Creación dinámica de procesos en LINDA

Leonardo Balbiani
Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires
Ciudad Universitaria
Av. Intendente Cantilo s/n
Buenos Aires - Argentina

Mónica Bobrowski
Escuela Superior LatinoAmericana de Informática-ESLAI
CC 3193 (1900)
Buenos Aires - Argentina

Abstract

Procesamiento concurrente, paralelismo, procesamiento distribuido, son términos cada vez más difundidos. Una de las propuestas que aparentemente resulta ser bastante novedosa en este tema, es la de *Comunicación Generativa* y, más específicamente, Linda. El mecanismo de creación dinámica de procesos de Linda no ha sido considerado detalladamente en la bibliografía. El objetivo de este trabajo es analizar dicho mecanismo. Se presenta, en primera instancia, un breve análisis del modelo de *Comunicación Generativa* y de Linda en general, pasando luego a estudiar detalladamente la primitiva para creación de procesos del modelo. A continuación se propone una implementación para la misma tomando en cuenta las consideraciones planteadas al respecto. Se presentan algunos ejemplos sencillos de programas que utilizan dicha facilidad. Finalmente, se presentan las conclusiones del trabajo.

1 Introducción

Procesamiento concurrente, paralelismo, procesamiento distribuido, son términos cada vez más difundidos. Sin embargo, el estudio de estos campos en informática es relativamente reciente y abre un amplio espectro para la investigación y el desarrollo. Día a día se presentan nuevos modelos, nuevos lenguajes, nuevas herramientas que intentan mejorar, entre otras cosas, la expresividad, la elegancia, la sencillez y la eficiencia de las ya existentes. Una de las propuestas que, aparentemente, resulta ser bastante novedosa es la de *Comunicación Generativa* y, más específicamente, Linda. Esta idea fue desarrollada principalmente en la Universidad de Yale, en New Haven, Connecticut, U.S.A., por David Gelernter, Nicholas Carriero y otros.

El modelo de *Comunicación Generativa*, sobre el cual se basa Linda, ha sido ya ampliamente analizado. Lo mismo puede decirse con respecto a las primitivas para comunicación entre procesos de Linda. Sin embargo, el mecanismo de creación dinámica de procesos del modelo no ha recibido un tratamiento similar. Son escasas las referencias existentes en la bibliografía y más escasas aun cuando de implementaciones se trata.

El objetivo de este trabajo es, entonces, analizar dicho mecanismo. Para esto se realizó en primera instancia un breve análisis del modelo de *Comunicación Generativa* y de Linda en general, pasando luego a estudiar detalladamente la primitiva para creación de procesos del modelo. Se diseñó una implementación de esta primitiva sobre el kernel Linda descrito en [2].

La estructura del trabajo es la siguiente: En 2 se describe brevemente el modelo (para una descripción más detallada ver [8]), haciendo hincapié en ciertos problemas que surgen y comentando la implementación descrita en [2]. En 3 se analiza específicamente la primitiva para creación dinámica de procesos de Linda, sobre la cual la bibliografía existente es bastante escasa. Se muestran las ambigüedades de la semántica de esta primitiva y se comentan algunas decisiones tomadas al respecto. En 4 se estudia la implementación de esta primitiva sobre C extendido con el Kernel Linda propuesto en [2]. Se plantean los problemas de esta implementación y se analizan posibles soluciones. En 5 se presentan algunos ejemplos sencillos y bastante generales de programas que crean procesos dinámicamente y finalmente en 6 se presentan las conclusiones.

2 LINDA

Linda es un modelo de programación paralela basado en la idea de *Comunicación Generativa*, la cual unifica las nociones de creación de procesos, comunicación y sincronización. El concepto de *Espacio de Tuplas* es central a este modelo: una región de memoria (lógica) compartida asociativa que puede contener tanto objetos *pasivos* (datos), como procesos *activos* computándose en forma de *tuplas*.

Si bien el modelo forma parte de los conocidos como de *memoria compartida*, posee algunas particularidades dignas de mencionarse en este contexto:

- La unidad de almacenamiento en el espacio de tuplas es la *tupla lógica* -a diferencia de las memorias convencionales en las cuales la unidad de almacenamiento es, por ejemplo, el byte físico. Aquí los elementos son accedidos por *contenido*, i.e., cualquier selección de los valores de la tupla, para lo cual se necesita un mecanismo de *puttern-matching* -en las memorias standard, se accede simplemente por dirección.
- Las operaciones sobre el espacio de tuplas son *leer*, *insertar* y *remover* -sobre una memoria convencional son leer y escribir-, con lo cual las tuplas no pueden ser alteradas en el mismo espacio; deben primeramente ser removidas, luego alteradas y finalmente insertadas en el espacio de tuplas.

- La comunicación entre los procesos se da a través de la inserción y remoción (o lectura) de tuplas. Este tipo de comunicación es *desacoplada en tiempo* y *desacoplada en espacio* . Estas características son propias del modelo de memoria compartida. Se ve que la existencia de las tuplas es independiente de la de los procesos.

Linda consiste de unas pocas simples operaciones sobre el espacio de tuplas. Un lenguaje *base* con el agregado de estas operaciones puede transformarse en un lenguaje de programación paralela.

Las operaciones básicas de Linda sobre el espacio de tuplas son cuatro:

- **out**(t): La tupla t es agregada al espacio de tuplas. Los argumentos de t se evalúan *antes* de que la tupla sea insertada. Esta operación es *asincrónica*, i.e., el proceso que la invocó continúa su ejecución inmediatamente.
- **in**(s): Si existe alguna tupla t en el espacio de tuplas que haga matching exitoso con el *template* s, ésta es removida . Sus parámetros reales son asignados a los respectivos formales de s. Esta operación es bloqueante, i.e., si no existe matching exitoso, el proceso se suspende hasta que aparezca, en cuyo caso prosigue la ejecución del proceso invocante. Si el matching no es único, se elige una tupla de forma no determinística.
- **rd**(s): Esta operación es similar a **in**, sólo que la tupla elegida no es removida del espacio de tuplas.
- **eval**(t): es similar al **out**, pero difiere el momento en que los argumentos son evaluados. La tupla t se agrega al espacio de tuplas en forma *no evaluada*. Esta es la primitiva de creación de procesos de Linda que es objeto del presente trabajo, por lo tanto su descripción detallada se posterga.

Un *matching* entre una tupla y un template es exitoso si:

1. La tupla y el template tienen el mismo número de argumentos.
2. Argumentos correspondientes de la tupla y el template tienen el mismo tipo.
3. Parámetros reales correspondientes son iguales, donde la igualdad es la definida en el lenguaje base para objetos de ese tipo.
4. La tupla y el template no tienen parámetros formales correspondientes.

Nota: Algunas descripciones de Linda agregan como primitivas a **inp** y **rdp**, las cuales se comportan exactamente igual a **in** y **rd** salvo en lo que se refiere a bloqueo: **inp** y **rdp** nunca se bloquean; en lugar de ello devuelven un valor indicando éxito (si hallaron la tupla) o fracaso (en caso contrario). La implementación de estas primitivas se deriva fácilmente de las anteriores y no serán tenidas en cuenta en el presente trabajo. Para más detalles, ver [8].

El control del acceso a cada una de las tuplas del espacio de tuplas se distribuye entre los procesos, lo cual se conoce como *distributed sharing*.

En Linda (en principio) hay un solo espacio de nombres asociado a todos los procesos que conforman un programa. Éstos no tienen necesidad de indicar explícitamente cuál es el espacio de nombres asociado a cada uno. Esto se conoce como *distributed naming*. Según [14], todos los procesos Linda que son parte del mismo programa tienen acceso al mismo espacio de tuplas lógico. En algún sentido, es el acceso a este objeto compartido el que define qué procesos constituyen un programa Linda.

Con este esquema, algunos de los problemas que se presentan son:

- Protección: El espacio de tuplas no prevee la posibilidad de manejar problemas de *alias*, ni tampoco especificar una forma de impedir el acceso de un proceso a ciertas tuplas (el espacio de tuplas es *flat*).
- Duplicación: El espacio de tuplas es un *multiset*, i.e., puede haber tuplas repetidas. No existe un mecanismo para detectar que se está intentando insertar una tupla repetida. Sería deseable poder especificar que cierto conjunto de tuplas tiene la estructura de *set* (i.e., sin duplicaciones) y que sea el sistema el que garantice la consistencia.

Se han presentado propuestas que intentan dar solución a estos y otros problemas utilizando *multiples espacios de tuplas* ([10]). Este nuevo enfoque complica considerablemente el comportamiento de la primitiva de creación de procesos, modificando sustancialmente su semántica y generando nuevos problemas que no parecen tener una solución simple a nivel de implementación. En el presente trabajo se estudia la versión original del **eval**, que es un paso previo indispensable para recién tentar la comprensión de la nueva propuesta.

A continuación se describirá brevemente la implementación de un soporte *run time* de Linda con las primitivas **in**, **out** y **rd** sobre el cual se diseñó una implementación de **eval**. Para más detalles, ver [1], [2].

La 3B Computer Network sobre la cual se trabaja, está constituida por sólo dos nodos host -*elena* y *eslai*. Cada nodo es una minicomputadora 3B 2 con 2.8 MB de memoria central libre que corre UNIX System V extendido con facilidades de networking.

El espacio de tuplas se divide en dos subespacios, uno en cada nodo. Así, cada nodo se encarga de *administrar* su propio espacio debiendo satisfacer todos los pedidos de tuplas que pertenezcan al mismo. El criterio empleado para la división fue partición por función distribuida, i.e., una función que para cada tupla determina unívocamente el subespacio que le corresponde.

El subespacio de tuplas instalado en cada nodo está administrado por un proceso que *vive siempre*, independientemente de la aplicación, llamado *gestor*.

El gestor se encarga de asignar una tupla particular (generada por un **out**) a un *template* (generado por un **in** o **rd**).

Los requerimientos de las aplicaciones son codificados en mensajes y transmitidos a través de la red . Cada mensaje contiene la operación, la tupla y el identificador del proceso que lo envía. El nodo al que se dirige el mensaje es determinado por la función de partición del espacio.

Cada **in** o **rd** puede o no encontrar una tupla que haga matching. En el primer caso, el gestor contesta inmediatamente el mensaje recibido enviando la tupla encontrada. En el segundo, guarda el mensaje para esperar el arribo de la tupla correcta. El conjunto de mensajes bloqueados constituye el estado de la red o estado del sistema.

Cada **out** provoca que el gestor revise los mensajes retenidos en busca de templates que hagan matching con la tupla arribada. Si no hay ninguno, la tupla es instalada en el espacio local. En otro caso, se contestan los mensajes siguiendo alguna estrategia.

El nombre lógico de la tupla (primer campo que siempre es una constante de tipo string) juega dos roles en el sistema de run time: clave para determinar el nodo y clave para ordenar el espacio de búsqueda. La estructura de datos elegida para representar el espacio de tuplas asociado a cada nodo fue un esquema de hashing, usando el nombre lógico como clave. Además, es necesario mantener el estado del sistema -i.e. mensajes retenidos correspondientes a procesos bloqueados por operaciones **in** o **rd** para las que no hay match. Se decidió utilizar el mismo esquema de hashing usado para el espacio de tuplas.

La tabla de hash fue implementada como un arreglo de estructuras de dos campos. El primero es una lista de tuplas y el segundo una lista de mensajes.

La forma interna de una tupla es una estructura que contiene la longitud y un arreglo de componentes. Se impone un límite a la longitud de las tuplas para poder transmitirlas de un nodo a otro con un único paquete de datos. Cada componente es una estructura de 3 campos: el tipo de la componente, si es formal o real y su valor en este último caso.

Una vez construida la representación interna de la tupla, el siguiente paso consiste en enviarla al nodo correspondiente. Para ello se construye un mensaje con el nombre de la puerta de lectura del proceso A , la operación y la tupla. El mensaje es empaquetado en formato ETHERNET y enviado a través de la red. Después el proceso A se queda bloqueado sobre su puerta de lectura esperando que el gestor le conteste. Cuando el gestor recibe el paquete enviado por A, lo decodifica y traduce a un formato interno adecuado. Si en el subespacio correspondiente hay una tupla que hace matching, el gestor arma con ella un paquete (siguiendo el mismo camino descrito antes) y lo transmite por la red con destino a la puerta de lectura del proceso A. Cuando el paquete es recibido, se extrae la tupla y se asignan los valores reales a los correspondientes formales.

En el soporte de run time de C-Linda implementado, el proceso gestor del espacio de tuplas es instalado en el momento de *start-up* del sistema.

Esta implementación es sólo un prototipo. De todas maneras, adolece de ciertas fallas que no siempre están relacionadas directamente con la eficiencia:

- El soporte de comunicaciones no es confiable. i.e., no hay garantías de que los mensajes lleguen a destino.
- El espacio de tuplas es global a todos los programas. Esto puede generar problemas de *aliasing* entre tuplas de distintos programas. Esto implica que no se pueda garantizar el comportamiento de dos o más programas que compartan algún nombre de tuplas.
- Los procesos tienen que ser *cargados* directamente por el usuario en el nodo en el que serán ejecutados. No existe creación dinámica de procesos ni distribución de los mismos desde el programa de aplicación.
- Todos los mensajes van a parar a la red, incluso cuando el destino final de la tupla (o del template) es el mismo nodo en que fue generada.
- El programador está obligado a dar explícitamente la *signatura de tipos* (tipos de los argumentos) de la tupla. Esto es inevitable por tratarse de un soporte en tiempo de ejecución.

Estos problemas condicionan ciertas decisiones del diseño propuesto en 4.

Esto completa la descripción de Linda y del soporte sobre el cual se trabajará.

3 Creación dinámica de procesos en LINDA

A continuación se describirá la evolución de los mecanismos de creación dinámica de procesos en Linda a partir de la bibliografía existente, para luego llegar a la versión que será considerada final en este trabajo, la cual será analizada en detalle.

En [9] se trata la creación dinámica de procesos en Linda de la siguiente manera: "La creación dinámica de procesos es una capacidad de los operadores de comunicación generativa en sí mismos. Brevemente, cualquier expresión libre de efectos colaterales puede aparecer en una tupla generada, y las tuplas son agregadas al espacio de tuplas en forma no evaluada. Tan pronto como la tupla ingresa al espacio de tuplas, la evaluación de todas sus componentes no evaluadas comienza simultáneamente. Luego, la sentencia out(P, f(x)) genera un nuevo proceso; su nombre es "P" y ejecuta f(x). Si f(x) devuelve un entero, la tupla resultante puede ser extraída del espacio de tuplas por medio de la sentencia in(P, i:integer). Si in() se ejecuta mientras f(x) está siendo evaluada, in() se bloquea hasta que se completa la evaluación." Como puede verse, aquí no hay primitivas separadas para crear tuplas de datos y tuplas de procesos. Una tupla que tiene algún argumento no evaluado (por ejemplo, un llamado a una función) es insertada en el espacio de tuplas y da lugar a la creación de un proceso que va a evaluar ese argumento y que se va a ejecutar concurrentemente con el proceso que invocó out(). Si la intención es que los argumentos se evalúen *antes* de que la tupla sea insertada en el espacio de tuplas, la evaluación tiene que ser realizada *localmente* y luego armar la tupla poniendo el resultado de las respectivas evaluaciones como argumentos.

Posteriormente, en [11] se propone una nueva primitiva para creación dinámica de procesos, **eval**(t), con t tupla. La descripción de esta primitiva es la siguiente: "**eval**(t) es similar al **out**(t), salvo que **eval** agrega una tupla no evaluada al espacio de tuplas. La evaluación de t comienza tan pronto como la tupla ingresa al espacio de tuplas; sus campos son evaluados en orden arbitrario y, cuando se completa la evaluación, la tupla resultante puede ser leída usando **rd** o removida usando **in**, tal como cualquier otra tupla. El proceso que se crea implícitamente para realizar la evaluación tiene disponible para sí el valor de cada nombre que aparece en t así como ese nombre fue definido en el ambiente del proceso que ejecuta el **eval** (si este nombre es una función o procedimiento, su valor es una expresión ejecutable). Es un error invocar, dentro de una tupla **eval**, una función cuyo cuerpo hace referencia a variables no locales- sus valores no van a estar disponibles para el proceso que realiza la evaluación.

Por ejemplo: ejecutar

eval("Q", f(x))

provoca que la tupla

(**eval**("Q"), **eval**(f(x)))

sea agregada al espacio de tuplas. La evaluación de "Q" y la de f(x) proceden concurrentemente con el proceso que ejecutó el **eval**. Si f(x) evalúa a un entero j, la *tupla activa* (**eval**("Q"), **eval**(f(x))) se transforma en la *tupla pasiva* ("Q", j) cuando se completa la evaluación. Esta tupla puede ser removida con **in**("Q", int i) o leída con **rd**("Q", int i) como cualquier otra tupla." En el mismo artículo se describe un programa C-Linda: "La ejecución de un programa Linda (C-Linda) siempre comienza con la rutina llamada main() (como en los programas C); main() inicializa cosas, luego crea n *workers* haciendo n veces:

eval(worker())

y después termina." En esta versión se presentan varias novedades en cuanto a la creación dinámica de procesos en Linda que vale la pena comentar:

1. Se provee un mecanismo *explícito* para creación de procesos, diferenciándolo de la inserción de datos en el espacio de tuplas. De aquí en más, se puede entender que los argumentos de una tupla **out**, aún cuando no estén evaluados en el momento de la invocación, serán evaluados antes de armar la tupla que será insertada en el espacio de tuplas. Todos los argumentos de una tupla **eval** son evaluados después de que la tupla ha sido insertada en el espacio de tuplas, aunque en algunos casos esto parezca no tener mucho sentido (por ejemplo, si hay argumentos que son costantes). Esto mantiene la *homogeneidad* del modelo.

2. Se diferencian dos clases de tuplas: *activas* y *pasivas*. Las tuplas activas son argumentos de **eval** que están siendo evaluados. Las tuplas pasivas son generadas por **out** o son el resultado de la evaluación de una tupla **eval**. Aquí surge naturalmente una pregunta: una tupla activa, puede ser removida del espacio de tuplas? De ser así, qué significa remover o leer una tupla activa?

3 Se pueden ver dos niveles de concurrencia. Por un lado, varios **eval** pueden ser ejecutados concurrentemente (por ejemplo, los n workers del main se ejecutan concurrentemente). Por otro lado, los distintos argumentos de un mismo **eval** pueden ser evaluados concurrentemente.

4. Toda tupla activa degenera finalmente en una tupla pasiva. Esto es, todos los procesos creados tienen que producir finalmente algún resultado si terminan (los workers, por ejemplo, podrían ser procesos que no terminen). Si bien esto preserva la homogeneidad del modelo y tiene utilidad a efectos de *sincronización* entre procesos y en casos en que se necesita conocer un resultado, no siempre un proceso produce un valor, o no siempre el valor producido es utilizado. Esto podría conducir a llenar el espacio de tuplas con tuplas que no son necesarias. Un garbage collector tradicional no puede decidir qué tuplas eliminar. Se podría eliminar este requerimiento y permitir que sean los mismos procesos los que generen tuplas pasivas al terminar, si es necesario. Pero esto podría generar problemas de sincronización, además de aumentar el costo de las comunicaciones y restarle elegancia y homogeneidad al modelo.

5. El **eval** introduce la necesidad de chequeos que no son necesarios en el lenguaje host. Por ejemplo las variables a las que pueden acceder los argumentos de un **eval** no siguen las leyes de alcance del lenguaje soporte, por lo tanto se deben añadir nuevos chequeos estáticos al compilador. De aquí se desprende que no alcanza con tener un macro-expansor que traduzca una invocación **eval** en una secuencia de instrucciones en el lenguaje host.

Haciendo un análisis un poco más profundo de la semántica del **out** y la del **eval**, se puede concluir que, en realidad, a nivel del modelo la primitiva **out** no es necesaria. Todo lo que puede ser realizado por un **out** también puede ser realizado por un **eval**. La ejecución *local* asociada al **out** puede obtenerse ejecutando las funciones en el espacio local, asignando sus resultados a variables y luego poniendo estas variables como argumentos de la tupla a ser insertada por medio de **eval** en el espacio de tuplas. Los procesos creados para evaluar dichos argumentos simplemente devolverán el valor de los mismos. Es cierto que hay diferencias de eficiencia: con **out** no es necesario crear procesos para todo, pero la potencia es la misma.

Out de alguna manera funciona como una *macro* que, al expandirse, resulta en lo siguiente:

```
int f();

out(f(x))
```

se traduce en:

int a;

a:= f(x);

eval(a): insertar la tupla (a) en el espacio de tuplas

Con el out propuesto originalmente en [9], que hace la distinción entre los argumentos para los cuales es necesario crear procesos que los evalúen y aquéllos para los cuales no, ni siquiera existen diferencias de eficiencia. Con esta sola primitiva se cubrirían tanto la creación dinámica de procesos como la inserción de tuplas de datos y por ende la comunicación entre procesos. De todas maneras, se ve que la implementación de la idea original no parece ser muy viable: se necesitaría poder reconocer estáticamente qué argumentos necesitan de la creación de un proceso para ser evaluados y cuáles no, lo cual puede no ser sencillo de detectar en tiempo de compilación para todas las tuplas que puedan ser generadas en un programa.

La información sobre eval no abunda en la bibliografía. Más escasa aún es la información sobre implementaciones de eval. Por ejemplo, en [8], donde se comentan alternativas para el diseño de un compilador para Linda, no se menciona el eval. Una de las pocas fuentes de información al respecto es [4]. Allí se comenta:

"eval(t) Es igual al out(t), excepto que t es evaluada después en lugar de antes de ingresar al espacio de tuplas; eval implícitamente hace *fork* de un nuevo proceso para realizar la evaluación.

La operacion eval es el mecanismo de creación de procesos del modelo de programación Linda (los sistemas Linda de memoria compartida emplean fork para crear los nuevos procesos especificados por un eval. Estos procesos son administrados por el sistema operativo multiprocesador y se manejan para ser ejecutados en procesadores libres). La implementación TSnet del eval provee ejecución remota de procesos a través del uso de las operaciones in y out de Linda.

Cada vez que se ejecuta una operación eval, una descripción del eval se inserta en el espacio de tuplas. La descripción incluye identificadores para campos de función y listas de argumentos. Todos los nodos, salvo el nodo maestro, continuamente corren un loop eval que hace in de alguna descripción disponible de eval y ejecuta las funciones especificadas. Una vez que una evaluación particular se completó, la tupla resultante se inserta en el espacio de tuplas y se obtiene una nueva descripción de eval."

Esta implementación es susceptible de ser criticada por varias razones:

- No se comenta cómo hacen los distintos nodos para conocer las funciones que pueden llegar a tener que ejecutar. Este punto es uno de los más dificultosos a la hora de la implementación .
- Desde el momento en que es *levantada* y hasta que se inserta la tupla pasiva correspondiente en el espacio de tuplas, la tupla parece no existir como tal.

- La descripción del **eval** involucra la inserción por parte del sistema de una tupla de nombre lógico "eval" como primera componente. Esto provoca problemas si un usuario inserta o remueve una tupla con ese nombre lógico. Contando solamente con un soporte en tiempo de ejecución esta dificultad no puede ser subsanada. Un compilador podría resolver el problema restringiendo el uso de tuplas con primera componente "eval" por parte del usuario.

- El proceso que *levanta* tuplas **eval**, como usa la primitiva **in** que remueve tuplas por el pattern-matching descrito anteriormente, si no se quiere que dependa del tipo de los argumentos que puede tener la tupla (i.e., que no haya un proceso distinto por cada posible signatura de tipos que pueda haber), debería manejar internamente todos los argumentos como pertenecientes a algún tipo unificado, con lo cual se necesita codificar de alguna manera la tupla como ese tipo antes de insertarla en el espacio de tuplas.

En [7] se describe una implementación de **eval** similar a la de [4]. La única aclaración que agrega es que los descriptores de **eval** contienen punteros a procedimientos, cuyas imágenes están cargadas en todos los nodos e incluyen también los valores de todas las variables mencionadas explícitamente en la invocación a **eval**. No se aclara cómo se obtiene la dirección de los procedimientos en cada nodo ni tampoco cómo y cuándo son cargados en ellos.

En [12] se dice que cuando se encuentra un **eval** se generan, para su evaluación, tantos procesos como argumentos tiene la tupla hallada. También se aclara que la tupla va a ser activa por un tiempo, pero tan pronto como sus argumentos han sido evaluados, la tupla que queda es una tupla pasiva como cualquier otra.

Vale la pena destacar que muchos de los problemas del **eval** no tienen que ver con una implementación particular de Linda, sino con la definición misma del mecanismo de creación dinámica de procesos. Muchos detalles que hacen a su semántica quedaron librados al azar, provocando serios problemas no sólo al intentar una implementación, sino también al intentar programar usando **eval**. Entre otras cosas, no queda claro:

- En qué momento se realiza el *binding* de las variables de las funciones que son argumentos de un **eval**.
- Cuál es el ambiente en que se ejecutan los argumentos de un **eval**.
- Qué cosas puede hacer una función que es argumento de un **eval**.
- Qué clase de parámetros son permitidos en una función que es argumento de un **eval**.
- Qué valores pueden ser resultado de la ejecución de una función que es argumento de un **eval**.

Para diseñar una implementación de **eval** se debe tomar decisiones sobre la interpretación de los puntos anteriormente citados.

4 Alternativas de diseño de EVAL

En esta sección se analizará detalladamente una implementación de la primitiva **eval** sobre el kernel Linda ya descrito en 2. En primer lugar se comentarán cuáles son los aspectos a tener en cuenta al diseñar una tal implementación, los problemas que surgen, las modificaciones y restricciones que deben imponerse al lenguaje *host* (de acuerdo a lo comentado en 3), para luego pasar a explicar cómo se resolvieron dichos inconvenientes en este caso. Cabe destacar que quedan varias cosas en el tintero y que, evidentemente, muchas de las decisiones de diseño tomadas pueden ser mejoradas.

En primer lugar, hay que considerar el equipo sobre el cual se va a trabajar: arquitecturas MIMD con memoria compartida o no, redes de computadoras, etc. Esto, si bien incide en la implementación de un kernel Linda en general, cobra mayor importancia cuando se trata de creación dinámica de procesos, ya que puede condicionar algunas de las decisiones a tomar. En una máquina de memoria compartida, el espacio de tuplas se instala en un segmento de memoria compartido por todos los procesos. Muchos de los problemas que se comentan luego y que están relacionados con la implementación de Linda sobre una red de computadoras no aparecen en ese tipo de máquinas. La principal dificultad reside en que en una red cada nodo tiene su propio espacio de memoria, sus propios periféricos y su propio sistema operativo que puede diferir del de otro nodo. Es importante recordar que la evaluación de un **eval** puede hacerse en un nodo distinto de aquel en el cual fue invocado y que dicho nodo no es conocido por el llamador.

Entre los aspectos importantes a considerar se cuentan:

- Restricciones al lenguaje host: no todas las construcciones del lenguaje host pueden ser usadas en las funciones que son argumentos de un **eval** o como argumentos del mismo (aun si no se consideran las expresiones que tienen efectos colaterales y se prohibe el uso de variables globales en las funciones que son argumentos de un **eval**). Incluso se puede llegar a tener que restringir ciertas cosas en general por no tener sentido semánticamente o por provocar problemas que no permitan garantizar cuál será su comportamiento. Algunos ejemplos que se pueden dar en distintos lenguajes son:
  - Archivos: pueden ser usados?, bajo qué condiciones?, pueden ser argumentos de tuplas?. El uso de archivos dentro de las funciones puede no tener sentido, ya que la única forma de compartir datos que podría permitirse en C-Linda es a través del espacio de tuplas.
  - Variables del tipo static: Su uso puede no tener sentido si las funciones pueden ejecutarse alternativamente en diferentes nodos. Lo mismo sucede con las cosas declaradas como *extern* (en Fortran, por ejemplo, habría que analizar qué sucede con las áreas declaradas *common*).
  - Punteros: Los punteros hacen referencia a una locación de memoria dentro de un determinado nodo. Qué sentido tiene transferir un puntero o hacer referencia a él si no se conserva también información sobre el nodo donde su valor está definido.

Aun el contar con esta información no sería suficiente, ya que se necesitaría contar con algún mecanismo de sincronización de acceso al valor apuntado.

- Fork: Se permite utilizar otro mecanismo de creación dinámica de procesos fuera de **eval**? Tendría sentido darle semántica a, por ejemplo, **eval**(fork())?
- Llamadas al sistema en general: Cuáles son las consecuencias de no conocer el nodo donde serán invocadas ciertas llamadas al sistema?
- Definiciones de tipos: Cómo son conocidas por todas las funciones que pueden ser ejecutadas en nodos no determinados de la red?
- Etc.

Una de las características de las primitivas de Linda más destacada en la bibliografía, es la *ortogonalidad* de las mismas con respecto a cualquier lenguaje *host* dentro del cual pueden ser implementadas. Esto parece ser consecuencia de la falta de consideración que ha recibido el **eval** frente al resto de las primitivas del modelo. Como se ve, Linda no parece ser *ortogonal* al lenguaje host. Los mayores problemas surgen al intentar implementar **eval**.

- **Eval** remoto vs. **eval** local: Se podría implementar un **eval** local, haciendo que las tuplas se evalúen en el mismo nodo en que son generadas, concurrentemente con el proceso que las genera. Por otro lado, se puede pensar en una implementación que distribuya las tuplas a ser evaluadas a lo largo de la red, de acuerdo a diversos criterios.

- Balance de carga: Al implementar un **eval** remoto, puede considerarse el balance de carga del sistema a efectos de determinar la distribución de las tuplas. Se puede considerar (dependiendo de las facilidades provistas por el sistema) el balance de carga *dinámico*, esto es, en tiempo de ejecución tomando en cuenta *todos* los procesos en ejecución en ese instante, o el balance de carga *estático*, esto es, determinar el destino de una tupla activa en tiempo de compilación de acuerdo *sólo* a la carga introducida en el sistema por el programa. También podrían considerarse híbridos.

- Nivel de distribución de los procesos: La distribución de los procesos puede darse a nivel de **eval** (i.e., distintos **eval** se evalúan concurrentemente) o a nivel de argumentos de un mismo **eval** (i.e., los distintos argumentos de un mismo **eval** se evalúan concurrentemente).

- Pasaje de parámetros a los argumentos de un **eval**: Si se implementa un eval remoto, no tiene sentido hablar de pasaje de parámetros por referencia a funciones que son argumentos de un **eval**. En el caso de C, esto se traduce en no poder usar punteros como argumentos de dichas funciones ni tampoco devolver punteros (C tiene sólo pasaje de parámetros por valor). El alcance de esta restricción es mayor, ya que en C los tipos estructurados sólo pueden pasarse por medio de punteros. Se ve que ésta es una seria limitación de **eval**. Notar que el problema aquí no es el hecho de pasar punteros en sí -se podrían pasar los objetos apuntados por ellos- sino cómo chequear que estos valores no han sido modificados, para evitar inconsistencias.

- Cómo se conocen las funciones: Un punto delicado es cómo conocen todos los nodos las funciones que eventualmente pueden llegar a ejecutarse en ellos. En otros lenguajes y mecanismos para procesamiento distribuido (por ejemplo, [6]) las funciones *residen* en determinados nodos y se puede manejar una tabla que diga, para cada nombre de función, dónde se la puede encontrar. Si la información del sistema no está centralizada, se puede guardar una copia de esta tabla en cada nodo y actulizarla mediante algún mecanismo de *broadcast*. En el caso de Linda, de alguna manera las funciones *viajan* junto con las tuplas. No siempre es posible determinar con antelación dónde se va a ejecutar una determinada función. Incluso se puede llegar a ejecutar en diferentes nodos simultáneamente. Este tema es bastante complejo, ya que en alguna medida la eficiencia de la implementación puede depender de él. Una solución sería cargar *a mano* una tabla con todas las funciones que pueden ser llamadas como argumento de eval en todos los nodos de la red. Se ve que esta solución no es la óptima. Más adelante se propone otra alternativa que presupone la existencia de un preprocesador.

El diseño que se detalla a continuación se basa en la implementación de un kernel Linda sobre C que se describe en [3], [1], [2] y que ya fue comentado en 2.

Los tipos permitidos como argumentos de tuplas pasivas son entero, real y string (constantes o variables). El tipo string, que en C es un puntero a caracter, se permite, a pesar de lo comentado anteriormente con respecto a punteros, ya que se asume como constante cuyo valor no se modifica, aunque no se chequea que esto realmente ocurra.

Los tipos permitidos como argumentos de tuplas activas son entero, real, string (constantes o variables) y funciones que reciban argumentos de esos tipos y que devuelvan valores de esos tipos. Con respecto al tipo string, vale el comentario hecho en el caso de tuplas pasivas.

Las tuplas activas y las tuplas pasivas se manejan por separado, esto es, se manejan realmente dos espacios de tuplas. Este punto es transparente al programador. Esta decisión se tomó para facilitar el manejo de las tuplas activas y para lograr un mayor nivel de protección (por ejemplo, evitar que se intente remover o leer tuplas activas). Existe un único gestor para ambos espacios de tuplas, cuyo funcionamiento se describe más adelante. La idea es aprovechar la implementación existente, minimizando las modificaciones. Para ello, se exige que las tuplas activas tengan un *nombre*,esto es, una constante de tipo string que las identifique. Esto permite el posterior armado de la tupla pasiva a insertar y su inclusión en el espacio de tuplas usando la primitiva out ya implementada.

Se debe contar también con la *signatura de tipos* de la tupla activa (ver [3], [1], [2]). En un primer momento, se puede exigir al programador incluirla como uno de los campos de la tupla, para luego contar con un preprocesador que se ocupe de generarla. En el caso de las tuplas activas existen dos alternativas (notar que el problema se presenta porque se deben incluir nombres de funciones y los argumentos con los que serán llamadas): una es incluir en la descripción de una función el número de argumentos además del tipo de la función y tomar los siguientes n campos de la tupla armada (donde n es el número de argumentos de

la función) como argumentos. Otra opción es identificar de manera especial los campos que son argumentos de una función. Vale la pena destacar que este problema no se presenta en el caso de las tuplas pasivas, ya que en éstas lo argumentos se evalúan *antes* de armar la tupla que será insertada.

Por ejemplo, la signatura de tipos de la tupla

eval("tupla", f(x), w, g(y))

(con f, g funciones que toman un entero y devuelven un entero; w,x,y enteros) sería, según la primera alternativa:

"sfi1vivifi1vi"

y según la segunda alternativa:

"sfiavivifiavi"

donde:

- s: constante string
- f: nombre de función
- i: entero
- v: variable
- a:argumento

Notar que, como los argumentos de las funciones de una tupla **eval** sólo pueden ser pasados por valor, en realidad, en el caso de argumentos, no es necesario distinguir si originalmente eran variables o constantes.

A su vez, el espacio de tuplas activas en cada nodo se divide en dos partes: la tabla de tuplas activas en espera (que no han sido *levantadas*) y la tabla de tuplas ejecutándose. La idea de esto es que una tupla activa no puede ser removida del espacio de tuplas activas hasta no haber derivado en una tupla pasiva. O sea, mientras una tupla está siendo evaluada, es necesario tener alguna forma de saber que existe. Si una tupla activa es *levantada* del espacio de tuplas y recién insertada la tupla pasiva correspondiente al finalizar la evaluación, puede haber problemas de consistencia. Mantener información sobre las tuplas que están siendo evaluadas permite eventualmente implementar mecanismos de recuperación de fallas y extender el soporte para considerar, por ejemplo, lectura y remoción de tuplas activas. Notar que la extracción final de la tupla activa del espacio de tuplas activas y la inserción

de la tupla pasiva correspondiente en el espacio de tuplas pasivas debería ser una operación atómica. Sin embargo, si bien esto es cierto a nivel conceptual, no es realmente necesario en la implementación, ya que no se permite la lectura o remoción de tuplas activas y la primitiva in es bloqueante.

La tabla de tuplas activas en espera contiene una entrada por cada tupla activa que ha llegado a ese nodo y que no ha comenzado su evaluación. La tabla de tuplas ejecutándose tiene una entrada por cada tupla que ha comenzado a ser evaluada y guarda sólo la signatura de tipos de la tupla -eventuales extensiones podrían exigir más información. Ambas tablas se guardan en un segmento de *memoria compartida* provisto por Unix, lo que permite que sean compartidas por un padre y sus hijos. La sincronización del acceso a las tablas se da por medio de *semáforos* también provistos por el sistema. Cada tabla tiene un semáforo asociado para garantizar que se acceda a ellas bajo *exclusión mutua*.

En cada nodo existirá un proceso que se encargará de *levantar* tuplas activas y mandarlas a ejecutar. Este proceso (para evitar hacer *busy-waiting*) será activado por medio de un semáforo asociado a él. El gestor del espacio de tuplas será el encargado de despertarlo al arribar una tupla activa, si es que no está despierto. El proceso levantará una tupla activa de la tabla, la mandará a ejecutar y luego buscará más tuplas activas. En caso de hallarse vacía la tabla, se dormirá por medio de su semáforo asociado. En la inicialización del kernel Linda debe incluirse la inicialización de los semáforos a utilizar, además de la creación de tablas, gestores, procesos, etc.

A continuación se describirán, con mayor detalle, los procesos que se añaden al núcleo Linda ya existente y se explicará su funcionamiento.

El gestor de operaciones del nodo origen debe armar la tupla activa a ser enviada, determinar a qué nodo de la red debe *viajar* esta tupla y finalmente enviarla.

Una tupla activa que *viaja* a través de la red contiene:

- la signatura de tipos de la tupla (que puede ser parte de la tupla original o ser armada por el preprocesador).
- la información de los argumentos de la tupla (funciones, argumentos, etc.).
- número de eval asociado (permite obtener, como se verá luego, la información necesaria para evaluar los argumentos de la tupla).
- nodo donde fue compilado el programa del que forma parte (información que permite optimizar el pedido de funciones para ejecutar si no están residentes en el nodo donde se llevará a cabo la evaluación, lo cual hace que no sea necesario guardar todas las funciones en todos los nodos de la red).
- señal de tupla activa (para saber en qué espacio de tuplas debe guardarse).

Para distribuir las tuplas activas entre los nodos de la red, se usa, en esta etapa de la implementación, una función aleatoria que determina un nodo entre los componentes de la

red. Notar que no se utiliza el mismo sistema que para las tuplas pasivas, o sea, hash sobre el *nombre* de la tupla, si bien de todas maneras se exige que las tuplas tengan nombre, como se explicó anteriormente. Lo ideal sería que la elección del nodo se realice en función de otras consideraciones, como por ejemplo, balance de carga dinámico de la red (determinando, en tiempo de ejecución, la carga del sistema), balance de carga estático de la red (determinando, en tiempo de compilación, la carga que va a imponer el programa al sistema y asignando en ese momento nodos a las tuplas activas), etc. Esto tiene que ver con que la distribución de procesos se da a nivel de *tuplas* y no a nivel de *argumentos* de las tuplas (en cuyo caso habría que hacer otro tipo de consideraciones), o sea, se crea un proceso, en algún nodo, para evaluar una tupla eval completa.

El gestor del espacio de tuplas de un nodo debe ser:

```
Gestor()
{
Leer de la red
Si es tupla pasiva
  entonces como antes
si no
  P(sem-tabla-tuplas-activas)
  poner tupla en tabla de tuplas activas
  V(sem-tabla-tuplas-activas)
  Si No-despierto(levanta-eval)
     entonces V(sem-levanta-eval)
}
```

donde P y V son las operaciones usuales sobre semáforos.

El proceso *levanta-eval* debe ser:

```
Levanta-eval()
{
loop
  P(sem-tabla-tuplas-activas)
  Si tabla vacia
    entonces { V(sem-tabla-tuplas-activas)
               P(sem-levanta-eval) }
  Sacar tupla activa
  V(sem-tabla-tuplas-activas)
  Pedir informacion para ejecucion
  P(sem-tabla-tuplas-ejec)
  Poner tupla en tabla de tuplas ejec
  V(sem-tabla-tuplas-ejec)
  Ejecutar tupla
```

```
forever
}
```

Pedir información para ejecución es testear si el archivo que será ejecutado para evaluar la tupla -el cual se determina a partir del número asociado al eval que contiene la tupla activa y que es generado por el preprocesador- se encuentra en el nodo actual. De no ser así, se envía un mensaje al nodo donde fue compilado el programa (información que se encuentra en la tupla activa), para que mande una *copia* de ese archivo. El archivo no se transfiere porque se pueden necesitar varias instancias del mismo al mismo tiempo en diferentes nodos.

Ejecutar tupla va a crear un proceso hijo que se va a encargar de evaluar los argumentos. Entonces, se tiene:

```
Ejecutar tupla()
{
Fork()
si es el padre
   entonces mandar argumentos al hijo
si no, ejecutar archivo asociado.
}
```

El *archivo asociado* es creado por el preprocesador. Se crea un archivo para cada tupla eval cuyo nombre queda unívocamente determinado a partir del número asociado al eval, que también es generado por el preprocesador. Este archivo contiene el código de las funciones que son argumentos de dicho eval y de todas las que pueden ser invocadas por ellas. Recibe los argumentos de la tupla y genera los llamados a funciones necesarios para poder evaluarlos. Tiene variables declaradas para almacenar los resultados de la evaluación de los argumentos y armar posteriormente la tupla pasiva. Tiene punteros declarados a todos los tipos de argumentos que pueden recibir las funciones, con lo cual la cantidad puede ser determinada en tiempo de ejecución. Tiene acceso a las primitivas Linda. Finalmente, arma la tupla pasiva a insertar, remueve la entrada correpondiente en la tabla de tuplas ejecutándose e inserta la tupla pasiva en el espacio de tuplas, por medio de la primitiva out.

La implementación de eval descrita es un prototipo y claramente puede ser mejorada. La simplicidad guió muchas de las decisiones tomadas. Sin embargo, a pesar de no buscar eficiencia se ve la necesidad de imponer restricciones al lenguaje host, en cuanto al tipo de argumentos que puede tener un eval y en cuanto al tipo de argumentos de los argumentos de un eval. Por ejemplo, los parámetros sólo pueden ser pasados por valor (pasar punteros a lo largo de la red no tiene sentido por los problemas de chequeos de modificaciones comentados anteriormente), lo cual impide en el caso de C usar tipos estructurados. También esto restringe las funciones que pueden ser argumento de un eval. Los tipos definidos por el usuario pueden traer dificultades en el caso de eval (si no se incluyen en el archivo generado). Por otro lado, queda clara la necesidad de contar con un preprocesador, no sólo para realizar los chequeos que son propios del eval, sino también para poder garantizar su ejecución.

5 Algunos ejemplos con EVAL

A continuación se presentan algunos ejemplos de programas simples en C-Linda que usan eval.

- Un ejemplo clásico es el ya presentado en 3 para manejar *workers* en una cola de tareas. Cada uno de ellos es generado por medio de un **eval**.

- Otro ejemplo interesante tiene que ver con funciones recursivas. Un caso clásico es el cálculo del factorial de un número natural que, si bien el método que se propondrá a continuación para resolverlo no difiere mucho del tradicional, permite plantear una forma de encarar cierta clase de problemas que involucra algún grado de paralelismo.

  La idea es reemplazar la llamada recursiva por un **eval** de la función. Esto genera un nuevo proceso que ejecuta una nueva instancia de la función, eventualmente en paralelo con la que ya está ejecutándose. Si las funciones sólo usan los argumentos que reciben como parámetros y variables declaradas localmente, hay ciertos cálculos (para los cuales no se requiere el resultado de la recursión anterior) que podrían ser realizados sin necesidad de esperarlos. La primitiva **in** provee la sincronización necesaria. Es interesante analizar la relación de este método con *data-flow*. Las funciones se van ejecutando a medida que los argumentos que necesitan están disponibles.

  Para el caso del factorial, se podría tener:

```
                proglinda:
main()
{
int n;
eval(fac(6));
in(''fac'', 6, n);
...
}

int fac(n)
int n;
{
int aux;
if (n == 0)
   out(''fac'',0, 1);
else
   {
   if n == 1
     out(''fac'', 1, 1);
   else
     {
```

```
            eval(fac(n-1));
        \* cualquier cosa que no requiera
            el resultado de la llamada anterior *\
        in(''fac'', n-1, aux);
        out(''fac'', n, n*aux);
        }
    }
}
            finlinda.
```

- Otro ejemplo del uso de **eval** es el caso de una iteración (tipo *for*) en que se ejecuta una función para cada valor del índice de iteración y las distintas ejecuciones son independientes. Entonces, se podría tener:

```
for i = 1 to n do
eval(f(i));
```

Con esto, las distintas llamadas a la función podrían ejecutarse en paralelo.

6 Conclusiones

- Se describió el mecanismo de creación dinámica de procesos de Linda.
- Se detalló por qué se creee que Linda no es ortogonal a cualquier lenguaje de programación.
- Se comentaron diversos problemas a nivel teórico y a nivel de implementación que surgen al considerar la primitiva **eval** en Linda.
- Se analizó una posible implementación de **eval** sobre C extendido con las primitivas **in**, **out** y **read**.
- Se presentaron algunos ejemplos de programas que utilizan **eval**.

References

[1] R. Alvez and S. Yovine. Alternativas de diseño de soportes en tiempo de ejecución para Linda. Informe de pasantía. ESLAI, 1989.

[2] R. Alvez and S. Yovine. Ampliación de servicios de la red Eslai. Informe de pasantía. ESLAI, 1989.

[3] R. Alvez and S. Yovine. Linda: un modelo de memoria compartida. Informe de pasantía. ESLAI, 1989.

[4] M. Arango and D. J. Berndt. Tsnet: A Linda implementation for networks of unix-based computers. Yaleu/dcs/rr-739. Yale University, August 1989.

[5] H. Bal, J. Steiner, and A. Tanenbaum. Programming languages for distributed computing systems. *ACM Computing Surveys*, 21(3):261–322, September 1989.

[6] A. Birrel and B. Nelson. Implementing remote procedure calls. *ACM Transactions on Computer Systems*, 2(1):39–59, Feb 1984.

[7] R. Bjornson, N. Carriero, and D. Gelernter. The implementation and performance of hypercube Linda. R. r. yaleu/dcs/rr-690, Yale University, March 1989.

[8] N. Carriero. Implementation of tuple space machines. Res. rep. yaleu/dcs/rr-567. Yale University, Dec 1987.

[9] D. Gelernter. Generative communication in Linda. *ACM TOPLAS*, 17(1):80–112. Jan 1985.

[10] D. Gelernter. Multiple tuple spaces in Linda. In *PARLE 89*, March 1989. Invited paper.

[11] D. Gelernter, N. Carriero, S. Chandran, and S. Chang. Parallel programming in Linda. In *Proc. int. Conf. Parallel Processing*, August 1985.

[12] S. C. Hupfer. Melinda: Linda with multiple tuple spaces. Technical report, Yale University, September 1989.

[13] A. Tanenbaum and R Van Renesse. Distributed operating systems. *Computing Surveys*, 17(4):419–470, Dec 1985.

[14] R. Whiteside and J. Leichter. Using Linda for supercomputing on a local area network. Yaleu/dcs/tr-638, Yale University, June 1988.