BACKTRACKING INTELIGENTE PARA PROLOG

SILVIA TAKAHASHI

UNIVERSIDAD DE LOS ANDES
Apartado Aereo 4976

RESUMEN

En la ejecución normal de un programa Prolog, cuando falla un objetivo se reactiva el objetivo anterior. Esto puede ocasionar una pérdida de tiempo si el objetivo anterior no ayuda a solucionar la falla. En este caso, el interpretador terminará por saltar más atrás al terminar de explorar todas las posibilidades del objetivo al cual se devolvió. Si solamente se devuelve a objetivos que tengan posibilidad de resolver la falla, se debe producir un ahorro considerable de tiempo.

Se podría pensar entonces, que un interpretador de Prolog que tuviera esta facilidad sería muy superior a uno sin ella, pero pesando los problemas que trae la implementación con las ventajas que produce, esto no es muy claro. El problema radica en que el programador puede escribir sus programas de manera que no se vea la diferencia entre uno y otro. Todo depende de la eficiencia de los programas.

0. Introducción.

Este artículo explica el backtracking inteligente aplicado a la interpretación de PROLOG [ROU73]. Además analiza la dificultad de la implementación de esta forma de hacer bactrack sobre una máquina intermedia Prolog con estructura de pilas. Finalmente analiza las ventajas y desventajas de agregar esta mejora a los interpretadores de Prolog.

M. Bruynooghe [BRU78], [BRU80] propuso el backtracking inteligente como una forma de hacer bactrack en programas lógicos cortando las recontestaciones inútiles.

Este trabajo está basado en la tesis de grado de autora [TAK87] en la que se describe la implementación del bactracking inteligente sobre la máquina intermedia Prolog descrita por D. Warren [WAR77] e implementada por R. Arias en su tesis [ARI85].

El lector debe estar familiarizado con conceptos básicos de Prolog tales como unificación, bactrack y corte. Además debe conocer las principales características de la máquina descrita por Warren.

1. Definición de Términos.

Un programa Prolog se compone de una base de datos y de un objetivo inicial. La base de datos es una sucesión de cláusulas de Horn de dos tipos, hechos y reglas. Las reglas se componen de una cabeza y un cuerpo separados por el símbolo ":-" y terminan con un punto: cabeza:-cuerpo. . Los hechos son reglas que no tienen cuerpo: cuerpo. . El objetivo inicial puede verse como una cláusula que no tiene cabeza; es un cuerpo precedido por el símbolo "?-" : ?-cuerpo. .

Las cabezas son fórmulas atómicas y los cuerpos son sucesiones de fórmulas atómicas separadas por comas, que en lo sucesivo llamaremos subobjetivos o llamadas.

Como es bien conocido, el lenguaje Prolog tiene dos semánticas [BRU78]: la declarativa y la procedimental. Utilizándolas se puede analizar la corrección de los programas escritos en Prolog. La semántica declarativa se basa en el significado lógico de las cláusulas y la procedimental en el control de ejecución.

La semántica que se necesita para poder describir un algoritmo de interpretación es la procedimental. Utilizando esta semántica las cláusulas que componen la base de datos son definiciones de procedimientos y los subobjetivos que conforman el cuerpo de las cláusulas y el objetivo inicial son llamadas a estos procedimientos. El objetivo inicial es el programa principal.

El conjunto de cláusulas cuyas cabezas tienen el mismo nombre constituye la definición del procedimiento con ese nombre. A las cabezas de estas cláusulas se les da el nombre de puntos de entrada del procedimiento. El hecho de que haya más de un punto de entrada para algunos procedimientos da la posibilidad de hacer backtrack para encontrar la solución de un objetivo inicial.

Los subobjetivos se van resolviendo en orden secuencial de izquiera a derecha y las cláusulas para resolverlos se van escogiendo en orden descendente.

A grandes rasgos, la ejecución de un programa Prolog se hace en la forma descrita a continuación.

Supóngase el siguiente objetivo inicial:

?-a1, ... , ar. (r $\geq$ 0).

En este momento este es el objetivo actual.

Se toma la primera llamada del objetivo actual a1 y se busca el primer punto de entrada para esa llamada, la cláusula b :- c1, ... ,cm. (m $\leq$ 0), que cumpla con la condición de que los argumentos de b y de a1 unifiquen con un unificador más general θ. Esto da como resultado otro objetivo:

?- (c1, ... ,cm,a2, ..., ap.) θ

A este objetivo lo llamamos objetivo derivado y al proceso aplicado lo denominamos proceso de avance.

Ahora el objetivo derivado es el actual.

El proceso de avance se aplica repetidamente sobre los objetivos derivados. Este ciclo se detiene cuando se llega a un objetivo derivado vacío o cuando no se encuentra una cláusula que unifique con el primer subobjetivo del objetivo derivado.

Si se da el primer caso, se dice que el programa termina exitosamente. Se contesta sí y como resultado se dan los valores que deben tomar las variables del objetivo inicial para que éste se cumpla. Estos valores se obtienen haciendo la composición de todas las substituciones aplicadas.

Si se da el segundo caso, se debe hacer backtrack . Para explicar este concepto necesitamos hacer algunas convenciones. Al objetivo que, al aplicarle el proceso de avance, dio como objetivo derivado el objetivo actual, lo llamamos punto de backtrack y a la cláusula aplicada en este proceso de avance la llamamos cláusula errónea . El proceso de hacer backtrack consiste, entonces, en devolverse al punto de backtrack y tratar de volver a aplicar el proceso de avance.

Vemos que la ejecución de un programa Prolog describe una lista de objetivos que se obtienen aplicando repetidamente el proceso de avance. Hacer back - track se reduce a devolverse al objetivo anterior.

Cuando decimos resolver el objetivo i, recontestar el objetivo i o falló el objetivo i nos estamos refiriendo a la primera llamada del i-ésimo objetivo.

Si se da el caso de que el objetivo inmediatamente anterior no se pueda recontestar porque la cláusula que se utilizó para resolverlo era el último punto de entrada de la definición de ese procedimiento, decimos que este objetivo es un objetivo determinístico [PER82]. Una definición más formal de este concepto es la siguiente: "Un objetivo es determinístico si su primera llamada después de resuelta no tiene posibilidad de ser recontestada." Para ahorrar tiempo, al fallar un objetivo debe devolverse, en lugar de al objetivo inmediatamente anterior, al objetivo no determinístico más reciente. Este es ahora su punto de back track.

Otra definición que también es necesaria es la del concepto de padre de un objetivo. El i-ésimo objetivo es padre del j-ésimo objetivo si al pasar del objetivo i al i+1 se generó la primera llamada del objetivo j. Es decir, esta llamada formaba parte del cuerpo de la cláusula que se utilizó para resolver el objetivo i.

Cuando hay un corte, el problema se reduce a marcar como determinísticos los objetivos desde el actual hasta su padre. Es decir, si el corte es la primera llamada en el objetivo i y el padre de éste es j entonces todos los objetivos desde i hasta j son marcados como determinísticos. Si falla algún objetivo después del corte el interpretador se comporta como si fallase la cláusula que lo activó.

2. Arbol de ejecución.

Para facilitar la explicación de algunos conceptos, en este documento se describe la ejecución de un programa Prolog por medio de un árbol que se va creando durante la ejecución.

Un nodo es un subobjetivo y sus hijos son los subobjetivos de la cláusula utilizada para resolverlo. Si esta cláusula es un hecho, el hijo del subobjetivo es un nodo que llamamos hoja vacía . La raiz de este árbol es un subobjetivo ficticio que tiene como hijos a los subobjetivos del objeto inicial.

Vemos entonces, que tenemos tres tipos de nodos: la raiz, los subobjetivos o llamadas y las hojas vacías.

A cada nodo de tipo llamada le corresponde lo siguiente:
- El conjunto de las variables que aparecen en la llamada.
- El conjunto de las cláusulas que conforman la definición del procedimiento y que no han sido desechadas. En un principio incluye a todas las cláusulas en la definición.
- La cláusula que fue aplicada para resolver el subobjetivo. Si todavía no se ha resuelto no está definida.
- El conjunto de las cláusulas que han sido desechadas ya sea porque no unificaron o porque se hizo backtrack al objetivo en el cual ese subobjetivo era la primera llamada y se escogieron para resolverlo.

A estos conjuntos los llamamos VARS , DISP , APLIC y ELIM respectivamente.

Vemos que en cualquier momento de la ejecución y para cualquier llamada a se cumple que:

$$ELIM \cap \{APLIC\} \cap DISP = \emptyset$$
$$ELIM \cup \{APLIC\} \cup DISP = \text{Conjunto de las cláusulas que componen la definición del procedimiento para } a .$$

Los nodos terminales de tipo llamada representan las llamadas que todavía no se han resuelto. En un momento dado de la ejecución, el objetivo que se está tratando de resolver está formado por estos nodos terminales. A partir de esto podemos tener la lista de objetivos que se van derivando durante ejecución.

Si el objetivo actual es i debemos conocer además de i a todos los objetivos menores a i. Además para cada objetivo debemos conocer su padre, su punto de backtrack y, si ya se resolvió, las substituciones hechas para pasar al siguiente objetivo.

3. Motivación.

La idea fundamental de hacer backtracking inteligentemente en Prolog consiste en evitar que al fallar un objetivo i en el interpretador se devuelva a un objetivo j cuya recontestación no ayude a cambiar el hecho de que i falle. Además al devolverse no deben deshacerse llamadas cuyas ejecuciones no se vean afectadas por la recontestación de ese objetivo.

Se ve la necesidad de usar backtracking inteligente en el siguiente ejemplo.

Dado el siguiente programa Prolog:

```
1. g1(X1) :- g(X1,X1).
2. g(X2,Y2) :- f(X2,Z2),h(Y2,W2).
```

```
3. g(a,Y3) :- l(Y3).
4. l(a).
5. f(a,b).
6. f(a,c).
7. h(n,m).

?-g1(a)
```

La lista de objetivos que se van derivando es:
1. g1(a). unifica con la cláusula 1.
2. g(a,a). unifica con la cláusula 2.
3. f(a,Z2),h(a,W2). unifica con la cláusula 5.
4. h(a,W2). falla,el último objetivo no determinístico es el 3.
3. f(a,Z2),h(a,W2). unifica con la cláusula 6.
4. h(a,W2). falla, el último objetivo no determinístico es el 2.
2. g(a,a). unifica con la cláusula 3.
3. l(a). unifica con la cláusula 4.
4. EXITO

Vemos que el devolverse al objetivo 3 al fallar el objetivo 4, no cambió el hecho de que éste fallara. Si se hiciese backtracking inteligentemente se ahorraría este paso inútil porque se daría cuenta de que el objetivo 3 no tiene posibilidad de solucionar la falla y se devolvería directamente al objetivo 2.

La recontestación de algunos objetivos no sirve. Si se devuelve a un objetivo que no ayuda a solucionar la falla, en algún momento se acaban las posibilidades para recontestarlo y termina devolviéndose más arriba. Haciendo backtracking inteligentemente se ahorran estos pasos inútiles.

Si al tratar de resolver un objetivo i, no tenemos éxito, debemos devolvernos solamente a objetivos de los cuales dependa i. Es decir, a objetivos que tengan alguna posibilidad de remediar la causa de la falla.

4. Dependencia. [BRU78] , [BRU80]

La primera llamada de un objetivo i existe y algunas de las variables en sus argumentos tienen ciertos valores debido a la aplicación del proceso de avance a ciertos objetivos para resolverlos. Diremos que el objetivo i depende de estos objetivos y al conjunto de éstos lo llamamos D-Set. Denotamos el conjunto de dependencias del i-ésimo objetivo D-Set(i).

Si un objetivo falla sólo debemos hacer backtrack a objetivos dentro de su D-Set. Escogemos uno de ellos, digamos el más reciente, y lo reactivamos.

La relación de dependencia es una relación transitiva. Sólo es necesario guardar las dependencias directas en el D-Set. Vemos entonces que i depende de j si j pertenece al D-Set(i) o existe k en D-Set(i) tal que k depende de j .

La definición del conjunto de dependencias debe darse operacionalmente. Este conjunto se va creando durante ejecución y es más natural este tipo de definición.

Supóngase que estamos tratando de resolver el objetivo i. Inicialmente su conjunto dependencias incluye únicamente a su padre. Obviamente un objetivo depende de su padre. El conjunto se va aumentando durante la ejecución del algo-

ritmo de unificación. Supóngase que estamos unificando uno de los argumentos de la primera llamada de i con un argumento de la cabeza de la cláusula con la cual se está tratando de resolver. Si es necesario acceder a substituciones hechas al resolver un objetivo j distinto de i y del cual i no depende, para encontrar el valor de uno de sus argumentos, debemos agregar j al conjunto de dependencias de i. Cada vez que se agregue un elemento deben eliminarse los elementos que son redundantes por transitividad. Otra simplificación de los conjuntos de dependencia se obtiene no guardando en ellos objetivos determinísticos. Si en un momento debemos agregar a D-Set(i) un elemento j y j es un objetivo determinístico no lo agregamos a D-Set(i); en lugar de hacer esto unimos D-Set(i) con D-Set(j) y a esta unión le quitamos las redundancias. Esto se justifica para no perder soluciones.

Otra clase de dependencia debe ser tenida en cuenta. Supóngase que estamos en un objetivo i y que éste falla. Hacemos backtrack a un objetivo j que esté en D-Set(i) y lo reactivamos. Estamos tratando de solucionar este objetivo debido a la falla de i, entonces D-Set(j) debe modificarse. Esta modificación es la siguiente:

$$D\text{-}Set(j) := D\text{-}Set(j) \cup D\text{-}Set(i) - \{j\}$$

Después las redundancias deben ser eliminadas.

5. Más definiciones.

Basándonos en el árbol de ejecución descrito en el parágrafo 2, daremos otras convenciones y definiciones necesarias para describir el algoritmo de interpretación con backtracking inteligente. Llamamos Conjunto de Entrada [BRU80] de un objetivo i y de una cláusula (digamos la j-ésima en la base de datos), con la cual se resolvió o trató de resolver el objetivo, al conjunto de objetivos que pueden ser responsables de una eventual falla. Es como un conjunto de dependencias para la unificación. Denotaremos este conjunto de entrada como Input(i,j). Utilizando el árbol de ejecución como referencia, vemos que debe haber un conjunto de entrada para cada cláusula APLIC y para las cláusulas en ELIM de cada subobjetivo. Para cada objetivo, su conjunto de dependencias es la unión de los conjuntos de entrada de la cláusula APLIC y de las cláusulas en ELIM de su primer subobjetivo. Además su D-Set debe incluir a su padre. Estos conjuntos pueden simplificarse para no guardar dependencias indirectas ni objetivos determinísticos.

Una definición más exacta del conjunto de entrada de un objetivo i y de una cláusula j se obtiene definiendo para las variables otro conjunto, que llamaremos LIG [PER82]. Este conjunto se define así: pertenecen al conjunto LIG los objetivos que dieron valor a esa variable al resolverlos. Por ejemplo, si una variable X está ligada al valor a(I,O) e I está ligada a q y O a s, LIG de X debe contener los objetivos que al resolverse ocasionaron que X, I y O se ligaran a los valores que tienen ahora. Además de esto, si al resolver un objetivo j se ligaron dos variables que pertenecían a un mismo subobjetivo, j debe estar en LIG de ambas variables. El último caso se da cuando en la cabeza de una cláusula aparece una variable más de una vez y unifica con dos variables libres diferentes.

El conjunto LIG se va modificando durante la ejecución del algoritmo de unificación. Si durante esta ejecución se le da valor a una variable, se debe agregar a LIG el objetivo que se está tratando de resolver. Si se liga una variable con valor a una libre entonces se debe copiar el conjunto LIG de la ligada en el de la libre. Si se ligan dos variables que están en la misma llamada y ambas están libres, se debe agregar al LIG de ambas el objetivo que se está tratando de

resolver.

El conjunto de entrada de un objetivo con una cláusula también se va creando durante la ejecución del algoritmo de unificación. Cuando se esté unificando una variable con un valor, se une el conjunto LIG de esa variable con el conjunto de entrada y cuando se unifican dos variables y ninguna está libre se agrega el conjunto LIG de cada una al conjunto de entrada.

6. Un algoritmo de unificación. [TAK87]

Lo más importante para un algoritmo de interpretación de Prolog que hace backtracking inteligentemente es el algoritmo de unificación.

Supongamos que:
1) El objetivo actual es el i-ésimo.
2) Estamos tratando de unificar su primera llamada, o1 con la cabeza de la k-ésima cláusula en la base de datos.
3) Tenemos el conjunto de dependencias de i, D-Set(i).
4) Los argumentos de la llamada son X1, ..., Xn.
5) Los argumentos de la cabeza de la cláusula con la que se está tratando de unificar son Y1, ... , Yn.

Después de la ejecución del algoritmo de unificación debemos analizar dos casos. Si tuvo éxito la unificación se deben hacer efectivas las substituciones generadas por la unificación. Si falló la unificación, las substituciones que se hayan alcanzado a hacer deben ser desechas. En ambos casos, se debe crear el conjunto de entrada para el objetivo i y la cláusula k y aumentar el conjunto de dependencias de i con este conjunto de entrada.

Este algoritmo de unificación se diferencia de uno normal, únicamente en que en él se deben tener en cuenta las modificaciones de los conjuntos LIG e INPUT descritas en la sección anterior.

7. La interpretación.

Habiendo dado las convenciones y descrito el algoritmo de unificación sólo resta explicar el ciclo de ejecución para el interpretador.

No hay cambio en el proceso de avance; solamente se presentan problemas cuando hay que devolverse al fallar.

El interpretador descrito en el parágrafo 1 escoge como punto de backtrack al último objetivo no determinístico. Este interpretador debe escoger, para devolverse, un objetivo en su conjunto de dependencias.

En [BRU78] se prueba que cualquier objetivo en el conjunto de dependencias puede ser escogido como punto de backtrack sin causar pérdida de soluciones. Esta observación es válida para programas lógicos en los cuales no se especifica el orden en que se solucionan las llamadas ni el orden en que se escogen las cláusulas. En el caso de Prolog, se debe devolver al objetivo más reciente, ya que de otra forma la semántica procedimental se vería afectada. Es claro que el devolverse al objetivo más reciente no cambia en nada la ejecución. Supóngase que falle el i-ésimo objetivo y que el (i-3)-ésimo es el objetivo más reciente en D-Set(i). No recontestar los objetivos i-1 e i-2 no cambia el ciclo de ejecución, ya que el recontestarlos no arregla la falla. Cuando haya pasado por todas

las recontestaciones de i-1 , trata de recontestar i-2 y para cada una de sus recontestaciones vuelve a explorar las recontestaciones de i-1. Como ninguna de éstas arregla la falla, finalmente se devuelve al objetivo i-3. Devolverse directamente ahorra tiempo sin alterar el resultado.

En [ERU80] se afirma que al fallar un objetivo i y devolverse a tro objetivo j , no se deben deshacer las ejecuciones de todos los objetivos desde i hasta j .

Esta modificación hace que sea necesario tener una lista de objetivos explícitamente. Digamos que al tratar de resolver el i-ésimo objetivo fallamos y que escogimos el j-ésimo como punto de backtrack. Como no se deshacen todas las ejecuciones, el objetivo que se reactiva no es el mismo j-ésimo. Para simular esto no nos devolvemos; creamos un objetivo siguiente (i+1) el cual es una copia del j-ésimo quitando las llamadas cuya ejecución debe mantenerse. Cada una de estas llamadas fue resuelta en un objetivo k ($j < k < i$). Si este objetivo no depende de j su ejecución debe mantenerse. Aún si no se mantiene su ejecución podemos no sacar todas las llamadas del conjunto ELIM, de su primer subobjetivo, para pasarlas al conjunto DISP del mismo. De pronto la recontestación no cambia el hecho de que estas cláusulas no unifiquen. Para saber si debemos sacarlas de ELIM y pasarlas a DISP debemos conocer sus conjuntos de entrada. Debemos sacar de ELIM aquellas cláusulas en cuyos conjuntos de entrada esté j (directamente o por transitividad). Para mantener la semántica procedimental, si sacamos la k-ésima cláusula, debemos sacar también todas las cláusulas que estén definidas antes de k en la base de datos del conjunto de entrada. Esta parte del backtracking inteligente consiste en cambiar el orden de ejecución cuando no se ve afectada la solución final para aumentar así la eficiencia.

8. La implementación. [TAK87]

La última parte explicada del backtracking inteligente no fue implementada, pues habría sido necesario cambiar totalmente la filosofía de la máquina intermedia Prolog (MIP). En la implementación no es necesario hacer copias de los objetivos al hacer backtrack, simplemente se devuelve. Tampoco necesitamos estructuras de datos para los conjuntos de entrada. No son necesarios, pues la parte no implementada del backtracking inteligente es la que, basándose en este conjunto, decide cómo modificar los conjuntos ELIM de los subobjetivos del punto de backtrack. Este es el único momento en que se usan. De resto siempre se utiliza el conjunto de dependencias que se obtiene de la unión de todos los conjuntos de entrada con el padre del objetivo. Entonces , en el algoritmo de unificación modificamos directamente el conjunto de dependencias.

La dificultad principal radicó en el diseño de las estructuras de datos. Es claro que es mucho más natural una estructura de árbol que una de pilas para la implementación del backtracking inteligente. Fue necesario definir una nueva estructura como una lista de objetivos,para que el algoritmo fuera más sencillo y eficiente. Sin embargo así se aumentó considerablemente la cantidad de memoria utilizada. La implementación de los conjuntos LIG influyó también en el aumento de la cantidad de espacios necesarios para cada variable en la pila.

9. Ventajas y Desventajas. TAK87

La implementación del backtracking inteligente sobre un interpretador cualquiera de Prolog presupone un aumento en la cantidad de memoria utilizada.

Veamos que debe tener un interpretador de estos en contraposición con uno

normal. Se deben tener los conjuntos de dependencias para cada objetivo, los conjuntos de entrada para cada llamada en ELIM y el conjunto LIG para cada variable. El costo en espacio es grande; habría que ver que tanto se gana en tiempo. Utilizando este método se ahorran muchos pasos inútiles. Si la memoria no es un problema, sí vale la pena utilizarlo. En caso de tener limitaciones de memoria, no se ve muy claro si se justifica su uso.

En muchos de los casos en los que utilizando el backtracking inteligente, se presenta un comportamiento distinto al obtenido sin este método, esto se puede evitar cambiando el orden de los subobjetivos en las definiciones de las cláusulas. Al fin y al cabo el backtracking inteligente simula el hecho de que los subobjetivos estén en distinto orden. Los cortes también pueden ser utilizados si se desea que un objetivo no se devuelva al anterior. Otra forma de programar eficientemente se logra definiendo procedimientos auxiliares.

Si se programa cuidadosamente es posible que nunca se vean las ventajas del backtracking inteligente. En este caso, se habría pagado un costo alto en espacio sin recibir retribuciones en tiempo. Para decidir si vale la pena usarlo es necesario ver que tan frecuentemente se ve una diferencia en el comportamiento de los dos métodos. Todo depende del programador al cual está dirigido. Una ventaja obvia es que el programador no tiene que preocuparse porque el orden le afecte la eficiencia de sus programas ya que el interpretador hace este trabajo por él.

10. Conclusiones.

El backtracking inteligente usado para la interpretación de Prolog fue modificado para no entrar en conflicto con la semántica procedimental.

La implementación de esta mejora sobre una máquina intermedia con estructura de pilas fue un proceso complicado debido a que la descripción del algoritmo dada en [BRU78] y [BRU80] se amolda más a una estructura de árbol.

Por último es bueno decir que el backtracking inteligente, sólo se justifica en un número reducido de programas, ya que si se programa eficientemente, el comportamiento de un interpretador con backtracking inteligente y de uno sin él no se diferencian. Si la memoria es un recurso barato, valdría la pena implementarlo para liberar al programador de la preocupación por la eficiencia.

11. Referencias.

[ARI85] Arias R., Compilación de cláusulas Prolog. Tesis de Grado. U. de los Andes. 1985.

[BRU78] Bruynooghe M., "Intelligent Backtracking for an Interpreter of Horn Clause Logic Programs", Mathematical Logic in Computer Science, Proceedings of the Colloquium Salgotarjan, Hungary, Sept. 1978.

[BRU80] _____, "Analysis of Dependencies to Improve the Behavior of Horn Clause Logic Programs", Afdeling Toegepaste Wiskunde en Programmatie Katholieke Universiteit Leuven. 1980.

[PER82] Pereira, L.M. 7 Porto, A., "Selective Backtracking", Logic Programming, Academic Press, 1982.

[ROU73] Roussell, PROLOG, Manuel de Reference et d'utilisation. Groupe Intelligence Artificielle, Université Marseille, 1975.

[TAK87] Takahashi, S., Backtracking Inteligente para un Interpretador de Prolog. Tesis de Grado. Universidad de los Andes. 1987.

[WAR77] Warren D. et al., "Prolog: The Language and its Implementation compared with LISP", Proceedings of the ACM Symposium on Artificial Intelligence and Programming Languages, ACM 1977.