Otro intérprete más de PROLOG no secuencial

Héctor Jiménez
Fac. de Ciencias Físico Matemáticas, UAP

23 de octubre de 1995

1 Introducción

La regla de inferencia de resolución [1] ha proporcionado una de las mejores pautas en la demostración automática de teoremas. Rápidamente, al difundirse este resultado, se formuló la idea de usar la lógica matemática como un lenguaje de programación [2]. Esta idea representa precisamente la programación lógica. El primer intérprete concebido fue PROLOG (1975), donde la resolución se adapta con una serie de restricciones sobre las fórmulas que son premisas (programa) y el teorema a demostrar (meta). Tales restricciones llevan al concepto de clausulas definidas y una manera particular de seleccionar tanto la parte de la fórmula a demostrar como la premisa o derivación empleada (resolución lineal), además la utilización de la negación como falla, para extender la expresividad en los programas y metas.

En el presente reporte me ocupo de presentar de manera introductoria un análisis sobre PROLOG puro, enseguida los cambios necesarios para convertirlo en un intérprete de una versión parecida a Prolog Concurrente. Estas especificaciones pretenden no sólo profundizar en el conocimiento de PROLOG, sino además ofrecer la posibilidad de experimentar con extensiones de la programación lógica. A lo largo de esta exposición emplearé la sintaxis estandar de microProlog [3].

2 Brevísimo recordatorio de PROLOG

Como en cualquier otro lenguaje de programación, PROLOG utiliza constantes de los tipos básicos conocidos (enteros, reales y cadenas). Además su constructor de estructuras más complejas está basado en listas. Así, `938`, `15.26` y `(ab 77 list 11.4)` son constantes. El caso de las *variables* no es como en los lenguajes de programación imperativos. Las variables serán denotadas con letras finales del alfabeto minúsculas o mayúsculas (`x, y, z` , más adelante se usarán `R S T U V W`). éstas cumplen con la propiedad de que sólo una vez pueden ser asignadas. Hay que aclarar que la palabra asignación sólo se ha usado aquí para expresar la idea, mas no existe tal concepto en PROLOG, en cambio, se tiene la *unificación* que se puede ver como un *empate* entre expresiones la cual obtiene el conjunto vacío si éstas son idénticas o bien aquellos cambios (sustituciones) que deberán efectuarse para que lo sean. Por ejemplo, la sustitución que se obtiene de empatar las listas `(x y)` y `(a x)` es `{x/a, y/a}`. Emplearemos indistintamente empate y unificación.

Un *átomo* expresa una propiedad o estado de un objeto y/o sus partes u otros objetos. Tal es expresado con un nombre y la información concerniente a los objetos.

```
(da abad arroz zorra)
```

por ejemplo, podría estar expresando que “el abad da arroz a la zorra”. También vemos que este átomo representa una relación terciaria derivada del verbo “dar”.

Se ha visto que una de las combinaciones más adecuadas de átomos para obtener fórmulas con mayor expresividad es la *cláusula definida*. La cláusula definida se compone de una *cabeza* y un *cuerpo*. La cabeza es un átomo y el cuerpo es una conjunción lógica de átomos. Normalmente su notación es la siguiente:

$$C \leftarrow C_1, C_2, \ldots, C_n$$

Y su significado es la implicación del cuerpo a la cabeza, es decir, C se cumple si C_1 y $C_2, \ldots$ y C_n se cumplen. Nosotros denotaremos a la clásula de la siguiente manera:

```
(C C_1 C_2 ... C_n)
```

Un ejemplo de cláusula es:

```
((ancestro x z) (padre x y) (ancestro y z))
```

Puede ser que una cláusula no tenga cuerpo ($n = 0$), a éstas se les conoce como *hechos*. Por ejemplo:

```
((padre pancho oscar))
```

Los programas de PROLOG son conjuntos de cláusulas, y pueden verse organizadas en *procedimientos* (aquellas cláusulas con el mismo nombre) cuyo orden definirá cuál es elegida primero.

Lo anterior no está completo sin la definición de lo que activa un programa PROLOG, que es precisamente una *meta*. Una meta es una conjunción de átomos y expresa una pregunta, la cual puede responderse afirmativa o negativamente. Lógicamente una meta se *cumple* si existe, para cada uno de sus componentes, un empate con algún hecho o bien, recursivamente, empate con la cabeza de una cláusula tal que el cuerpo de ésta, al ser considerada como meta, se cumpla.

3 Interpretación de PROLOG secuencial

Aprovechando que PROLOG puede ser visto como un lenguaje de especificación, podemos en él especificarlo a él mismo. Podría expresarse la interpretación de un programa PROLOG y una meta en base a que la meta debe empatarse con una cabeza de alguna cláusula o con algún hecho, de manera que el átomo que se elija de la meta será:

1. reemplazado por el cuerpo de la cláusula elegida o bien,
2. eliminado, cuando sea empatado con un hecho.

Lo anterior puede ser una semantica operacional muy útil para el programador pero también para el implantador. Sin embargo en el caso del implantador implica una gran cantidad de operaciones ocultas. Dichas operaciones tratarán de ser desentraadas a lo largo de la presente sección.

Una primera aproximación a la interpretación de metas resultaría ser el siguiente programa:

```
((prueba () x y y))
((prueba (x|x1) y z1 z3)
 (elige y (x2|x3))
 (renom (x2|x3) (y2|y3))
 (unifica x y2 z1 z2)
 (app x1 y3 x4)
 (prueba x4 y z2 z3))
```

donde `(prueba X1 X2 X3 X4)` significa que se va a probar la meta `X1`, con un programa `X2` y una serie de asignamientos a variables `X3` y, si es posible hacer esta prueba, `X4` contendrá los nuevos asignamientos a las variables.

Por ejemplo:

```
?((prueba ((ap (a b) (c d) R))
          (((ap (R|S) T (R|U)) (ap S T U)) ((ap () R R)))
() X)(PP X))
```

Es claro por qué la primera opción de prueba tiene solución inmediata: cuando no hay meta el mismo conjunto de asignamientos a variables se regresa sin cambios. Aun la segunda opción resulta sencilla, pues expresa tratar de efectuar un empate entre un átomo de la cláusula (el primero de `(x|x1)`) y una cláusula que compone el programa `y`. Justamente una cláusula candidata es `(x2|x3)`, donde, naturalmente `x2` es la cabeza y `x3` el cuerpo. El empate se hace con `unifica`, pero antes debemos realizar el renombramiento de las variables. Por eso `(unifica x y2 z1 z2)` representa empatar el átomo seleccionado de la meta y la cabeza renombrada de una cláusula. Finalmente, debemos continuar con los átomos de la cabeza todavía no resueltos y, por supuesto, antes que nada con el cuerpo de la cláusula que empató con el átomo de la meta. Tales operaciones las realizan la apendización del cuerpo renombrado de la cláusula elegida y el resto de la meta y, la continuación recursiva de prueba.

Como parte final de esta sección se presenta una versión que mejora a la anterior.

```
((pru () z z))
((pru (x|x1) z1 z3)
 (elige x x2)
 (resol x x2 z1 z2)
 (pru x1 z2 z3))

((resol x (y|y1) z1 z3)
 (renom y (y2|y3))
 (unifica x y2 z1 z2)
 (pru y3 z2 z3))
((resol x (y|y1) z1 z2)
 (resol x y1 z1 z2))
```

La clásusla `resol` se dedica a probar una meta formada por un sólo átomo. En ella se ha definido adecuadamente la elección de la cláusula que resuelve a una meta considerando el nombre de esta última. La modularidad sobre la solución de cada uno de los átomos proporciona mayor control para implantar por ejemplo cracterísticas extralógicas como el cut. En el apéndice se presenta el programa completo de esta última versión.

4 Prolog Concurrente

Por sencillez y generalidad es común referirse al ejemplo del Productor-Consumidor en el caso de los sistemas concurrentes. Tomaremos el caso más simple: un productor produce eternamente caracteres, el consumidor los consume en tanto se hayan producido. De manera implícita en la definición anterior aparecen ya conceptos de comunicación y sincronización. Puesto que en un momento el productor se aduea del control y lo deja para más adelante volverlo a tomar, tendremos que referirnos a *procesos* más que a fórmulas átomicas que constituyen una meta. Todo esto hacer pensar en cambiar la función de selección de átomos de la meta a resolver dependiendo si es posible "ejecutar un átomo" o no; lo cual se traduce en que si le hace falta un recurso no es posible ejecutarlo. Es necesaria una *lista de procesos* además de la meta que se desee resolver, considerando que, más que una secuencia de átomos, esta meta es una lista de procesos en espera de ser atendidos. Tenemos entonces que expresar parte de estas ideas al nivel del usuario y junto con ellas otras al nivel de implantación. Veremos entonces en primer lugar los conceptos que un usuario requiere para hacer uso de un lenguaje de programación lógica concurrente; esto a través de los conceptos de Prolog Concurrente [4].

La idea principal de la concurrencia en Prolog Concurrente es el empleo de *variables de sólo lectura*. Una variable de sólo lectura es una presentación particular de cualquier variable. Tal presentación o referencia en un programa se hace mediante un sealamiento que indica su uso en modo sólo lectura. El modo sólo lectura implica que no es posible ejecutar un predicado si contiene una variable de este tipo en tanto no se haya instanciado.

Lo importante de las variables de sólo lectura es que, como será aclarado adelante, resultan ser *canales de comunicación* entre procesos y de esta manera premitir lo que se conoce como *paralelismo de flujo* (stream). Más aún, las variables lógicas resultan ser un modelo más de comunicación y sincronización, tal y como los semáforos, eventos, monitores y demás lo son en otros ambientes de programación. Así, para lenguajes como F-Parlog y GHC el modelo de sincronización y comunicación basado en variables lógicas es fundamental. Por eso la mayoría de conceptos que serán expuestos aquí resultan útiles para la programación lógica paralela en general [5]. Aclaremos las ideas vertidas anteriormente considerando la siguiente meta que pondría en funciones a un productor y un consumidor:

```
?((co ?X)(pr X))
```

en esta meta, o lista de procesos a ejecutar, hay una ocurrencia de la variable en modo de sólo lectura: `?X`. Lo cual significa que el consumidor podrá ponerse en funciones si ésta se encuentra instanciada. De no ser así el consumidor será removido de la lista y pasará a una lista de *procesos suspendidos*. Continuará el proceso productor donde la misma variable a la que refiere el consumidor esta en modo normal, el productor tiene la función de instanciar la variable produciendo algún caracter y definiéndolo como la cabeza de la lista `X`. De acuerdo a un criterio de despacho el proceso productor, por ejemplo, agotará su tiempo y será extraído un proceso suspendido. En este momento el consumidor tendrá ya instanciada al menos la cabeza de la lista `X` que está en modo sólo lectura y por ello podrá proceder a realizar sus funciones. También el consumidor agotará su tiempo o bien será suspendido por falta de elementos en su lista, repitiéndose el ciclo.

Es generalizado el uso de variables como `X` del ejemplo anterior, pues como vimos representan canales de comunicación entre procesos. Tales variables se conocen como *variables de flujo*, pues a través de ellas hay un flujo de datos desde un proceso hacia otro. También de importancia sobre las variables de flujo es que realizan funciones de sincronización, debido a que la semántica del lenguaje exige, como ya hemos dicho, la instanciación de una variable en modo sólo lectura para que un proceso pueda seguir ejecutándose.

En toda esta descripción estamos considerando una manera estandar del funcionamiento de estos procesos, pero veamos directamente como se expresan en Prolog Concurrente. En el siguiente ejemplo se presenta la definición de los procesos Productor-Consumidor, amalgamando en ellos la sintaxis de microProlog y la de Prolog Concurrente:

```
((pr X1)
   (EQ X1 (a|X2))
   (pr X2))
((co X1)
```

```
(EQ ?X1 (X3|X2))
(PP ?X3)
(co ?X2))
```

El proceso productor expresa, mediante `EQ`, la instanciación de la variable `X1`, la que hemos identificado como el canal de comunicación que se tiende hacia el consumidor. Como se observa sólo es colocado un elemento en este canal y el resto de los elementos se dejan para los sucesivos llamados, empleando para tal fin la nueva variable `X2`. Por su parte el consumidor ejecuta `EQ`, pero no como una forma de instanciación sino más bien como una desmantelación del contenido de la variable `X1`, pues ésta ocurre en modo sólo lectura. Reiteramos que si `X1` no está instanciada el proceso se suspenderá (`EQ`), deteniendo naturalmente al consumidor. Después de haber desmantelado a `X1`, imprime el valor que le fue enviado, el primer elemento: `X3`. Dejando para procesamiento posterior al resto de su canal de comunicación, siempre que esté instanciada `X2`.

Notemos que de manera casi explícita se encuentra el modelo de ACTOR en la interpretación de los procesos de la programación lógica paralela en general, debido a los siguientes hechos:

- Los actores requieren un buzón del que sustraen mensajes para realizar sus funciones, tal es el caso de las variables stream.
- Las tres operaciones básicas de los actores están consideradas también. `BECOMES` es la recursión: un proceso padre muere dejando a un hijo; `CREATE`, la prueba de algún predicado (llamado a procedimiento) y `SEND` es la instanciación de una parte de una variable stream.
- Cuando el buzón está vacío el actor no puede llevar a cabo tareas. Ello está presente con las variables de sólo lectura sin instanciar.

5 Interpretación de un Prolog No Secuencial

En este momento es posible mostrar el funcionamiento de un intérprete de Prolog No Secuencial. Llamamos así a esta versión porque, aún cuando sigue muy de cerca a Prolog Concurrente, no sigue en rigor ni la sintaxis ni la semántica de ningún lenguaje de programación lógica paralela y sólo ha sido concebido como una muestra de los conceptos de la programación lógica paralela para fines académicos.

Empezaremos por decir que la sintaxis antes mostrada deberá ajustar cambios que permitan su manejo desde microProlog. El programa del Productor-Consumidor quedaría de la siguiente forma:

```
((pr 1) true:
   (= 1 (a|2))
   (pr 2))
((co 1) true:
   (= -1 (3|2))
```

```
(! -3)
(co -2))
```

`EQ` esta sustituido por `=`, `PP` por `!` y las variables son números enteros. Particularmente, aquellas variables que son negativas están en modo sólo lectura (el signo menos esta representando a la interrogación).

Observamos también elementos nuevos de la programación lógica, el *commit* "`:`" y su *guardián*, que es el átomo que le precede (aunque podrían ser varios átomos). El significado de esta combinación es extender la noción de empatamiento de la meta con la cabeza de una cláusula hasta verificar el cumplimiento de ciertas condiciones. Estas condiciones se expresan por el guardián. El commit es sólo una marca sintáctica que indica donde empieza propiamente el cuerpo de la cláusula. Es común que el guardián se utilice para disparar la ejecución de la cláusula que primero satisfaga las condiciones de empatamiento. El guardián puede tener una referencia a una variable en modo sólo lectura y, debido a que en un ambiente paralelo las variables a las que hacen referencia las cláusulas suspendidas están distribuidas en diferentes procesadores, de manera no determinística una y sólo una de las cláusulas que está suspendida será disparada. En el ejemplo anterior el guardián no tiene ningún empleo, pero como la sintaxis lo exige, el productor y el consumidor hacen uso de un guardián que siempre se satisface, de ahí el nombre que tiene (`true`). Más adelante veremos un ejemplo donde el guardián tiene una función importante.

Hemos dicho que la interpretación de metas en PROLOG no secuencial requiere, además de la lista de procesos pendientes, una lista de procesos suspendidos. Es por ello que podemos adaptar fácilmente el intérprete ya analizado de PROLOG secuencial. Consideramos entonces el siguiente predicado:

```
(dem X1 X2 X3 X4)
```

donde `X1` es la lista de procesos suspendidos, `X2` la lista de procesos pendientes, `X3` los asignamientos de inicio a las variables y `X4` el asignamiento final a las variables. De forma semejante habrá que definir un predicado que corresponda a `resol`, el cual considera adicionalmente la lista de procesos suspendidos. En

```
(sol X1 X2 X3 X4 X5)
```

las variables, de acuerdo a su orden de ocurrencia, significan: el proceso a ejecutar, la lista de procesos suspendidos, el procedimiento para resolver el proceso y las listas de asignamientos inicial y final, respectivamente.

Tenemos varios casos que contemplar para dem:

1. todos los procesos se han ejecutado:

   ```
   ((dem () () z z)/)
   ```

2. la lista de procesos pendientes se ha agotado, no así la lista de procesos suspendidos:

```
((dem x () z1 z2)/
 (dem () x z1 z2))
```

3. el tiempo asignado a la ejecución de procesos se ha terminado y hay que efectuar u intercambio entre la lista de pendientes y suspendidos:

```
((dem x y z z1)
 (NOT EQ x ())
 (cota x1)(nivel x2)
 (LESS x1 x2)/
 (SUM x2 30 x3)
 (KILL cota) (PP swap x) (PP ambient z)
 (ADDCL ((cota x3)))
 (dem y x z z1))
```

4. el proceso elegido de la lista de procesos pendientes no tiene variables en modo sólo lectura sin instanciar, por tanto habrá que ejecutarlo (resolverlo):

```
((dem x1 (x|y1) z1 z3)
 (EQ x (X1|X2))
 (dfrost X2 z1)/
 (elige x x2)
 (sol x x1 x2 z1 z2)(PP resolvio x)
 (dem x1 y1 z2 z3))
```

5. el proceso elegido de la lista de procesos pendientes tiene una variable en modo sólo lectura sin instanciar, por tanto éste se traslada a la lista de procesos suspendidos:

```
((dem x1 (x|y1) z1 z2) (PP congelad x)
 (dem (x|x1) y1 z1 z2))
```

El sitio donde se resuelven los predicados interconstruidos es `sol`:

```
((sol true X y z z)/)
((sol (= x1 x2) X y z z1)/
 (unifica x1 x2 z z1))
((sol (! x) X y z z)/
 (bondc x z z x1)
 (PP _ x1))
((sol (< x y) X z1 z z)/
 (bond x z x1)
 (bond y z y1)
 (LESS x1 y1))
```

Ahora se extiende la noción de empate con la cabeza de una cláusula a satisfacer el guardián. Un caso muy frecuente de guardianes es:

```
((sol x X (y|y1) z1 z3)
 (EQ y (x2 true : |x3))
 (renom y (y2 true : |y3))
 (unifica x y2 z1 z2)
 (dem X y3 z2 z3))
```

El otro es el caso general:

```
((sol x X (y|y1) z1 z3)
 (renom y (y2 y4 : |y3))
 (gunifica y4 x y2 z1 z2)
 (dem X y3 z2 z3))
```

Finalmente, y siguiendo a `resol` tenemos el backtracking:

```
((sol x X (y|y1) z1 z2) (PP backtrack x)
 (sol x X y1 z1 z2))
```

Con la anterior descripción de las extensiones de `pru` y `resol` es posible revisar en conjunto el programa del intérprete de PROLOG no secuencial, que comparte mucho con el de PROLOG secuencial. Por último analizaremos un ejemplo muy citado en la programación lógica paralela.

El quicksort es un algoritmo interesante porque sigue la estrategia de "divide y vencerás", lo cual permite su paralelización, pero además es atractivo porque muestra elementos de paralelismo de tipo flujo. El programa en PROLOG no secuancial es el siguiente:

```
((quicksort () 1 1) true:)
((quicksort (1|2) 3 4) true:
   (parte -2 1 5 6)
   (quicksort -5 3 (1|7))
   (quicksort -6 7 4))
```

Los argumentos del procedimiento `quicksort` son: la lista a ordenar, la lista ordenada y una lista previamente ordenada que se apendiza a la lista ordenada. El procedimiento que particiona, `(parte X1 X2 X3 X4)`, regresa dos listas, la de menores (`X3`) y la de mayores (`X4`), respecto a un valor (`X2`), distribuye de la lista original (`X1`) los elementos en una de las resultantes, para ello se vale de "conocer", si el primer elemento de `X1` es menor o mayor que el pivote. Notemos que `parte` no procede mientras no esté instanciada su lista a particionar. Pero en cuanto tenga al menos el primer elemento puede proceder. Esta característica se conoce como paralelismo de flujo.

```
((parte () 1 () ()) true : )
((parte (1|2) 3 4 (1|5))
   (< 3 1) : (! 1)
   (parte -2 3 4 5))
((parte (1|2) 3 (1|4) 5)
   (<= 1 3) : (! 1)
   (parte -2 3 4 5))
```

A Listado del intérprete de Prolog secuencial

```
((COMEN ejemplo de programa))
((DB
( ((ap (R|S) T (R|U))
   (ap S T U))
  ((ap () R R)) )
( ((re () R R))
  ((re (R|S) T U)
   (re S (R|T) U)) )
))

((pl x)
 (KILL nivel)
 (ADDCL ((nivel 64)))
 (TIME z1 z2 z3)(PP z1 z2 z3)
 (pru x () z)
 (TIME y1 y2 y3)(PP y1 y2 y3)
 (PP z))

((COMEN prueba una meta conjuntiva))
((COMEN pru meta asigna-inicio asigna-final))
((pru () z z))
((pru (x|x1) z1 z3)
 (elige x x2)
 (resol x x2 z1 z2)
 (pru x1 z2 z3))

((COMEN resolver una meta))
((COMEN resol meta procedimiento asigna-inicio asigna-final))
((resol x (y|y1) z1 z3)
 (renom y (y2|y3))
 (unifica x y2 z1 z2)
 (pru y3 z2 z3))
((resol x (y|y1) z1 z2)
 (resol x y1 z1 z2))

((COMEN unifica dos expresiones))
((COMEN unifica expr1 expr2 asigna-inicio asigna-final))
((unifica x y z z)
 (bond x z x1)
 (bond y z y1)
 (atomo x1)
 (atomo y1) /
 (EQ x1 y1))
((unifica x y z ((x1 y1)|z))
```

```
 (bond x z x1)
 (bond y z y1)
 (var x1) /)
((unifica x y z ((y1 x1)|z))
 (bond x z x1)
 (bond y z y1)
 (var y1) /)
((unifica x y z z2)
 (bond x z (x1|x2))
 (bond y z (y1|y2))
 (unifica x1 y1 z z1)
 (unifica x2 y2 z1 z2))

((COMEN verifica si es indestructible))
((atomo ()))
((atomo x)
 (NUM x) /)
((atomo x)
 (NOT var x)
 (NOT LST x))

((COMEN lista de variables b\'asicas))
((Var R))
((Var S))
((Var T))
((Var U))
((Var V))
((Var W))

((COMEN verifica si es una variable de alg\'un nivel))
((var x)
 (STRINGOF (y|z) x)
 (Var y))

((COMEN enlaza una variable con su valor))
((COMEN bond variable asignaciones valor))
((bond x () x) /)
((bond x z x)
 (NOT var x) /)
((bond x ((x x1)|z) x1) /)
((bond x ((y y1)|z) x1)
 (bond x z x1))

((COMEN renombra una expresi\'on))
((COMEN reno expresi\'on nueva-expresi\'on))
((reno x x)
```

```
 (atomo x))
((reno x y)
 (Var x)
 (nivel z1)
 (CHAROF z2 z1)
 (STRINGOF (x z2) y))
((reno (x|y) (x1|y1))
 (reno x x1)
 (reno y y1))

((COMEN renombra las variables que ocurren en una expresi\'on))
((COMEN obtiene el \'{\i}ndice de renombramiento y define las nuevas variables con el))
((COMEN renom expresi\'on nueva-expresi\'on))
((renom x y)
 (DELCL ((nivel z1)))
 (SUM z1 1 z2)
 (ADDCL ((nivel z2)))
 (reno x y))

((COMEN inicia la elecci\'on de un procedimiento))
((COMEN elige \'atomo procedimiento))
((elige (x|x1) y)
 (DB|z)
 (elig x z y))

((COMEN busca en la lista de procedimientos de acuerdo al nombre))
((COMEN elig nombre lista))
((elig x ((((x|x1)|x2)|x3)|x4) (((x|x1)|x2)|x3) ))
((elig x (x1|x2) z)
 (elig x x2 z))
```

B Listado del intérprete de Prolog No Secuencial

```
((C cada lista contiene todas las
clausulas con el mismo nombre))
((DB
( ((= 1 1)) )
( ((! 1)) )
( ((m 1 2)) )

( ((pr 1) true:
   (= 1 (a|2))
   (pr 2))
```

```
   )
( ((co 1) true:
   (= -1 (3|2))
   (! -3)
   (co -2))
   )
( ((ap () 1 1) true :)
  ((ap (1|2) 3 (1|4)) true :
   (! 1)(ap 2 3 4))
   )
( ((ac () 1 1) true :)
  ((ac (1|2) 3 (1|4)) true :
   (ac -2 3 4))
   )
( ((re () 1 1) true :)
  ((re (1|2) 3 4) true:
   (re 2 (1|3) 4))
   )
))


((C ejecuta procesos))
((C dem procesos-suspendidos procesos asigna-inicio asigna-final))
((dem () () z z)/)
((dem x () z1 z2)/
 (dem () x z1 z2))
((dem x y z z1)
 (NOT EQ x ())
 (cota x1)(nivel x2)
 (LESS x1 x2)/
 (SUM x2 30 x3)
 (KILL cota) (PP sw x) (PP env z)
 (ADDCL ((cota x3)))
 (dem y x z z1))
((dem x1 (x|y1) z1 z3)
 (EQ x (X1|X2))
 (dfrost X2 z1)/
 (elige x x2)
 (sol x x1 x2 z1 z2)(PP reso x)
 (dem x1 y1 z2 z3))
((dem x1 (x|y1) z1 z2) (PP fr x)
 (dem (x|x1) y1 z1 z2))

((C verifica que el predicado este descongelado))
((dfrost x z)
 (atomo x)/)
```

```
((dfrost x z)
 (NUM x)
 (LESS 0 x)/)
((dfrost x z)
 (NUM x)
 (LESS x 0)
 (TIMES -1 x y)
 (bondc y z z x1)
 (freez x1)/
 FAIL)
((dfrost x z)
 (NUM x)
 (LESS x 0)/)
((dfrost (x|y) z)
 (dfrost x z)
 (dfrost y z))

((C prueba de variable congelada))
((freez x)
 (NUM x))
((freez (x1|x2))
 (NUM x1))

((C ejecuta un proceso))
((C sol proceso procesos-suspendidos lista-clausulas asigna-inicio asigna-final))
((sol true X y z z)/)
((sol (= x1 x2) X y z z1)/
 (unifica x1 x2 z z1))
((sol (! x) X y z z)/
 (bondc x z z x1)
 (PP _ x1))
((sol (m x y) X z1 z z)/
 (bond x z x1)
 (bond y z y1)
 (LESS x1 y1))
((sol x X (y|y1) z1 z3)
 (EQ y (x2 true : |x3))
 (renom y (y2 true : |y3))
 (unifica x y2 z1 z2)
 (dem X y3 z2 z3))
((sol x X (y|y1) z1 z3)
 (renom y (y2 y4 : |y3))
 (gunifica y4 x y2 z1 z2)
 (dem X y3 z2 z3))
((sol x X (y|y1) z1 z2) (PP btk x)
 (sol x X y1 z1 z2))
```

```
((C unifica y guarda))
((gunifica y x1 x2 z1 z2)
 (unifica x1 x2 z1 z3)
 (sol y () () z3 z2))

((C enlaza una expresion con su valor))
((bond (x|y) z (x1|y))
 /(bond1 x z x1))
((bond x z x)
 (NOT NUM x)/)
((bond x z y)
 (LESS x 0)/
 (TIMES x -1 x1)
 (bond1 x1 z y))
((bond x z y)
 (bond1 x z y))

((C busca el valor de una variable))
((bond1 x () x) /)
((bond1 x ((x x1)|z) x1) /)
((bond1 x ((y y1)|z) x1)
 (bond1 x z x1))

((C toma en cuenta las variables de modo lectura))
((bondc x y z x1)
 (LESS x 0)/
 (TIMES x -1 y1)
 (bondc y1 y z x1))
((bondc x () z x)/)
((bondc x ((x x1)|z) z1 x1)
 (atomo x1)/)
((bondc x ((x x1)|z) z1 y1)
 (NUM x1)/
 (bondc x1 z1 z1 y1))
((bondc x ((x (x1|x2))|z) z1 (y1|x2))/
 (bondc x1 z1 z1 y1))
((bondc x ((y x1)|z) z1 y1)
 (NOT EQ x y)/
 (bondc x z z1 y1))
```

Referencias

[1] Robinson, J.A. (1965), 'A machine oriented-logic based on the resolution principle', *J. of ACM*, 12, pp 23-41.

[2] Kowalski, R.A. (1974), 'Predicate logic as a programming language', en *Proc. of 4th IFIP world Congress*, Amsterdam, North-Hollan, pp 569-574.

[3] Logic Programming Associates (1984), Micro-Prolog 3.1 Programmer's Reference Manual, England.

[4] Kahn K. (1986), 'Objects in Concurrents Logic Programming Languages' en *OOPSLA, ACM*, pp 242-257.

[5] Takeuchi A., Furukawa K. (1982), 'Parallel Logic Programming Languages', en *LNCS 144*, Springer Verlag.