlag, 1985.
[NIL/80] N.Nilsson, "Principles of Artificial Intelligence", Tioga, 1980.
[PEA/84] J.Pearl, "Heuristics, Intelligent Search Strategies for Computer Problem Solving", Addison Wesley, 1984.
[PER/84] L.M.Pereira, "Logic Control With Logic", Implementations of Prolog, J.A.Campbell (ed), Ellis Horwood, 1984.
[POH/72] I.Pohl, "Bi-directional search", Machine Intelligence 7, Edinburgh Uviversity Press, New York.
[POR/84] A.Porto, "EPILOG:A Language for Extended Programming in Logic", Implementations of Prolog, J.A.Campbell (ed), Ellis Horwood, 1984.
[ROB/69] G.Robinson,L.Wos,:Paramodulation and Theorem-proving in First Order Theories with equality, Machine Intelligence 4, 103-33,1969.
[STE/86] Sterling L,E.Shapiro,"The Art of Prolog",MIT Press,1986.
[STI/85] M.E.Stickel,"Automated Deduction by Theory Resolution", Journal of Automated Reasoning 1, 1985.