# Planificación de Procesos para la Interpretación de Programación en Lógica Concurrente en PROLOG Secuencial

Ana G. Maguitman[1] [2] Claudio Delrieux[3]

Grupo de Investigación en Inteligencia Artificial (GIIA)
Instituto de Ciencias e Ingeniería de la Computación (ICIC)
Universidad Nacional del Sur
Av.Alem 1253 - (8000) Bahía Blanca, ARGENTINA
e-mail: {ccmagit,usdelrie}@criba.edu.ar

**Resumen**

Entre las propiedades esperadas para un *lenguaje declarativo ideal* podemos destacar la *abstracción de control*. En un programa, condicionar los resultados al orden de ejecución secuencial nos lleva a realizar supuestos que van más allá de su verdadero significado lógico. Presentamos un *lenguaje lógico concurrente* que permitirá liberarnos de estas imposiciones de control que desvirtúan a los lenguajes lógicos secuenciales (PROLOG).

En este trabajo, analizamos las características del lenguaje presentado, así como su implementación mediante un metaintérprete PROLOG. Mostramos las cláusulas con guardas, sus componentes y significado lógico. Comparamos la semántica operacional del nuevo modelo con la de PROLOG, lo que nos llevará al estudio de la semántica de procesos, del matching, del chequeo de guardas y de la interpretación don't care de no determinismo. Describimos un algoritmo de planificación que permite atender procesos derivados de la computación de manera ecuánime y respetando las restricciones impuestas por el programador, quien dispondrá de herramientas especiales para controlar la evaluación de las consultas.

Finalmente damos una serie de programas como ejemplo para investigar nuevas técnicas y aplicaciones que pasan a ser posibles gracias a la potencialidad del lenguaje en cuestión.

[1]Becaria de la Universidad Nacional del Sur, República Argentina.
[2]Departamento de Ciencias de la Computación,UNS.
[3]Departamento de Ingeniería Eléctrica,UNS.

# 1 Introducción

Muchas de las ventajas declarativas de la programación en lógica se pierden en su ejecución secuencial (PROLOG[6]) dado que no es posible una auténtica abstracción de control. Sin embargo, estas mismas suposiciones relativas al orden de evaluación no son necesarias en lo lenguajes de *programación en lógica concurrente*. Estos lenguajes, como manifestación concreta de la programación en lógica, son especialmente aptos para rescatar desde un nivel superior, y por lo tanto más abstracto, la naturaleza del formalismo en el que se basan.

El presente trabajo consiste en el estudio e implementación mediante un *metaintérprete* PROLOG de un modelo computacional para un *lenguaje concurrente de programación en lógica*. El sistema resultante, al estar basado en un formalismo lógico, tiene una serie de ventajas ya conocidas como ser la declaratividad y el uso de términos lógicos parcialmente instanciados para representar estructuras de datos.

Operacionalmente, el modelo consiste en un conjunto dinámico de procesos concurrentes que cooperan con el fin de resolver la consulta presentada por el usuario. Esto hace necesario un método apropiado de planificación que permita conmutar de un proceso a otro simulando paralelismo y permitiendo sincronización y comunicación.

La *estrategia de control* para este mecanismo computacional está compuesta por un *evaluador* y un *lenguaje de control* que se diferencian en varios aspectos a los provistos por PROLOG:

- El *evaluador* utiliza dos estructuras, distintas a una pila, para mantener los procesos asociados a las metas que aún no han sido resueltas.

  Si bien el procedimiento de prueba es similar a la *resolución*, veremos que existe una diferencia en la manera de generar ligaduras para las variables.

  Emplea la interpretación *don't care* de no determinismo en la cual no hay retorno o *backtracking*, característica saliente del evaluador de PROLOG.

- El *lenguaje de control* incorpora elementos que darán al programador mayor poder sobre la computación: *operadores para especificar conjunción paralela o secuencial de metas*, *anotaciones de sólo lectura* y *guardas*.

# 2 Un lenguaje de programación en lógica concurrente

## 2.1 Sintaxis

Los *programas* escritos en el *lenguaje de programación en lógica concurrente* están compuestos por un conjunto finito de *cláusulas con guardas*. La sintaxis para este lenguaje es similar a CONCURRENT PROLOG[5]. El esquema general para una cláusula con guarda es el siguiente:

$$H \leftarrow G_1 \& G_2 \& \ldots \& G_n << B_1 c_1 B_2 c_2 \ldots c_{k-1} B_k,$$

donde $H$, $G_1$, $G_2$,...$G_n$, $B_1$,$B_2$,...$B_k$, son átomos.

$H$ es llamada *cabeza* de la cláusula. El símbolo "<<" que divide la cláusula en *guarda* y *cuerpo* se denomina *operador de compromiso* (*commit operator*). Si la guarda está vacía, este operador es omitido. Para simplificar la semántica operacional del lenguaje, las guardas sólo pueden contener un cierto tipo de predicados: del sistema o hechos[4].

Los átomos que componen el cuerpo de la cláusula posiblemente contengan anotaciones de sólo lectura: un término es de sólo lectura si tiene la forma:

*término*?

Los conectivos $c_1$, $c_2$,...,$c_{k-1}$ pueden ser: "\\" o "&" para denotar conjunción paralela o estrictamente secuencial de átomos respectivamente[5].

## 2.2 Semántica Declarativa

De manera similar a las cláusulas definidas, las cláusulas con guarda tienen una lectura lógica:

- Todas las cláusulas se asumen universalmente cuantificadas. Las anotaciones de sólo lectura no tienen significado lógico;
- "←" denota implicación lógica;
- "<<", "&" y "\\" denotan conjunción.

## 2.3 Semántica Operacional

La semántica operacional de nuestros programas lógicos concurrentes es sustancialmente diferente a la empleada por PROLOG. En nuestro sistema, la consulta del usuario se transforma en una *red de procesos concurrentes* que se *comunican* instanciando variables compartidas, se *sincronizan* esperando que una variable compartida sea instanciada y se *suspenden* cuando hay una restricción de secuencialidad que respetar.

### 2.3.1 Semántica de procesos

En PROLOG vemos cada meta como una llamada a un procedimiento y cada cláusula como la definición de un procedimiento. En nuestro lenguaje las metas se ven como procesos y las cláusulas determinan el comportamiento que deben adoptar los procesos:

- En "$A \leftarrow G << true$" el proceso asociado a la meta $A$ es terminado.
- En "$A \leftarrow G << B$" el proceso asociado a la meta $A$ es cambiado por un proceso asociado a la meta $B$.

[4]Tales guardas se denominan *flat*.

[5]El operador "&" precede al operador "\\" y la asociatividad de ambos es de derecha a izquierda. Tanto asociatividad como precedencia pueden ser modificadas mediante el uso de paréntesis.

- En "$A \leftarrow G << B_1 c_1 B_2 c_2 ... c_{k-1} B_k$" el proceso asociado a la meta $A$ es convertido en $k$ procesos concurrentes asociados a las $k$ metas $B_i$.
  - El conectivo "&" hace que los procesos que involucra se ejecuten secuencialmente (de acuerdo al orden textual).
  - El conectivo "\\" determina la ejecución paralela de procesos.

### 2.3.2 Matching y chequeo de guardas

Otra característica de la semantica operacional del sistema es que en lugar de usar *unificación*, emplea *matching* y *chequeo de guardas*. Lo que hace distinto el *matching* a la *unificación* es el agregado de anotaciones de sólo lectura, las que restringen las sustituciones permitidas para las variables de la meta:

$Término_1?$ hace *match* con $Término_2$ si $Término_1 = Término_2\Theta$.

Si `match(Meta,Cabeza)` no es exitoso, en vez de fallar se analiza la posibilidad de que en un futuro valga `match(Meta,Cabeza)`. Esto podrá ocurrir si ciertas variables de sólo lectura de `Meta` son instanciadas. Si el testeo de un *match potencial* entre `Meta` y `Cabeza` resulta exitoso, se suspende el proceso asociado a la `Meta` actual para luego retomarlo.

**Ejemplos de matching**

1) $match(p(término_1), p(término_2))$ coincide con $unificar(término_1, término_2)$ si $término_1$ no tiene variables de sólo lectura.
2) $match(p(a?), p(X))$ devuelve $\Theta = \{X \leftarrow a\}$.
3) $match(p(b?), p(a))$ *falla.*
4) $match(p(X?), p(Y))$ devuelve $\Theta = \{Y \leftarrow X\}$.
5) $match(p(X?), p(a))$ *suspende.*
6) $match(p([1|X]?), p([C|Y]))$ devuelve $\Theta = \{C \leftarrow 1, Y \leftarrow X\}$.
7) $match(p(X?), p([C|Y])$ *suspende.*

Utilizando *matching*, una cláusula está habilitada siempre y cuando tenga sus variables lo suficientemente instanciadas. El mecanismo de *matching* es simple pero lo bastante potente como para resolver tareas de sincronización complejas.

El *chequeo de guardas* es un mecanismo que permite testear parte de la definición de la cláusula antes de escogerla. Decimos que una meta $A$ se *compromete* (commits) con una cláusula:

$$H \leftarrow G_1 \& G_2 \& ... \& G_n << B_1 c_1 B_2 c_2 ... c_{k-1} B_k,$$

si `match(A,H)`$=\Theta$ y si `chequeo(`$(G_1 \& G_2 \& ... \& G_n)\Theta$`)` tiene éxito. Una meta puede también comprometerse con un predicado del sistema.

En caso de que no se haya producido un compromiso, aún es posible que sí se lleve a cabo en un estado futuro de la computación. Esto significa que en el estado actual, las variables de la meta no están lo suficientemente instanciadas, por lo cual el proceso asociado a dicha meta deberá "esperar".

### 2.3.3 No detemininsmo "don't care"

Adoptamos la interpretación "don't care" de no determinismo. Con este tipo de no determinismo, al sistema "no le importa" cómo obtener la solución y por lo tanto no necesita buscar. Esto significa que no habrá *backtracking* sino que una vez que la computación se comprometa con un camino, no explorará otras alternativas[6]. Aquí se ve con claridad la necesidad de incorporar *guardas* en las cláusulas, ya que así habrá un testeo previo a la adopción de un camino para la computación (del cual no se podrá volver). Es entonces el programador quien le facilita al sistema información relativa a como encontrar la solución.

# 3 Un algoritmo para la planificación secuencial de procesos concurrentes

Presentada una consulta, el sistema debe crear una serie de procesos encargados de resolverla. La creación dinámica de estos procesos requiere de algún algoritmo de planificación, así como de estructuras flexibles para mantener el estado global del sistema.

Los procesos pueden estar en tres estados:

1. **Corriendo:** están siendo *realmente procesados.*
2. **Listos:** están preparados para ser atendidos.
3. **Dormidos:** no pueden ser atendidos hasta que hayan finalizado otros procesos[7].

La planficación consiste en hacer cambios de un proceso a otro atendiéndolos de manera intercalada, como si cada proceso tuviera su *procesador virtual*[8]. Para ello adoptaremos una estrategia de *multitarea* similar a la conocida como *round robin.* Se trata de mantener una cola de procesos listos, cada uno con un "tiempo máximo" de ejecución o *quantum* asociado. Un proceso (y cualquier subproceso que genere) es atendido mientras no se agote su *quantum.* Una vez agotado el mismo, el proceso es enviado al final de la cola y el control pasa al proceso siguiente. Esto sobrecarga en gran medida al sistema pero asegura que todos los procesos serán eventualmente atendidos. La computación es **exitosa** si el sistema llega a un estado en el cual todos los procesos han terminado, y **fracasa** si no es posible resolver la meta asociada a alguno de los procesos.

[6]El backtracking es un método que permite implementar otro tipo de no determinismo conocido como no determinismo "don't know ". Un sistema que adopta este tipo de no determinismo "no sabe " como encontrar la solución y por lo tanto necesita buscar por varios caminos alternativos.

[7]Hay un cuarto estado en el cual podrían caer los procesos. Este estado conocido como **deadlock** es aquel en el que la computación queda bloqueada indefinidamente. En este trabajo no nos preocuparemos por este estado.

[8]La frecuencia de estos cambios está dada por el parámetro $Q$ que define el *quantum.* Este valor puede ser modificado mediante los programas y/o consultas.

## 3.1 Estructuras de datos utilizadas

El estado global del sistema está dado por el predicado:

`estado(Listos,Dormidos,Cont_Id,Q).`

El estado queda entonces determinado por las siguientes variables y estructuras de datos:

- `Listos`: cola[9] de procesos preparados para ser activados. Los procesos *listos* tienen la forma:

  $p_L$`(Meta,TimeOut,Suspende)`

  - `Meta`: meta a ser resuelta por el proceso. Puede tener la forma:
    1. *true.*
    2. *Predicado del sistema.*
    3. *Predicado del programa.*
    4. *Meta* \\ *Meta.*
    5. *Meta & Meta.*
  - `TimeOut`: mantiene el tiempo que le queda al proceso y a los subprocesos generados por el mismo para resolver la meta antes de ser enviados al final de la cola.
  - `Suspende`: mantiene un identificador asociado a un proceso que suspende en la lista de `Dormidos`, mantiene 0 si no suspende a ningún proceso.

- `Dormidos`: conjunto de procesos suspendidos por uno o más procesos que aún no han sido finalizados. Los procesos *dormidos* tienen la forma:

  $p_D$`(Identificador,Meta,Cont_Susp,Suspende)`

  - `Identificador`: identificador del proceso dormido.
  - `Meta`: idem `Meta` para `Listos`
  - `Cont_Susp`: contador de procesos bajo los cuales está suspendido el actual. Este no podrá ingresar a la cola de `Listos` hasta que `Cont_Susp` sea 0. `Cont_Susp` se decrementa en 1 cada vez que un proceso que lo suspendía se termina.
  - `Suspende`: idem `Suspende` para `Listos`

- `Cont_Id`: guarda el número usado como último identificador para un proceso dormido.

- `Q`: *quantum* o porción máxima de *tiempo* asignada a un proceso (y sus subprocesos).

[9] Diremos que es una "cola" aunque en realidad en ocasiones, como veremos, se comporta como una "pila".

## 3.2 Algoritmo de planificación

La planificación se llevará a cabo de acuerdo al siguiente algoritmo:

Dada la consulta ← *Consulta*, inicializar el estado del sistema:

- Listos=[$p_L$(Consulta,Q,0)].
- Dormidos=[ ].
- Cont_Id=0.
- Q=quantum seteado por el programador y/o usuario.

REPETIR

- Si la cola de *Listos* está vacía devolver *EXITO!* y terminar.
- Sea $p_L$*(Meta,TimeOut,Suspende)*, el primer proceso de la cola de procesos *listos* con *TimeOut*$\geq$*1* (los procesos con *TimeOut=0* son puestos al final de la cola de *Listos* con *TimeOut=Q*).
  - Si *Meta=true*: si *Suspende*$\geq$*1* buscar en la lista de *Dormidos* el proceso $p_D$ cuyo *Identificador* coincida con *Suspende.*
    - Si en $p_D$, *Cont_Susp=1* entonces despertar al proceso $p_D$ y agregarlo al principo de la cola de *Listos* con su *Meta* y valor de *Suspende* y conservando el *TimeOut* del proceso que lo despertó decrementado en uno.
    - Si *Cont_Susp*$\geq$*2* hacer *Cont_Susp:=Cont_Susp-1* y mantener el proceso $p_D$ dormido.
  - Si *Meta=Predicado del sistema*: hacer "call(Meta)".
    Si *Suspende*$\geq$*1* buscar en la lista de *Dormidos* el proceso $p_D$ cuyo *Identificador* coincida con *Suspende.*
    - Si en $p_D$, *Cont_Susp=1* entonces despertar al proceso $p_D$ y agregarlo al principo de la cola de *Listos* con su *Meta* y valor de *Suspende* y conservando el *TimeOut* del proceso que lo despertó decrementado en uno.
    - Si *Cont_Susp*$\geq$*2* hacer *Cont_Susp:=Cont_Susp-1* y mantener el proceso $p_D$ dormido.
  - Si *Meta=Predicado del programa*:
    1. Si existe una cláusula $H \leftarrow G << B$, con la que *Meta* pueda comprometerse con unificador $\Theta$, entonces poner al principio de la cola de *Listos* el proceso:
    $$p_L(B\Theta, TimeOut - 1, Suspende).$$
    2. Si no vale 1, si existe una cláusula $H \leftarrow G << B$, con la que *Meta* pueda suspenderse, entonces poner al final de la cola de *Listos* el proceso:
    $$p_L(Meta, Q, Suspende).$$
    3. Si no vale 1 ni 2 entoces devolver *FRACASO!* y terminar.
  - Si $Meta = Meta_1 \setminus\setminus Meta_2$: agregar al principio de la cola de *Listos* los procesos:
    $$p_L(Meta_1, TimeOut//2, Suspende)$$
    $$p_L(Meta_2, TimeOut//2, Suspende)$$

Si *Suspende*$\geq 1$ buscar en la lista de *Dormidos* el proceso $p_D$ con *Identificador=Suspende* y hacer *Cont_Susp=Cont_Susp+1*.

- Si $Meta = Meta_1 \& Meta_2$, crear un nuevo *Identificador*.
  Agregar al principio de la cola de *Listos* el proceso:

$$p_L(Meta_1, TimeOut - 1, Identificador)$$

y agregar a la lista de *Dormidos* el proceso:

$$p_D(Identificador, Meta_2, 1, Suspende).$$

FIN REPETIR

## 3.3 Algunos detalles técnicos

Para cambiar de un estado a otro se utilizó el mecanismo conocido como *lazo dirigido por falla*. El control iterativo fue preferido sobre el recursivo con el fin de no agotar rápidamente el espacio de la *pila global del sistema*.

A la estructura de datos utilizada para mantener los procesos listos la hemos llamado "cola", lo que no es del todo preciso. Si bien la estrategia *round robin* se implementa sobre una cola, en nuestro caso, por la naturaleza lógica de los procesos que manipulamos también ulilizamos la estructura como "pila". Es claro que la utilizamos como cola cuando pretendemos simular paralelismo y por lo tanto atendemos a los procesos con la norma *primero en llegar, primero en salir*. Sin embargo, cuando un proceso genera subprocesos y estos últimos a la vez generan más subprocesos, estos son atendidos bajo la disciplina *último en llegar, primero en salir*, siempre que les reste *quantum*. En este caso la estructura cumple la función de una pila.

Cuando un programa es cargado para ser interpretado por nuestro sistema, éste es almacenado en la base de datos subyacente provista por PROLOG. PROLOG almacena sus términos utilizando un mecanismo de hash, creando una clave a partir del functor principal del término y la aridad del mismo. Para los términos que nosotros queremos almacenar: *cláusulas con guardas*, este mecanismo no es el más beneficioso ya que la mayoría de los términos se almacenarán bajo las claves "← /2" o "<< /2", lo que no hará eficiente el acceso a los mismos. Una posible mejora consitiría en preprocesar los programas y almacenar sus cáusulas creando la clave con el functor y aridad utilizados para la cabeza de las mismas, lo que luego hará más rápida la búsqueda.

# 4 Programación en lógica concurrente

Examinaremos ejemplos y técnicas de programación para el lenguaje presentado.

## Productores, consumidores y traductores

Una de las características más interesantes del lenguaje presentado, es que permite construir listas incrementalmente (*streams*). Los procesos encargados de generar estas listas reciben el nombre de productores y los que toman estas listas como entrada son llamados consumidores. Hay también procesos que a partir de una lista de entrada generan otra incrementalmente, estos se denominan traductores.

### Productor Incremental de números de Fibonacci[5]

Veamos la definición de un productor de un lista potencialmente infinita conteniendo la secuencia ordenada de números de Fibonacci:

```
fib(Ns) <- fib(0,1,Ns).

fib(N1,N2,Ns) <- Ns=[N1|Ns1] & N3 is N1+N2 & fib(N2,N3,Ns1).
```

Posiblemente sea de interés visualizar los números generados. Para ello podemos definir un consumidor:

```
consumir([Actual|Resto]) <- write(Actual?)  & write(' ') &
          consumir(Resto?).
```
[10]

Ahora sólo resta pedir que productor y consumidor se ejecuten en paralelo y que el argumento del consumidor sea de sólo lectura:

```
todos <- fib(Fs) \\ consumir(Fs?).
```

El proceso asociado a la meta **todos** nos devolverá uno a uno todos los números de Fibonacci.

### Productor incremental de números de Hamming[5]

El siguiente programa genera y muestra una lista de todos los números de la forma $2^i3^j5^k$.

El productor *Hamming* está definido a partir de los traductores *múltiplos* y *omerge*. La activación paralela de estos traductores da como resultado la creación de un conjunto de procesos cooperantes que se comunican y sincronizan a través de variables compartidas.

```
hamming([1|Xs]) <- multiplos([1|Xs?],2,X2) \\ multiplos([1|Xs?],3,X3) \\
          multiplos([1|Xs?],5,X5) \\ omerge(X2?,X3?,X23) \\
          omerge(X23?,X5?,Xs).

omerge([X|In1],[X|In2],Out) <- Out=[X|Out1] & omerge(In1?,In2?,Out1).

omerge([X|In1],[Y|In2],Out) <- X<Y << Out=[X|Out1] &
          omerge(In1?,([Y|In2])?,Out1).

omerge([X|In1],[Y|In2],Out) <- Y<X << Out=[Y|Out1] &
          omerge(([X|In1])?,In2,Out1).

omerge([],In2,Out) <- In2=Out.
```

[10] Acá se ve claramente la necesidad de un conectivo *and secuencial*. Este obliga a que **Actual** sea escrito antes que **Resto**.

```
omerge(In1,[],Out) <- In1=Out.

multiplos([],N,Out) <- Out=[].

multiplos([X|In],N,Out) <- Out=[Y|Out1] & Y is X*N & multiplos(In?,N,Out1).
```

### Criba de Eratóstenes[4]

El siguiente programa es una implementación paralela de la criba de Eratóstenes. Consiste en un productor de una lista con los números naturales, *natnums*, y de un conjunto dinámico de *filtros*, uno por cada número primo encontrado en la lista. Estos *filtros* se encargarán de borrar los múltiplos del primo que ellos representan del resto de la lista. El resultado es una lista incremental de números primos.

```
primos(Ps)<-natnums(2,Ns) \\ criba(Ns?,Ps).

natnums(Low,Ns)<-M is Low+1 \\ Ns=[Low|Rest] \\ natnums(M?,Rest).

criba([],Ps)<-Ps=[].

criba([P|Ns],Ps)<-Ps=[(P?)|P1s] \\ filtrar(P?,Ns?,Fs) \\ criba(Fs?,P1s).

filtrar(P,[],Ps)<-Ps=[].

filtrar(P,[N|Ns],Fs)<-0=\=N mod P << Fs=[N|F1s] \\ filtrar(P?,Ns?,F1s).

filtrar(P,[N|Ns],Fs)<-0=:=N mod P << filtrar(P?,Ns?,Fs).
```

Las soluciones a problemas basadas en el principio "divide y conquistarás" son especialmente aptas para ser implementadas en un lenguaje de programación concurrente. Esto es debido especialmente a que podemos realizar una mayor abstracción de control, pues no debemos preocuparnos por los efectos, a veces no deseados, del orden de evaluación secuencial impuesto por PROLOG estandar.

### Torres de Hanoi

Presentamos como ejemplo un programa que soluciona el problema de las *Torres de Hanoi*:

```
hanoi(1,A,B,C,[mover(A,B)]).
hanoi(N,A,B,C,Moves) <- N≥ 2 << N1 is N-1 &
          (hanoi(N1,A,C,B,Ms1) \\ hanoi(N1,C,B,A,Ms2) \\
          append(Ms1?,[mover(A,B)|Ms2?],Moves)).
```

Si realizamos un análisis comparativo entre una implementación en PROLOG secuencial y una en nuestro lenguaje de programación en lógica concurrente del problema de las *Torres de Hanoi* notaremos que en el primer caso, como consecuencia de la estrategia de control adoptada: *de izquierda a derecha* y *en profundidad*, no tendremos ningún tipo de resultado hasta que la computación finalice. Sin embargo, en el segundo caso, contamos con un *productor incremental* de resultados parciales que se vuelven disponibles a medida que son calculados. Estos pueden ser visualizados si agregamos al programa un proceso *consumidor* que se encargue de mostrarlos por pantalla y que se ejecute concurrentemente con el *productor* definido, compartiendo con éste la lista incremental (stream) de resultados. Esta ventaja se vuelve aún más notoria en los ejemplos previos: *Productor incremental de números de Fibonacci*, *Productor incremental de números de Hamming* y *Criba de Eratóstenes*, pues los procesos que involucran son "eternos"y las listas que producen son potencialmente infinitas. Por lo tanto, es realmente importante contar con la posibilidad de acceder a dichas listas mientras están parcialmente instanciadas, lo que sólo puede lograrse mediante la incorporación de procesos consumidores que se ejecuten en paralelo a los productores.

# 5 Conclusiones

Hemos analizado e implementado un lenguaje de programación que conjuga una serie de características:

- **Declaratividad**: el algoritmo de ejecución, al no hacer necesaria la suposición de evaluación secuencial, explota en mayor medida que PROLOG la **naturaleza declarativa** de la programación en lógica.
- **No determinismo "don't care"**, forma *inteligente* de no determinismo mediante la cual el sistema no debe buscar soluciones ciegamente, sino utilizando la información provista por el programador.
- Uso de **mensajes parcialmente instanciados** como canales de comunicación entre procesos.
- **Guardas, operador de conjunción secuencial, operador de conjunción paralela** y **anotaciones de sólo lectura** que dan al programador cierto control sobre la computación.

El mecanismo computacional descripto cuenta con:

- **Matching**, que permite la sincronización de procesos paralelos, administrada por disponibilidad de sustituciones a variables.
- Planificación **ecuánime** de procesos: cualquier proceso que pueda ocurrir, debe eventualmente ocurrir.

Para una implementación final, sería de interés resolver una serie de problemas:

- Implementar un mecanismo que controle que las guardas sean "seguras"[1]. Esto significa que no exista la posibilidad de instanciar a través del testeo de la guarda, variables de la cabeza de la cláusula. Este control podría realizarse de manera estática mediante un preprocesamiento de los programas a interpretar.

- Es posible que la computación caiga en un estado de "deadlock ": varios procesos pueden bloquearse, esperando indefinidamente la instanciación de sus variables. Sería importante agregar al metaintérprete el poder de impedir deadlocks o de detectarlos y recuperarse de ellos.

- Cuando el *matching* entre la meta y la cabeza de una cláusula queda suspendido, la meta pasa al final de la cola de procesos listos. Una mejora consistiría en utilizar una estrucura auxiliar para almacenar los procesos asociados a metas suspendidas y un mecanismo que facilite la activación de estos procesos una vez que estén realmente listos.

El prototipo de sistema que hemos implementado ha sacrificado eficiencia a cambio de algunos beneficios. La lentitud de la ejecución se debe en parte a la sobrecarga que involucra la estrategia adoptada para planificar los procesos, que hace necesario manipular estructuras dinámicas con metodos lentos (por ejemplo, poner un proceso en la cola de listos, despertar un proceso, etc.). Esto queda justificado por la ventaja que significa contar con un mecanismo de planificación ecuánime y un lenguaje de control enriquecido, lo que en definitiva contruibuye a la declaratividad y expresividad del lenguaje.

# 6 Referencias

[1] Steve Gregory. Parallel Logic Programming in PARLOG. The Language and its Implementation. Addison-Wesley publishing company. 1987.

[2] C. A.R. Hoare. Communicating Sequential Processes. Communications of the ACM. August 1978, Vol. 21, No 8.

[3] Ulf Nilsson and Jan Matuszyński. Logic Programming and Prolog. John Wiley & Sons. 1990.

[4] G. A. Ringwood. Pattern-Directed, Markovian, linear, guarded definite clause resolution. Department of Computing. Imperial College.

[5] Ehud Shapiro. The Family of Concurrent Logic Programming Languages. ACM Computing Surveys, Vol. 21, No.3, September 1989.

[6] Leon Sterling and Ehud Shapiro. The Art of Prolog. Advanced Programming Techniques. The MIT Press. Cambridge, Massachussettss,1987.

[7] THE ARITY/PROLOG LANGUAGE REFERENCE MANUAL. Arity Corporation, Concord,MA,1988.