2

UNIVERSIDAD NACIONAL AUTONOMA DE MEXICO

FACULTAD DE CIENCIAS

SEMANTICA DE LENGUAJES PARALELOS DE ORDEN SUPERIOR. EL CASO DE GAMMA.

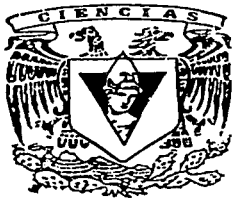
T E S I S

QUE PARA OBTENER EL TITULO DE:

LICENCIADO EN CIENCIAS DE LA COMPUTACION

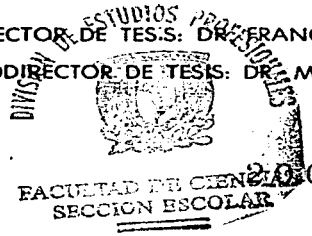
P R E S E N T A :

GUSTAVO DE LA CRUZ MARTINEZ



DIRECTOR DE TESIS: DR. FRANCISCO HERNANDEZ QUIROZ

CODIRECTOR DE TESIS: DR. MANUEL ROMERO SALCEDO



2002

VNIVERSIDAD NACIONAL
AVFNOMA DE
MEXICO

M. EN C. ELENA DE OTEYZA DE OTEYZA
Jefa de la División de Estudios Profesionales de la
Facultad de Ciencias
Presente

Comunicamos a usted que hemos revisado el trabajo escrito:

"Semántica de lenguajes paralelos de Orden Superior. El caso de Gamma"

realizado por DE LA CRUZ MARTINEZ GUSTAVO

con número de cuenta 09438450-3 , quién cubrió los créditos de la carrera de C. DE LA COMPUTACION

Dicho trabajo cuenta con nuestro voto aprobatorio.

Atentamente

Director de Tesis Propietario	DR. FRANCISCO HERNÁNDEZ QUIROZ	
Propietario	DR. MANUEL ROMERO SALCEDO	
Propietario	MTRO. MIGUEL CARRILLO BARAJAS	
Suplente	M. EN C. MARIA DE LA LUZ GASCA SOTO	
Suplente	DRA. HANNA OKTABA	

Consejo Departamental de MATEMATICAS

DRA. AMPARO LOPEZ GAONA
FACULTAD DE CIENCIAS
CONSEJO DEPARTAMENTAL
DE
MATEMATICAS

Agradecimientos

En primer lugar agradezco a Dios, a mis padres y mis hermanos: Roberto, Cecilia, Martha, Jesús, Karen y Heriberto, ya que sin su apoyo no hubiera podido realizar mis estudios.

Agradezco al Dr. Francisco Hernández por aceptarme como coloborador en su proyecto de investigación, de donde se derivó esta tesis.

También doy gracias a mis profesores de licenciatura que me han ayudado a lo largo de mi carrera, en particular al Dr. Fernando Gamboa, quien siempre me ha apoyado e impulsado.

De la misma manera, me siento agradecido con todos mis amigos por la amistad que me han brindado, en especial a (en estricto orden alfabético): Canek, Citlali, Edgar, Egar, Erick, Federico, Karina, Liliana, Oscar, Lucio, Rafael, Ricardo y Yazmín.

Y por último gracias al Instituto de Investigaciones en Matemáticas Aplicadas y en Sistemas por permitirme trabajar en sus instalaciones durante el desarrollo de la presente; así como al Sistema de transporte colectivo "metro" pues sin sus servicios el translado a la universidad hubiera sido muy difícil.

Índice general

Capítulo 1

Introducción

1.1. Lenguajes de programación y semántica

Esta tesis se desarrolla alrededor de la semántica de los lenguajes de programación, de tal forma que es conveniente tener una noción de la semántica en general y en particular en el tema que nos ocupa, los lenguajes de programación.

La semántica es un término utilizado en el estudio de los lenguajes. La semántica de un lenguaje nos define el significado de las cadenas de símbolos que pertenecen al lenguaje, así que en el estudio de los lenguajes también es importante el poder decidir qué cadenas de símbolos pertenecen o no a él. La pertenencia o no de una cadena de símbolos al lenguaje la define su sintaxis.

En los lenguajes de programación, su sintaxis está definida mediante reglas que indican cómo una secuencia de elementos básicos, llamados *palabras*, forman *oraciones* que pertenecen al lenguaje. Las *palabras* generalmente no son átomos, es decir, están construidos a partir de caracteres dentro de un alfabeto. Así, la sintaxis de un lenguaje está definida por dos conjuntos de reglas: *reglas léxicas* y *reglas sintácticas*. Las reglas léxicas especifican el conjunto de caracteres que constituyen el alfabeto del lenguaje y la forma en que los caracteres son combinados para formar palabras válidas. Las reglas sintácticas indican como las palabras válidas se combinan para formar las oraciones del lenguaje.

Las reglas léxicas son de especial importancia en la implementación del lenguaje, pues es la base del proceso de traducción de las oraciones a instrucciones para cada máquina en particular; como en este trabajo no se considera la implementación de los lenguajes que se estudian, en general, al hablar de sintaxis nos referiremos a las reglas sintácticas que definen a cada lenguaje.

Una vez definida la sintaxis de un lenguaje de programación, debemos especificar el significado de cada una de las oraciones del lenguaje, es decir, indicar de que forma una oración modificará el ambiente del programa. Esta descripción la podemos hacer de manera informal indicando como debería cambiar el estado, o de manera formal, utilizando una descripción abstracta del estado o contenido del programa, mediante reglas especificar la manera en que éste cambiará con la ejecución del programa.

La sintaxis y la semántica de un lenguaje de programación tienen un impacto directo en el desarrollo de programas, pues un lenguaje cuya sintaxis o semántica sea muy confusa, o no estén relacionados de forma lógica (que una instrucción no realice lo que normalmente se esperaría), no será bien recibido por los programadores y el lenguaje pasará desapercibido aunque tenga muchas ventajas respecto a otros lenguajes más usados.

En el diseño de lenguajes de programación se han seguido diversas ideas respecto a la forma de representar las estructuras de datos, la manera en que se ejecutan las instrucciones o el modelo abstracto en el que se basa el lenguaje. Esto ha propiciado la aparición de los llamados paradigmas de programación (los cuales definen propiedades generales de los lenguajes), pero sin importar el paradigma que se utilice, una idea que ha prevalecido es la de poder modificar el programa en tiempo de ejecución. Algunos de los lenguajes que permiten estas acciones son denominados de orden superior. Considerar esta característica es importante pues ofrece la posibilidad de tener modelos de programación más dinámicos y también fomentan la reutilización de código. En esta tesis también se abordará la semántica de este tipo lenguajes, en especial el caso de Gamma.

A partir de una semántica de un lenguaje de programación que esté basada en un modelo matemático, podemos entender el comporta-

miento de un programa, así como establecer algunas propiedades que dicho programa cumple o debe cumplir, de esta manera, podremos analizarlo e indicar si realiza o no la tarea para la que fue diseñado. Así, el estudio de la semántica de los lenguajes de programación ,es importante para el análisis y la verificación de programas desde diversos puntos de vista.

Se han realizado diversas investigaciones a cerca de la semántica de los lenguajes de programación. En este caso nos enfocaremos en los trabajos relacionados con las semánticas formales. Winskel ha tratado éste tema a profundidad, en su libro "The Formal Semantics of Programing Languages. An Introduction" [11], donde establece una caracterización de la semántica y establece tres tipos de semántica: *semántica operacional*, la cual describe el significado de un lenguaje de programación a partir de como se ejecutaría en una máquina abstracta; *semántica denotacional*, esta técnica describe el significado de un lenguaje mediante conceptos matemáticos abstractos, como los ordenes parciales, funciones continuas y puntos fijos; y la *semántica axiomática*, ésta intenta establecer el significado de las construcciones del lenguaje en base a reglas y sus demostraciones mediante un lenguaje lógico.

La semántica operacional es muy útil para la implementación de los lenguajes, pues nos da una buena idea del comportamiento esperado del lenguaje, en este caso, utilizaremos ésta semántica para definir el comportamiento de Gamma.

1.2. Paralelismo

La computación paralela se basa en la subdivisión de una tarea computacional en diversas subtareas que se realizan simultáneamente. La subdivisión de tareas se realiza por medio de una combinación de hardware y software específicos. Sin embargo, en la mayor parte de los casos se cuenta con un lenguaje de programación en el que se especifica la división de tareas.

El formalismo Gamma fue propuesto como una respuesta a la evolución de la computación, en especial al surgimiento de modelos no secuenciales, pues captura la idea intuitiva de tener una colección de

elementos que interactúan libremente. Gamma es un lenguaje que puede entenderse con la metáfora de una reacción química. Diversos estudios se han realizado sobre la semántica de Gamma. Bânatre y Le Métayer [1] son algunos de los principales investigadores al respecto. Hernández [7] proporciona una semántica denotacional para Gamma.

La semántica de Gamma tiene sus bases en los sistemas de reducción compuestos. Sands [10] presenta un estudio de estos sistemas. En dicho artículo proporciona entre otras cosas una semántica operacional general para este tipos de sistemas. Esta idea fue extendida para Gamma y algunas de sus versiones como Gamma estructurado y Gamma de orden superior (que es la versión de Gamma adaptada para manejar tipos de orden superior, y es en la que se centra la presente tesis). En esta tesis no se trata el tema de la relación entre los sistemas de reducción compuestos y Gamma. Bermúdez [2] lo aborda con más profundidad.

Una razón de por qué es importante el estudio de Gamma es que en la actualidad los sistemas paralelos y distribuidos están alcanzando una gran importancia, cada vez más sistemas se están diseñando o rediseñando para permitir su colaboración en ambientes distribuidos y paralelos. La concurrencia y la distribución en principio hace que los sistemas sean más eficientes o flexibles, pero así como crece la eficiencia y la flexibilidad también se incrementa la complejidad en el diseño. Gamma está pensado para este tipo de problemas, pues con Gamma es más fácil diseñar programas distribuidos que programas secuenciales. La ventaja de Gamma es que tiene una semántica bien definida y por esta razón el proceso de verificación de los programas es más sencillo.

La presente tesis fue desarrollada como parte del proyecto "Semántica del paralelismo: en busca de una unificación" de Hernández Quiroz. Uno de los principales objetivos del proyecto era que la generalización del modelo semántico de Gamma podría ser la base para un modelo semántico de los sistemas compuestos de reducción y, de este modo, de una subclase de lenguajes paralelos. Además, se esperaba que el mismo modelo pudiera extenderse para abarcar otros lenguajes que no necesariamente pertenecen a este grupo (como Gamma de orden superior). Por esta razón, junto con Bermúdez, se estudió el modelo semántico de Gamma y una variación de él, Gamma estructurado.

Bermúdez en su tesis de "Sistemas de reducción compuestos" [2],

enfoca el estudio de Gamma como un sistema compuesto de reducción, mientras que el trabajo de esta tesis se centra en el estudio de Gamma de orden superior teniendo como base los resultados encontrados para Gamma. Ambas tesis toman como base a Gamma, por lo que una parte importante de ellas es la presentación de los fundamentos necesarios para su estudio como son: la sintaxis y semántica (operacional) de un lenguaje de programación sencillo donde los comando se ejecutan de manera secuencial, la extensión de dicho lenguaje a un nuevo lenguaje tal que los comandos se pueden ejecutar de forma paralela, por supuesto y el estudio de Gamma y Gamma estructurado.

1.3. Organización de la tesis

La tesis está organizada como sigue: el capítulo 2 es una introducción al tema. Ahí se indican algunos motivos de por qué el diseño de los lenguajes se ha dirigido hacia paradigmas en donde se promueva la cooperación entre las entidades individuales.

Para tener una idea más clara de la sintaxis y semántica de los lenguajes primero se presentan éstas para un lenguaje de programación sencillo **IMP** (sección 2.2), la sintaxis del lenguaje será descrita en base a reglas para las distintas expresiones (aritméticas, booleanas y comandos) y una semántica operacional definida a partir de estas reglas. En esta parte se introduce la idea de la transformación de estados mediante la ejecución de los comandos.

Esta misma idea (la transformación entre estados, a partir de un programa) es utilizada para definir un programa determinista.

Ya definido el concepto de programa determinista presentaremos la idea del no determinismo y el paralelismo. La sección 2.3 introduce el tema del no determinismo y las motivaciones que han llevado al desarrollo de esta área. Nuevamente para introducir las bases de los lenguajes de programación paralelos, extendemos el lenguaje **IMP** con una operación de composición paralela. En este parte surgen algunos puntos que deben ser tratados con mayor atención, estos están relacionados con las asignaciones de valores a variables en comandos que se ejecutan en paralelo, pues las asignaciones llevadas a cabo por un programa

pueden afectar el estado en el termina la ejecución de la composición de los programas, dependiendo de cuál asignación se haga primero. De este forma presentamos un nuevo modelo para establecer las relaciones entre los estados y sus comandos.

Se estudian dos modelos básicos para el paralelismo, uno (*comandos resguardados*) basado en el uso de una memoria compartida, y otro (*procesos comunicados*) en un mecanismo de envío de mensajes entre programas para indicar el intercambio de valores. El primero se puede ver como una extensión de IMP. Ambos modelos se basan en el lenguaje propuesto por Dijkstra de comandos resguardados. Estos modelos son tratados más a fondo por Winskel [11].

Un punto central de la tesis es Gamma de orden superior, así que antes de comenzar su análisis se describen brevemente las características de los lenguajes de orden superior en general (sección 2.4), y Gamma en particular, el cual maneja como única estructura de datos a los multiconjuntos. Teniendo un panorama general de los lenguajes de orden superior, los modelos básicos para el paralelismo y el formalismo Gamma, se inicia el estudio de las características de Gamma de orden superior (sección 3.2). Tomando como referencia al trabajo de Le Métayer [8] sobre Gamma de orden superior presentamos una sintaxis y una semántica operacional, finalizando este capítulo con algunos ejemplos de programas en Gamma de orden superior.

Una de las desventajas de Gamma y de Gamma de orden superior es la falta de herramientas para manejar estructuras de datos más complejas que el multiconjunto de manera clara para todos los programadores. Como una solución a este problema, Fradet y Le Métayer [6] proponen una versión de Gamma que permite el manejo de tipos a partir de la definición original de Gamma. Esta versión fue llamada Gamma estructurado. Un primer resultado de esta tesis es el adaptar el modelo de Gamma estructurado a Gamma de orden superior. Mediante un conjunto de reglas para la nueva sintaxis de Gamma de orden superior y tomando el modelo de tipo de Fradet y Le Métayer, obtenemos un modelo de Gamma de orden superior que permite el manejo de datos estructurados.

Finalmente, presentamos un par de aplicaciones de Gamma de orden superior. La primera lo utiliza como un lenguaje para dar una especifi-

cación formal de un algoritmo que únicamente tiene una especificación informal. La otra es un programa que sirve para el problema de la confiabilidad en bases de datos distribuidas.

Para la primera aplicación, tomando como referencia el artículo de Bourgois [3] y obtenemos una especificación formal del algoritmo de replicación de Lotus Notes, desarrollada a partir de la especificación existente para dicho algoritmo. En la parte final de esta sección se hace un análisis de la primera versión del algoritmo y se encuentran algunos puntos no muy bien definidos en la especificación informal pero que son corregidos en la versión formal del algoritmo. Se indican también algunas otras ventajas de las especificaciones formales respecto a las especificaciones informales.

La otra aplicación es un algoritmo en Gamma de orden superior que ayuda en el proceso de *commit* en las bases de datos distribuidas. Para esto, se presentan las bases de datos distribuidas y se llega al algoritmo en pseudocódigo de un protocolo para el manejo del *commit*. Basándose en este algoritmo se obtiene un algoritmo en Gamma de orden superior que realiza las acciones indicadas por el protocolo y que tiene algunas ventajas. Una de ellas es que la nueva versión se puede ejecutar de forma paralela mientras que el otro sigue el paradigma secuencial.

Capítulo 2

Lenguajes de programación y paralelismo

Este capítulo es una introducción a los sistemas paralelos. En él se hablará sobre su lógica y semántica. Para entender fácilmente los conceptos e ideas básicas de los lenguajes de programación paralelos usaremos una extensión de un lenguaje imperativo simple. De esta forma empezaremos hablando brevemente de la semántica tanto de los lenguajes deterministas como de los no deterministas.

2.1. Introducción

La noción de computación secuencial ha tenido un papel central en el diseño de muchos de los lenguajes de programación en el pasado. Esto se ha presentado por dos razones principalmente:

- Los modelos de ejecución secuencial proveen una buena forma de abstracción de los algoritmos.
- Las implementaciones de los programas habían sido sobre arquitecturas de un solo procesador, reflejando el punto de vista secuencial.

Sin embargo el panorama de la ciencia de la computación ha evolucionado considerablemente y las limitaciones del modelo secuencial han sido más y más obvias. Estos cambios han sido causados por el incremento en el tamaño y complejidad de las aplicaciones reales de software

y el progreso en la tecnología del hardware, impactando el desarrollo de esta ciencia de la siguiente forma:

- **La evolución del software** se ha producido como resultado del crecimiento de las necesidades de procesamiento de la información y el decremento del costo del hardware. Ha surgido una innumerable cantidad de sistemas de software, en particular en aspectos que intervienen en nuestra vida diaria (controles remotos para televisores, trenes de control automático, etc). Esto introdujo nuevos problemas como son:
  - Más y más sistemas de software son usados en aplicaciones críticas, como la industria nuclear y el control de aeroplanos, entre otros, de manera que es necesario un alto nivel de confianza en estas aplicaciones. La forma de cumplir los requerimientos es desarrollando software en forma rigurosa basándose en "métodos formales". Un lenguaje formal (Z o B) puede ser usado para especificar el software de forma que no haya ambigüedades y la especificación debe servir como una base para el desarrollo del software. Esto implica que, como lo indica Dijkstra, una "disciplina de programación" es necesaria, la cual debe llevar estos requerimientos a los lenguajes. De la misma forma, los lenguajes de programación deben ser cuidadosamente diseñados para hacer posible el demostrar las propiedades de los programas.
  - El software tiende a crecer en tamaño y complejidad, ya que frecuentemente es desarrollado a lo largo de un gran periodo, lo cual puede provocar que sea muy difícil de entender y mantener. Hoy en día, uno de los principales objetivos en el desarrollo de software es el proveer formas de organización para hacer que las grandes aplicaciones sean manejables, lo mismo que fomentar la reutilización de los productos de software existentes. Diversos lenguajes han sido propuestos recientemente para atacar estos problemas. Estos han sido llamados lenguajes de arquitectura de software o lenguajes de coordinación. Una importante característica de estos lenguajes es el permitir la descripción de las interacciones entre los componentes

individuales, los cuales pueden algunas veces estar escritos en diferentes lenguajes de programación.

- **La evolución del hardware** se ha producido en los últimos años, ya que con el desarrollo de redes electrónicas se ha requerido un rápido progreso en la tecnología de comunicación, haciendo que cada computadora deje de ser considerada un ente aislado, para ser vista como un nodo de una gráfica que representa un sistema distribuido. Por otro lado, cada computadora individual puede tener más de un procesador, con esto el paralelismo se presenta en todos los niveles de la computación.

La situación creada por esta doble evolución ha presentado nuevas necesidades para el diseño de los lenguajes, de forma que la secuencialidad ya no será considerada como el principal paradigma de programación sino que será una forma posible de cooperación entre entidades individuales.

2.2. El lenguaje IMP

En el estudio de los lenguajes de programación hay varios aspectos importantes, entre ellos tenemos la verificación y transformación de programas. En términos generales, la verificación de programas tiene como objetivo la demostración por métodos matemáticos precisos de que un programa es correcto, es decir, que cumple con la tarea para la cual fue diseñado. La transformación de programas, por su parte, crea técnicas para desarrollar versiones más eficientes de un programa dado sin alterar la semántica del programa, pues de este modo se asegura que el programa sigue siendo correcto. Es en esta parte donde la semántica juega un papel importante, pues es el fundamento teórico de ambos aspectos.

Así, para entender la semántica de los lenguajes paralelos comenzaremos primero con una breve descripción de ésta para los lenguajes secuenciales.

En general diremos que la semántica nos define el significado de un programa, es decir, la manera en que asignamos a cada elemento de un

dominio sintáctico (entendiendo como el dominio sintáctico a los programas o componentes de programas, ambos definidos por la sintaxis del lenguaje), a su significado o interpretación, es decir, a un elemento del dominio semántico (como dominio semántico entenderemos alguna estructura algebraica que refleje el contenido del programa). De esta manera para entender el comportamiento de un lenguaje primero debemos hablar de su sintaxis para después estudiar su semántica.

2.2.1. Sintaxis

Para fijar ideas respecto a los lenguajes secuenciales presentaremos la sintaxis de un lenguaje de programación, **IMP**, un lenguaje imperativo pequeño. Este lenguaje básicamente maneja números enteros y valores de verdad y permite la asignación de éstos valores a localidades, así como la ejecución secuencial de comandos. Formalmente, el comportamiento de **IMP** esta descrito por reglas, las cuales especifican cómo son evaluadas sus expresiones y cómo son ejecutados sus comandos. Esta versión de **IMP** fue propuesta por Winskel [11]. Las reglas producen una semántica operacional de **IMP** (la semántica operacional nos describe el significado de un lenguaje de programación a partir de cómo se debe ejecutar en una máquina abstracta).

Los conjuntos sintácticos que maneja **IMP** son:

- números $\mathbb{Z}$, enteros positivos y negativos con el cero,
- valores de verdad $\mathbb{B} = \{\mathbf{verdadero}, \mathbf{falso}\}$,
- Localidades $\mathbf{Loc}$,
- Expresiones Aritméticas $\mathbf{A_{exp}}$,
- Expresiones booleanas $\mathbf{B_{exp}}$,
- Comandos $\mathbf{Com}$.

Para formalizar las reglas de formación de los elementos sintácticos utilizaremos una variante de BNF (Backus-Naur Form) asumiendo que:

- n, m pertenecen a los números $\mathbb{Z}$

- X, Y pertenecen a las localidades **Loc**
- a, a_0, a_1 son expresiones aritméticas de $\mathbf{A_{exp}}$
- b, b_0, b_1 son expresiones booleanas de $\mathbf{B_{exp}}$
- c, c_0, c_1 son comandos de **Com**

Las reglas para las expresiones aritméticas y booleanas, así como para los comandos son:

$\mathbf{A_{exp}}$:

$$a ::= n \mid X \mid (a_0 + a_1) \mid (a_0 - a_1) \mid (a_0 \times a_1)$$

$\mathbf{B_{exp}}$:

$$b ::= \mathbf{verdadero} \mid \mathbf{falso} \mid a_0 = a_1 \mid a_0 \leq a_1 \mid \neg b \mid (b_0 \wedge b_1) \mid (b_0 \vee b_1)$$

Com :

$$c ::= \mathbf{skip} \mid X := a \mid c_0; c_1 \mid \mathbf{if}\ b\ \mathbf{then}\ c_0\ \mathbf{else}\ c_1 \mid \mathbf{while}\ b\ \mathbf{do}\ c$$

donde $:=$ es el operador de asignación y ; es el operador de composición secuencial de comandos.

El comando **skip** no realiza cambio alguno sobre el contenido del programa, el comando $X := a$ realiza la asignación del valor obtenido de la evaluación de la expresión a en la localidad X, $c_0; c_1$ es la ejecución secuencial de los comandos c_0 y c_1, el comando **if** b **then** c_0 **else** c_1 indica que se ejecutará el comando c_0 si la evaluación de b da verdadero, de lo contrario se ejecutará el comando c_1, y el comando **while** b **do** c indica que el comando c se ejecutará mientras la evaluación de b no sea falso.

2.2.2. Semántica

Ahora para formalizar el comportamiento de **IMP** definimos su semántica. Como ya mencionamos, las reglas de la sintaxis anterior producen una semántica operacional, así que debemos seleccionar un

modelo de una máquina abstracta sobre la cual se ejecutarán los comandos de IMP. Para ésto, necesitamos definir *estados* y la *evaluación* de expresiones aritméticas y booleanas, para así llegar a la *ejecución* de comandos.

El conjunto de estados σ consiste de funciones $\sigma : Loc \to N$. $\sigma(X)$ es el valor o contenido de la localidad X en el estado σ.

Considerando la evaluación de una expresión aritmética a en un estado σ, denotado $\langle a, \sigma \rangle$, entonces

$$\langle a, \sigma \rangle \to n$$

indica que la expresión a en el estado σ se evalúa a n. El par $\langle a, \sigma \rangle$ se llama configuración.

De esta forma la evaluación de expresiones aritméticas sigue estas reglas:

Evaluación para números :

$$\langle n, \sigma \rangle \to n$$

Evaluación para localidades :

$$\langle X, \sigma \rangle \to \sigma(X)$$

Evaluación de sumas :

$$\frac{\langle a_0, \sigma \rangle \to n_0 \qquad \langle a_1, \sigma \rangle \to n_1}{\langle a_0 + a_1, \sigma \rangle \to n}$$ donde n es la suma de n_0 y n_1

Evaluación de productos :

$$\frac{\langle a_0 \rangle \to n_0 \qquad \langle a_1, \sigma \rangle \to n_1}{\langle a_0 \times a_1, \sigma \rangle \to n}$$ donde n es el producto de n_0 y n_1

La evaluación de expresiones booleanas esta dada por:

Evaluación de valores de verdad :

$$\langle \textbf{verdadero}, \sigma \rangle \to \textbf{verdadero}$$

$$\langle \textbf{falso}, \sigma \rangle \to \textbf{falso}$$

Evaluación de igualdades :

$$\frac{\langle a_0, \sigma\rangle \to n \qquad \langle a_1, \sigma\rangle \to m}{\langle a_0 = a_1, \sigma\rangle \to \textbf{verdadero}} \quad \text{si } n \text{ y } m \text{ son iguales.}$$

$$\frac{\langle a_0, \sigma\rangle \to n \qquad \langle a_1, \sigma\rangle \to m}{\langle a_0 = a_1, \sigma\rangle \to \textbf{falso}} \quad \text{si } n \text{ y } m \text{ no son iguales.}$$

Evaluación de comparaciones :

$$\frac{\langle a_0, \sigma\rangle \to n \qquad \langle a_1, \sigma\rangle \to m}{\langle a_0 \leq a_1, \sigma\rangle \to \textbf{verdadero}} \quad \text{si } n \text{ es menor o igual que } m$$

$$\frac{\langle a_0, \sigma\rangle \to n \qquad \langle a_1, \sigma\rangle \to m}{\langle a_0 \leq a_1, \sigma\rangle \to \textbf{falso}} \quad \text{si } n \text{ no es menor o igual que } m$$

Evaluación de otras operaciones lógicas :

$$\frac{\langle b, \sigma\rangle \to \textbf{verdadero}}{\langle \neg b, \sigma\rangle \to \textbf{falso}} \qquad \frac{\langle b, \sigma\rangle \to \textbf{falso}}{\langle \neg b, \sigma\rangle \to \textbf{verdadero}}$$

$$\frac{\langle b_0, \sigma\rangle \to t_0 \qquad \langle b_1, \sigma\rangle \to t_1}{\langle b_0 \wedge b_1, \sigma\rangle \to t}$$

donde t es **verdadero** si $t_0 \equiv$ **verdadero**
y $t_1 \equiv$ **verdadero**, y **falso** en otro caso

$$\frac{\langle b_0, \sigma\rangle \to t_0 \qquad \langle b_1, \sigma\rangle \to t_1}{\langle b_0 \vee b_1, \sigma\rangle \to t}$$

donde t es **verdadero** si $t_0 \equiv$ **verdadero**
o $t_1 \equiv$ **verdadero**, y **falso** en otro caso

De esta forma ahora podemos definir la forma en que se ejecutan los comandos. En el estado inicial σ_0 tenemos la propiedad que $\sigma_0(X) = 0$ para todas las localidades. Un par $\langle c, \sigma\rangle$ representa una configuración, en la que se ejecuta el comando c en el estado σ. Definimos entonces la relación

$$\langle c, \sigma\rangle \to \sigma'$$

donde σ' es el estado al que llegamos luego de ejecutar c en el estado σ

Entonces las reglas para los comandos son:

Comandos atómicos :

$$\langle \mathbf{skip}, \sigma \rangle \to \sigma$$

$$\frac{\langle a, \sigma \rangle \to m}{\langle X := a, \sigma \rangle \to \sigma[m/X]}$$

donde σ es un estado, $a \in A_{exp}$, $m \in \mathbb{Z}$ y $X \in \mathbf{Loc}$ entonces $\sigma[m/X]$ es el estado obtenido a partir de σ reemplazando el contenido de X por m, es decir,

$$\sigma[m/X](y) = \begin{cases} m & \text{si } Y = X, \\ \sigma(Y) & \text{si } Y \neq X \end{cases}$$

Secuenciales :

$$\frac{\langle c_0, \sigma \rangle \to \sigma'' \qquad \langle c_1, \sigma'' \rangle \to \sigma'}{\langle c_0; c_1, \sigma \rangle \to \sigma'}$$

Condicionales :

$$\frac{\langle b, \sigma \rangle \to \mathbf{verdadero} \qquad \langle c_0, \sigma \rangle \to \sigma'}{\langle \mathbf{if}\ b\ \mathbf{then}\ c_0\ \mathbf{else}\ c_1, \sigma \rangle \to \sigma'}$$

$$\frac{\langle b, \sigma \rangle \to \mathbf{falso} \qquad \langle c_1, \sigma \rangle \to \sigma'}{\langle \mathbf{if}\ b\ \mathbf{then}\ c_0\ \mathbf{else}\ c_1, \sigma \rangle \to \sigma'}$$

Ciclos :

$$\frac{\langle b, \sigma \rangle \to \mathbf{falso}}{\langle \mathbf{while}\ b\ \mathbf{do}\ c, \sigma \rangle \to \sigma}$$

$$\frac{\langle b, \sigma \rangle \to \mathbf{verdadero} \quad \langle c, \sigma \rangle \to \sigma'' \quad \langle \mathbf{while}\ b\ \mathbf{do}\ c, \sigma'' \rangle \to \sigma'}{\langle \mathbf{while}\ b\ \mathbf{do}\ c, \sigma \rangle \to \sigma'}$$

De esta manera tenemos descrito el comportamiento de un lenguaje secuencial sencillo y podemos ahora definir la idea de un programa determinista. Diremos que un programa p, es determinista si y solo si:

$$\forall \sigma \text{ estado}, \quad (\langle p, \sigma \rangle \to \sigma' \ \text{ y } \ \langle p, \sigma \rangle \to \sigma'') \Rightarrow \sigma = \sigma''$$

2.3. No Determinismo y Paralelismo

Con la aparición en los últimos años de una nueva generación de máquinas paralelas caracterizadas por el gran número de procesadores, como la "Connection Machine", ha surgido la necesidad de diseñar nuevos lenguajes de programación que exploten las características de dichas máquinas. Desde el punto de vista del programador, es imposible que él maneje mentalmente los detalles de la descomposición de los programas en un gran número de tareas individuales, por lo que requiere de un lenguaje de programación de alto nivel en el cual el paralelismo sea implícito y tenga un compilador el cual sea capaz de llevar éste a una arquitectura paralela.

El determinismo quiere decir que, activando un programa en cierto estado generaremos exactamente una secuencia de cómputo. Sin embargo este nivel de detalle es innecesario, por ejemplo cuando dos diferentes secuencias de cómputo llegan al mismo estado final. El fenómeno en el cual un programa puede generar más de una secuencia de cómputo a partir de un estado dado es llamado no determinismo.

Una forma simple para introducir algunas ideas básicas de los lenguajes de programación paralela es extendiendo el lenguaje imperativo simple IMP con una operación de composición paralela. Para los comandos los comandos c_0 y c_1 su composición paralela, denotada por $c_0||c_1$, ejecuta c_0 o c_1 sin preferencia por alguno de los comandos. Un problema se presenta cuando c_0 y c_1 asignan un valor a la misma variable, uno llevará a cabo la asignación, posiblemente seguido por el otro. Esto plantea que la asignación llevada a cabo por un programa puede afectar el estado en el que termina la ejecución de la composición de los programas, dependiendo de cual asignación se haga primero. De esta manera no podemos describir correctamente el modelo de ejecución de los comandos en paralelo usando una relación entre configuraciones de comandos y estados finales. Esta relación la debemos entender como la representación de pasos ininterrumpidos de una ejecución y que permite a los comandos afectar el estado de otros con los cuales está trabajando en paralelo.

Un paso ininterrumpido está determinado por las reglas de ejecución de los comandos y la evaluación de las expresiones. Las reglas para la

evaluación paralela son:

$$\frac{\langle c_0, \sigma\rangle \rightarrow_1 \sigma'}{\langle c_0 || c_1, \sigma\rangle \rightarrow_1 \langle c_1, \sigma'\rangle} \qquad \frac{\langle c_0, \sigma\rangle \rightarrow_1 \langle c_0', \sigma'\rangle}{\langle c_0 || c_1, \sigma\rangle \rightarrow_1 \langle c_0' || c_1, \sigma'\rangle}$$

$$\frac{\langle c_1, \sigma\rangle \rightarrow_1 \sigma'}{\langle c_0 || c_1, \sigma\rangle \rightarrow_1 \langle c_0, \sigma'\rangle} \qquad \frac{\langle c_1, \sigma\rangle \rightarrow_1 \langle c_1', \sigma'\rangle}{\langle c_0 || c_1, \sigma\rangle \rightarrow_1 \langle c_0 || c_1', \sigma'\rangle}$$

donde c_0', c_1' son los comandos obtenidos después de ejecutar c_0, c_1 respectivamente.

La simetría en las reglas para la composición paralela introduce una impredecibilidad en el comportamiento de los comandos, ya que para un comando $c_0||c_1$ no podemos determinar cual de los comandos (c_0 o c_1) se ejecutará primero. Esta impredecibilidad es llamada no determinismo.

Existen dos modelos básicos para el paralelismo, uno se basa en el uso de una memoria compartida y lo podemos ver como una extensión de IMP, y el otro utiliza un mecanismo de envío de mensajes entre los programas para indicar el intercambio de valores. Un lenguaje que se basa en el primer modelo es el de *comandos resguardados* y para el segundo modelo tenemos los *procesos comunicados.*

2.3.1. Comandos resguardados

El uso disciplinado del no determinismo puede mejorar la presentación de los algoritmos, ya que la realización de un objetivo puede no depender del orden en que las diversas tareas se realicen. Dijkstra propone un lenguaje basado en comandos resguardados (guarded commands) para ayudar al programador en la especificación de los programas. El lenguaje de Dijkstra tiene expresiones aritméticas y booleanas

$$a \in \mathbf{A_{exp}} \ y \ b \in \mathbf{B_{exp}}$$

las cuales son las mismas que para IMP, pero ahora se agruparán en dos conjuntos sintácticos nuevos, los comandos (denotados con c) y los comandos resguardados (denotados por gc). Así la sintaxis abstracta esta dada por las reglas:

Comandos :

$$c ::= \textbf{skip} \mid \textbf{abort} \mid X := a \mid c_0; c_1 \mid \textbf{if } gc \textbf{ fi} \mid \textbf{do } gc \textbf{ od}$$

Comandos resguardados :

$$gc ::= b \rightarrow c \mid gc_0 \;[\!]\; gc_1$$

Un comando resguardado generalmente tiene la forma:

$$(b_1 \rightarrow c_1) \;[\!]\; \ldots \;[\!]\; (b_n \rightarrow c_n)$$

En este contexto las expresiones booleanas son llamadas *guardias*, ya que la ejecución del comando c_1 depende de que el correspondiente guardia b_1 sea verdadero. Si la evaluación de la guardia no da verdadero para un estado, el comando resguardado indica falla, en cuyo caso dicho comando no llega a un estado final. De lo contrario ejecuta no deterministamente uno de los comandos c_i el cual esta asociado al guardia b_i que se evaluó como verdadero. El comando **abort** no nos lleva a algún estado final. El comando **if** gc **fi** ejecuta el comando resguardado gc. El comando **do** gc **od** ejecuta repetidamente el comando gc mientras gc no falle.

Para las reglas de ejecución de los comandos y comandos resguardados, se utilizarán las relaciones de evaluación para $\mathbf{A_{exp}}$y $\mathbf{B_{exp}}$ de **IMP**. Una configuración de un comando tiene la forma $\langle c, \sigma \rangle$ para un comando c y un estado σ.

Las configuraciones iniciales para los comandos resguardados son pares $\langle gc, \sigma \rangle$, para comandos resguardados gc y estados σ, solo que en este caso un paso en su ejecución puede ser una configuración de comandos o una nueva configuración llamada **fail**. Así las reglas para la ejecución son:

Reglas para comandos :

$$\langle \textbf{skip}, \sigma \rangle \rightarrow \sigma$$

$$\frac{\langle a, \sigma \rangle \rightarrow m}{\langle X := a, \sigma \rangle \rightarrow \sigma[m/X]}$$

$$\frac{\langle c_0, \sigma \rangle \rightarrow \sigma'}{\langle c_0; c_1, \sigma \rangle \rightarrow \langle c_1, \sigma' \rangle} \qquad \frac{\langle c_0, \sigma \rangle \rightarrow \langle c_0', \sigma' \rangle}{\langle c_0; c_1, \sigma \rangle \rightarrow \langle c_0'; c_1, \sigma' \rangle}$$

$$\frac{\langle gc, \sigma \rangle \rightarrow \langle c, \sigma' \rangle}{\langle \textbf{if } gc \textbf{ fi}, \sigma \rangle \rightarrow \langle c, \sigma' \rangle}$$

$$\frac{\langle gc, \sigma \rangle \rightarrow \textbf{fail}}{\langle \textbf{do } gc \textbf{ od}, \sigma \rangle \rightarrow \sigma} \qquad \frac{\langle gc, \sigma \rangle \rightarrow \langle c, \sigma' \rangle}{\langle \textbf{do } gc \textbf{ od}, \sigma \rangle \rightarrow \langle c; \textbf{do } gc \textbf{ od}, \sigma' \rangle}$$

Reglas para comandos resguardados :

$$\frac{\langle b, \sigma \rangle \rightarrow \textbf{verdadero}}{\langle b \rightarrow c, \sigma \rangle \rightarrow \langle c, \sigma \rangle}$$

$$\frac{\langle b, \sigma \rangle \rightarrow \textbf{falso}}{\langle b \rightarrow c, \sigma \rangle \rightarrow \textbf{fail}}$$

$$\frac{\langle gc_0, \sigma \rangle \rightarrow \langle c, \sigma' \rangle}{\langle gc_0 \parallel gc_1, \sigma \rangle \rightarrow \langle c, \sigma' \rangle} \qquad \frac{\langle gc_1, \sigma \rangle \rightarrow \langle c, \sigma' \rangle}{\langle gc_0 \parallel gc_1, \sigma \rangle \rightarrow \langle c, \sigma' \rangle}$$

$$\frac{\langle gc_0, \sigma \rangle \rightarrow \textbf{fail} \quad \langle gc_1, \sigma \rangle \rightarrow \textbf{fail}}{\langle gc_0 \parallel gc_1, \sigma \rangle \rightarrow \textbf{fail}}$$

La regla alternativa $gc_0 \parallel gc_1$ introduce el no determinismo, ya que un comando resguardado puede ejecutar gc_0 o gc_1 y de esta forma no podemos determinar cuál es la secuencia generada a partir del estado actual.

2.3.2. Procesos comunicados

En la segunda mitad de los 70 Hoare y Milner proponen un modelo distinto al de comandos resguardados, basado en procesos que tienen

la posibilidad de intercambiar valores. Supongamos que un proceso se comunica con otro por medio de canales. Se permite el uso de canales ocultos para que la comunicación por un canal en particular pueda ser recibida por dos o más procesos. Un proceso puede estar preparado para recibir o enviar datos a un canal. Sin embargo esto puede suceder solamente si hay un proceso en su ambiente que realiza la acción complementaria de recibir o enviar. Cuando se realiza el proceso de enviar un valor, éste es copiado del proceso fuente al proceso destino.

Para la sintaxis del lenguaje de procesos comunicados consideraremos un conjunto de localidades $b \in \mathbf{B_{exp}}$ y expresiones aritméticas $a \in \mathbf{A_{exp}}$ y asumimos que:

$$\begin{aligned}&\text{Nombres de los canales } \alpha, \beta, \gamma, \ldots, \in \mathbf{Chan}\\ &\text{Expresiones de entrada } \alpha?X \quad \text{donde} \quad X \in \mathbf{Loc}\\ &\text{Expresiones de salida } \alpha!a \quad \text{donde} \quad a \in \mathbf{A_{exp}}\end{aligned}$$

Comandos:

$$\begin{aligned}c \quad ::= \quad & \mathbf{skip} \mid \mathbf{abort} \mid X := a \mid \alpha?X \mid \alpha!a \mid c_0; c_1 \mid \mathbf{if}\ gc\\ & \mathbf{fi} \mid \mathbf{do}\ gc\ \mathbf{od} \mid c_0 || c_1 \mid c\backslash\alpha\end{aligned}$$

Comandos resguardados:

$$gc \ ::= \ b \ \rightarrow \ c \mid b \wedge \alpha?X \ \rightarrow \ c \mid b \wedge \alpha!a \ \rightarrow \ c \mid gc_0 \ [\!] \ gc_1$$

No todos los comandos y comandos resguardados son bien formados. Una composición paralela $c_0||c_1$ está bien formada en caso de que los comandos c_0 y c_1 no contengan una localidad común. En general un comando es bien formado si todos sus subcomandos de la forma $c_0||c_1$ son bien formados. Una restricción $c\backslash\alpha$ oculta el canal α, de forma que solo comunicaciones internas puedan ocurrir sobre éste.

Para formalizar el comportamiento de este lenguaje de procesos comunicados, los estados serán funciones de localidades a los valores que éstas contienen y una configuración tendrá la forma $\langle c, \sigma \rangle$ o σ con c un comando y σ un estado.

Ahora consideremos la configuración de un comando en particular que tiene la siguiente forma:

$$\langle \alpha?X; c, \sigma \rangle$$

Esta representa un comando que se preparará primero para recibir una comunicación sincronizada de un valor de X a través del canal α. Para que esto se realice es necesario que en paralelo otro comando esté preparado para realizar la acción complementaria de enviar un valor por el canal α. En esta semántica etiquetaremos las transiciones de forma que la etiqueta de una transición indique cual es la acción complementaria para que se lleve a cabo la comunicación con este comando, es decir, para un comando $\alpha?X; c$ la transición estaría etiquetada como $\alpha!n$, y viceversa para el comando complementario. De esta forma el conjunto de etiquetas esta definido como:

$$\{\alpha?n \mid \alpha \in \mathbf{Chan} \ \& \ n \in N\} \cup \{\alpha!n | \alpha \in \mathbf{Chan} \ \& \ n \in N\}$$

La etiqueta

$$\langle \alpha?X; c_0, \sigma \rangle \xrightarrow{\alpha!n} \langle c_0, \sigma[n/X] \rangle$$

representa el hecho que el comando $\alpha?X; c_0$ puede recibir un valor n por el canal α y almacena este en una localidad X y así modifica el estado. Las etiquetas de la forma $\alpha?n$ representan la capacidad de recibir un valor n del canal α. Así necesitamos un comando

$$\langle a!e; c_1, \sigma \rangle \xrightarrow{\alpha!n} \langle c, \sigma \rangle$$

donde $\langle e, \sigma \rangle \rightarrow n$. De esta forma para que halla la posibilidad de comunicación de dos comandos que se ejecuten en paralelo tendríamos

$$\langle (\alpha?X; c_0) || (\alpha!e); c_1, \sigma \rangle \rightarrow \langle c_0 || c_1, \sigma[n/X] \rangle$$

Este lenguaje no incluye la posibilidad de que un comando reciba un valor del ambiente y no de otro comando, y viceversa.

Para la semántica de este lenguaje asumimos que las expresiones aritmética y booleanas tienen la misma forma que en **IMP** e inherentemente las reglas de evaluación para ellas. Los comandos resguardados pueden ser tratados de manera similar a la semántica anterior, pero permitiendo la comunicación en los guardias. Consideramos también el comando vacío $*$ que satisface las reglas:

$$*; c \equiv c; * \equiv *||c \equiv c||* \equiv c \qquad \text{y} \qquad *; * \equiv *||* \equiv *$$

Reglas para comandos :

$$\langle \mathbf{skip}, \sigma \rangle \rightarrow \sigma$$

$$\frac{\langle a, \sigma \rangle \rightarrow n}{\langle X := a, \sigma \rangle \rightarrow \sigma[n/X]}$$

$$\langle \alpha?X, \sigma \rangle \xrightarrow{\alpha?n} \sigma[n/X]$$

$$\frac{\langle a, \sigma \rangle \rightarrow n}{\langle \alpha!a, \sigma \rangle \xrightarrow{\alpha!n} \sigma}$$

$$\frac{\langle c_0, \sigma \rangle \xrightarrow{\lambda} \langle c_0', \sigma' \rangle}{\langle c_0; c_1 \sigma \rangle \xrightarrow{\lambda} \langle c_0'; c_1, \sigma' \rangle}$$

$$\frac{\langle gc, \sigma \rangle \xrightarrow{\lambda} \langle c, \sigma' \rangle}{\langle \mathbf{if}\ gc\ \mathbf{fi}, \sigma \rangle \xrightarrow{\lambda} \langle c, \sigma' \rangle}$$

$$\frac{\langle gc, \sigma \rangle \xrightarrow{\lambda} \langle c, \sigma' \rangle}{\langle \mathbf{do}\ gc\ \mathbf{od}, \sigma \rangle \xrightarrow{\lambda} \langle c; \mathbf{do}\ gc\ \mathbf{od}, \sigma' \rangle} \qquad \frac{\langle gc, \sigma \rangle \rightarrow \mathbf{fail}}{\langle \mathbf{do}\ gc\ \mathbf{od}, \sigma \rangle \rightarrow \sigma}$$

$$\frac{\langle c_0, \sigma \rangle \xrightarrow{\lambda} \langle c_0', \sigma' \rangle}{\langle c_0 || c_1, \sigma \rangle \xrightarrow{\lambda} \langle c_0' || c_1, \sigma' \rangle} \qquad \frac{\langle c_1, \sigma \rangle \xrightarrow{\lambda} \langle c_1', \sigma' \rangle}{\langle c_0 || c_1, \sigma \rangle \xrightarrow{\lambda} \langle c_0 || c_1', \sigma' \rangle}$$

$$\frac{\langle c_0, \sigma \rangle \xrightarrow{\alpha?n} \langle c_0', \sigma \rangle \quad \langle c_1, \sigma \rangle \xrightarrow{\alpha!n} \langle c_1', \sigma \rangle}{\langle c_0 || c_1, \sigma \rangle \rightarrow \langle c_0' || c_1', \sigma' \rangle}$$

$$\frac{\langle c_0, \sigma \rangle \xrightarrow{\alpha!n} \langle c_0', \sigma \rangle \quad \langle c_1, \sigma \rangle \xrightarrow{\alpha?n} \langle c_1', \sigma' \rangle}{\langle c_0 || c_1, \sigma \rangle \rightarrow \langle c_0' || c_1', \sigma' \rangle}$$

$$\frac{\langle c, \sigma \rangle \xrightarrow{\lambda} \langle c', \sigma' \rangle}{\langle c \backslash \alpha, \sigma \rangle \xrightarrow{\lambda} \langle c' \backslash \alpha, \sigma' \rangle}$$

donde $\lambda \equiv \alpha?n$ o $\lambda \equiv \alpha!n$

Reglas para comandos resguardados :

$$\frac{\langle b,\sigma\rangle \to \textbf{verdadero}}{\langle b \to c,\sigma\rangle \to \langle c,\sigma\rangle} \qquad \frac{\langle b,\sigma\rangle \to \textbf{falso}}{\langle b \to c,\sigma\rangle \to \textbf{fail}}$$

$$\frac{\langle b,\sigma\rangle \to \textbf{falso}}{\langle b \wedge \alpha?X \to c,\sigma\rangle \to \textbf{fail}} \qquad \frac{\langle b,\sigma\rangle \to \textbf{falso}}{\langle b \wedge \alpha!a \to c,\sigma\rangle \to \textbf{fail}}$$

$$\frac{\langle gc_0,\sigma\rangle \to \textbf{fail} \quad \langle gc_1,\sigma\rangle \to \textbf{fail}}{\langle gc_0 || gc_1,\sigma\rangle \to \textbf{fail}}$$

$$\frac{\langle b,\sigma\rangle \to \textbf{verdadero}}{\langle b \wedge \alpha?X \to c,\sigma\rangle \xrightarrow{\alpha?n} \langle c,\sigma[n/X]\rangle}$$

$$\frac{\langle b,\sigma\rangle \to \textbf{verdadero} \quad \langle a,\sigma\rangle \to n}{\langle b \wedge \alpha!a \to c,\sigma\rangle \xrightarrow{\alpha!n} \langle c,\sigma\rangle}$$

$$\frac{\langle gc_0,\sigma\rangle \xrightarrow{\lambda} \langle c,\sigma'\rangle}{\langle gc_0 || gc_1,\sigma\rangle \xrightarrow{\lambda} \langle c,\sigma'\rangle} \qquad \frac{\langle gc_1,\sigma\rangle \xrightarrow{\lambda} \langle c,\sigma'\rangle}{\langle gc_0 || gc_1,\sigma\rangle \xrightarrow{\lambda} \langle c,\sigma'\rangle}$$

De esta forma hemos presentado dos lenguajes que se basan en el modelo de Dijkstra de comandos resguardados para introducir el paralelismo. Este enfoque se basa en la idea de considerar la composición paralela como la interacción no determinista de acciones atómicas de los componentes, pero existen otros modelos como las redes de Petri y los modelos de eventos los cuales representan el paralelismo como una forma de independencia entre las acciones, es decir, cada acción se puede ejecutar independientemente, solo que la decisión de cuál se ejecuta se toma apartir de eventos que se van produciendo en el ambiente del programa.

2.4. Lenguajes de Orden Superior

Algunos lenguajes de programación reflejan de manera casi exacta la arquitectura del hardware subyacente (Occam). Otros en cambio, tratan de ser menos dependientes de una arquitectura particular. Tienen como objetivo principal el expresar algoritmos de la forma más general posible

con el fin de facilitar su comprensión, desarrollo y verificación. Uno de estos lenguajes es Gamma.

Los lenguajes de programación manejan distintos tipos de estructuras de datos, algunos manejan únicamente una estructura y debemos tratar de modelar todas las demás en base a dicha estructura. Otros, por el contrario, permiten manejar una gran variedad de estructuras lo que puede ayudar a la comprensión de un programa. Además de las estructuras de datos que manejan los lenguajes de programación, algunos pueden manipular el programa mismo, estos lenguajes son llamados *lenguajes de orden superior.*

Decimos que un lenguaje es de *primer orden* si no considera a los programas como parte de los datos que puede manejar y además el programa nunca puede ser modificado por si mismo o por otro programa.

En los lenguajes de orden superior no se hace distinción entre el programa y las estructuras de datos que maneja, de esta manera, en términos no formales la ejecución de un programa de orden superior toma al programa y las estructuras de datos correspondientes y los va modificando según lo indiquen la semántica del programa.

Los lenguajes de orden superior al igual que los lenguajes de primer orden presentan diversos paradigmas: los lenguajes imperativos de orden superior que extienden a los lenguajes imperativos agregando la posibilidad de tratar a los programas de la misma forma que las estructuras de datos; los lenguajes lógicos de orden superior que se basan en una lógica de orden superior; los lenguajes funcionales de orden superior que permiten que las funciones tomen como parámetros otras funciones.

Por lo tanto, llamamos *tipos de orden superior*, a los programas, funciones o predicados, dependiendo del paradigma de programación. En general podemos decir que un lenguaje de orden superior manejará tipos de orden superior.

Los lenguajes de orden superior son importantes pues aumentan el poder expresivo y funcional de un lenguaje, por ejemplo, podemos diseñar programas más genéricos donde algunas operaciones se definan hasta el momento de la ejecución del programa.

Capítulo 3

Gamma de Orden Superior

3.1. Gamma

El formalismo Gamma fue presentado hace pocos años para permitir la derivación sistemática de programas sin secuencialidad artificial (entendiendo por artificial aquella secuencialidad que no está implícita en la lógica del programa). Gamma fue presentado por Banâtre y Le Métayer en [1]. El modelo de Gamma puede ser descrito como un transformador de multiconjuntos: los cómputos son una sucesión de aplicaciones de reglas las cuales consumen elementos del multiconjunto mientras producen nuevos elementos en el conjunto. Los cómputos terminan cuando ninguna de las reglas puede aplicarse.

3.1.1. Multiconjuntos

De forma sencilla, un multiconjunto es una colección de elementos con repeticiones. Más formalmente, un multiconjunto de elementos de D es una función $M : D \to \mathbb{N}$. Consideraremos únicamente los multiconjuntos finitos, es decir, funciones donde hay solo un número finito de elementos en D con $M(X) > 0$. Usaremos la siguiente notación para multiconjuntos.

$$\{\!|x_1, \ldots, x_n|\!\},$$

donde los x_i no son necesariamente distintos. El conjunto de multiconjuntos finitos de elementos en D será denotado por $\mathbb{M}(D)$. Si tenemos

M y $N \in \mathbb{M}(D)$, definimos las funciones para la unión, diferencia e intersección, respectivamente:

$$\begin{aligned}(M \uplus N)(x) &= M(x) + N(x) \\ (M - N)(x) &= \begin{cases} M(x) - N(x) & \text{si } M(x) \geq N(x) \\ 0 & \text{en otro caso} \end{cases} \\ (M \uplus N)(x) &= \min(M(x), N(x)).\end{aligned}$$

Para la inclusión tenemos que: $M \subseteq N$ si y solo si $M(x) \leq N(x)$ para toda $x \in D$.

El conjunto de booleanos es $\mathbb{B} = \{\textbf{verdadero}, \textbf{falso}\}$. Dado un conjunto arbitrario T, un predicado de cardinalidad n es una función $R^n : T^n \to \mathbb{B}$. El conjunto de predicados de cardinalidad n es $\mathbb{R}^n$ y el conjunto de predicados es

$$\mathbb{R} = \bigcup_{n \in \mathbf{N}} \mathbb{R}^n.$$

Estos son llamados *condiciones de reacción.*

Sean $f_1 : T^n \to T, f_2 : T^n \to T, \ldots, f_m : T^n \to T$ funciones. La función $A^n : T^n \to \mathbb{M}(T)$ definida como:

$$A^n(x_1, \ldots, x_n) = \{\!| f_1(x_1, \ldots, x_n), \ldots, f_m(x_1, \ldots, x_n) |\!\}$$

es llamada una *acción.* La notación $(x_1, \ldots, x_n) \to A^n(x_1, \ldots, x_n)$ se usa también para denotar acciones. $\mathbb{A}^n = \{A^n : T^n \to \mathbb{M}(T)$ es el conjunto de acciones de cardinalidad n. Así que,

$$\mathbb{A} = \bigcup_{n \in \mathbf{N}} \mathbb{A}^n.$$

es el conjunto de acciones.

3.1.2. La metáfora de la reacción química

La única estructura de datos en Gamma es el multiconjunto. Podemos considerar que los programas de Gamma están hechos de *reglas de*

reescritura. Las reglas de reescritura son como *reacciones químicas* que actúan sobre una solución química, es decir, los multiconjuntos.

De esta forma un programa de Gamma verifica si la condición de la reacción se cumple para un multiconjunto y lo transforma de acuerdo a las reglas de la acción. Las reacciones atómicas pueden ser combinadas de forma paralela o secuencial. Formalmente, la sintaxis de un programa Gamma es:

$$P ::= (x_1, \ldots, x_n) \rightarrow A^n(x_1, \ldots, x_n) \Leftarrow R^n(x_1, \ldots, x_n) \mid P_1 \circ P_2 \mid P_1 + P_2,$$

donde R^n es un predicado y A^n es una acción. El conjunto de todos los programas es $\mathbb{G}$.

Un reacción atómica transforma un multiconjunto M de la siguiente manera: toma una tupla de M que satisfaga R^n y la reemplaza por el resultado de aplicar A^n a dicha tupla y la reacción se aplica nuevamente a este multiconjunto. Si en M no hay una tupla que satisfaga la condición el multiconjunto permanece sin cambios y la reacción química finaliza, es decir, el multiconjunto alcanza un estado estable.

$P_2 \circ P_1$ denota la composición secuencial de dos programas, en donde el programa P_2 se aplicará al multiconjunto M si y solo si M no puede reaccionar más con P_1.

$P_1 + P_2$ es la composición paralela de dos programas, donde cualquiera P_1 o P_2 pueden reaccionar con un multiconjunto M en cualquier momento, es decir, puede cada programa tomar una tupla distinta de M y al mismo tiempo aplicar la transformación a M, o bien puede solo uno transformar al multiconjunto a la vez. Para terminar, ambos P_1 y P_2 deben no poder reaccionar más con el multiconjunto. La composición paralela puede presentar dos conductas que aparentemente son distintas pero que en realidad son equivalentes. Esto es tratado más adelante, después de la presentación de la semántica de Gamma.

Entonces podemos considerar la función: $S : \mathbb{A}^n \times \mathbb{R}^n \times \mathbb{M}(T) \rightarrow \mathbb{P}_{fin}(\mathbb{M}(T))$ definida como:

$$S(A^n, R^n, M) = \{N | N = (M - \{\!|x_1, \ldots, x_n|\!\}) \uplus A^n(x_1, \ldots, x_n)\}$$

donde $\{\!|x_1, \ldots, x_n|\!\} \subseteq M$ y $R^n(x_1, \ldots, x_n)$ se cumple. De esta forma cuando una regla de reescritura no puede reaccionar con un multiconjunto M entonces $S(A^n, R^n, M) = \emptyset$. Ahora para generalizar esta definición a cualquier programa de Gamma consideramos

$$S : \mathbb{G} \times \mathbf{M}(\mathbf{T}) \to \mathbb{P}_{fin}(\mathbb{M}(T))$$

y entonces:

$$\begin{aligned} S(P_2 \circ P_1, M) &= \begin{cases} S(P_2, M) & \text{si } S(P_1, M) = \emptyset \\ S(P_1, M) & \text{en otro caso} \end{cases} \\ S(P_1 + P_2, M) &= S(P_1, M) \cup S(P_2, M). \end{aligned}$$

La semántica operacional para Gamma está dada por las siguientes reglas:

$$\frac{\{\!|a_1, \ldots, a_n|\!\} \subseteq M, \quad R(a_1, \ldots, a_n)}{\langle (A \Leftarrow R), M\rangle \to \langle (A \Leftarrow R), (M - \{\!|a_1, \ldots, a_n|\!\}) \uplus A(a_1, \ldots, a_n)\rangle}$$

$$\frac{\neg\exists \{\!|a_1, \ldots, a_n|\!\} \subseteq M . R(a_1, \ldots, a_n)}{\langle (A \Leftarrow R), M\rangle \to M}$$

$$\frac{\langle Q, M\rangle \to M}{\langle P \circ Q, M\rangle \to \langle P, M\rangle} \qquad \frac{\langle Q, M\rangle \to \langle Q', M'\rangle}{\langle P \circ Q, M\rangle \to \langle P \circ Q', M'\rangle}$$

$$\frac{\langle P, M\rangle \to \langle P', M'\rangle}{\langle P + Q, M\rangle \to \langle P' + Q, M'\rangle} \qquad \frac{\langle Q, M\rangle \to \langle Q', M'\rangle}{\langle P + Q, M\rangle \to \langle P + Q', M'\rangle}$$

$$\frac{\langle P, M\rangle \to M \quad \langle Q, M\rangle \to M}{\langle P + Q, M\rangle \to M}$$

Como en los lenguajes anteriores, $\to$ indica un solo paso en la transición. $\to^*$ denota la cerradura transitiva y $\langle P, M\rangle$ es una *configuración*, es decir, un par hecho de un programa P y un multiconjunto M, sobre el cual el programa es aplicado. Estas reglas reflejan la definición anterior de la función S. Las primeras dos corresponden a la posible consecuencia de $S(A \Leftarrow R, M)$. Las siguiente cuatro reglas se refieren a la composición secuencial y paralela también corresponden a S aplicado al mismo tipo de programas. La última expresa los requerimientos

para la terminación de dos componentes paralelos de un programa si el último a terminado.

En esta semántica, presentamos a la composición paralela como una transformación paso a paso, es decir, solo uno de los programas modifica al multiconjunto en cada paso (semántica paso a paso). La selección de cuál programa actúa es no determinista, pero solo uno puede hacer cambios al multiconjunto. Como mencionamos anteriormente, tenemos otra posibilidad: cuando ambos programas modifican al mismo multiconjunto a la vez, solo existe la restricción de que los programas deben tomar elementos ajenos. Chaudron [4] presenta una semántica operacional de Gamma que cubre este último caso (semántica de múltiples pasos) y demuestra que ambas semánticas son equivalentes. De manera informal, el objetivo es demostrar que podemos simular una transición de múltiples pasos por una secuencia de transiciones paso a paso, de aquí obtenemos que la semántica de múltiples pasos puede ser simulada por una semántica de paso a paso, para el otro caso solo es necesario notar que las reglas de inferencia para una semántica paso a paso son un subconjunto de las reglas de una semántica de múltiples pasos.

3.1.3. Ejemplos

Para aclarar más las ideas acerca de Gamma presentaremos algunos pocos ejemplos de programas:

$$max = x, y \rightarrow \{\!|x|\!\} \Leftarrow x \geq y$$

este programa calcula el máximo elemento de un multiconjunto no vacío. Aquí $x \geq y$ especifica la propiedad que deben satisfacer los elementos seleccionados x y y. Estos elementos son reemplazados en el multiconjunto por el valor x. Una característica importante en la definición de este programa es que no se especifica el orden de la evaluación de las comparaciones, de esta forma si varios pares de elementos satisfacen la condición, las comparaciones y los reemplazos pueden ser realizados en paralelo.

El programa max podría realizar la siguiente transformación sobre el multiconjunto $\{\!|3, 3, 5, 8, 1, 2, 3, 9|\!\}$:

$$\langle max, \{\!|3, 3, 5, 8, 1, 2, 3, 9|\!\}\rangle \rightarrow \langle max, \{\!|3, 3, 5, 8, 2, 3, 9|\!\}\rangle$$

en este caso selecciona a los elementos 8 y 1, realiza la comparación y únicamente agrega al multiconjunto el elemento 8. Así podría generar la siguiente sucesión de configuraciones:

$$\begin{aligned}
\langle max, \{\!|3,3,5,8,1,2,3,9|\!\}\rangle &\rightarrow \langle max, \{\!|3,3,5,8,2,3,9|\!\}\rangle \\
&\rightarrow \langle max, \{\!|3,5,8,2,3,9|\!\}\rangle \\
&\rightarrow \langle max, \{\!|3,5,8,2,9|\!\}\rangle \\
&\rightarrow \langle max, \{\!|5,8,2,9|\!\}\rangle \\
&\rightarrow \langle max, \{\!|5,2,9|\!\}\rangle \\
&\rightarrow \langle max, \{\!|5,9|\!\}\rangle \\
&\rightarrow \langle max, \{\!|9|\!\}\rangle \\
&\rightarrow \{\!|9|\!\}
\end{aligned}$$

En el último multiconjunto ($\{\!|9|\!\}$) no es posible tomar dos elemento que cumplan la condición de reacción del programa por lo tanto su ejecución termina, es decir, el multiconjunto alcanza un estado estable. En este caso, aunque el programa hubiera transformado al multiconjunto de forma distinta a la antes mostrada el estado estable sería el mismo.

Ahora consideremos otro ejemplo, un programa para la ordenación de un multiconjunto

$$sort = (i,x),(j,y) \rightarrow \{\!|(i,y),(j,x)|\!\} \Leftarrow (i > j) \ \wedge \ (y < x)$$

para este caso consideramos un multiconjunto de pares $(index, value)$, donde el primer elemento $(index)$ indica el índice de valor $(value)$ en una lista de valores que es representada por el multiconjunto y el programa intercambia al i-ésimo valor hasta que se alcanza el estado estable donde todos los valores están bien ordenados. Esto es la lista $[3,1,6,8,3,0]$ estaría representada por el multiconjunto $\{\!|(1,3),(2,1),(3,6),(4,8),(5,3),(6,0)|\!\}$ y el programa *sort* daría como resultado el multiconjunto $\{\!|(1,0),(2,1),(3,3),(4,3),(5,6),(6,8)|\!\}$

El siguiente es un ejemplo más, el programa calcula el factorial de n, en este caso el programa inicia con el multiconjunto $\{\!|n|\!\}$ y termina

con el multiconjunto $\{\!|n!|\!\}$

$$\begin{aligned} par &= x \to \{\!| -1, x |\!\} \Leftarrow x \text{ es par} \\ pred &= x \to \{\!| x-1, -x |\!\} \Leftarrow x > 0 \\ cero &= x \to \{\!| -1 |\!\} \Leftarrow x = 0 \\ prod &= x, y \to \{\!| xy |\!\} \Leftarrow \textbf{verdadero} \end{aligned}$$

$$factorial = prod \circ (pred + cero) \circ par$$

El programa *par* toma al multiconjunto original y si n es par entonces agrega al multiconjunto un -1. El programa *pred* calcula los $n-1$ números anteriores a n, solo que estos tendrán signo negativo para indicar cuales ya fueron calculados. El programa *cero* agrega el factorial de cero pero con signo negativo. Finalmente el programa *prod* calcula el producto de todos elementos en el multiconjunto, solo que como estos son negativos necesitaríamos hacer una corrección del valor final, pero esto ya fue hecho por el programa *par*.

Entonces el programa *factorial* presentaría las siguientes configuraciones:

$$\begin{aligned} &\langle prod \circ (pred + cero) \circ par, \{\!|4|\!\}\rangle \\ &\quad \to \langle prod \circ (pred + cero), \{\!|4, -1|\!\}\rangle \\ &\quad \to^* \langle prod, \{\!| -4, -3, -2, -1, -1, -1|\!\}\rangle \\ &\quad \to^* \{\!|24|\!\} \end{aligned}$$

3.1.4. Ventajas y desventajas

La falta de una secuencialidad artificial en Gamma tiene tres importantes consecuencias:

- Gamma tiene un alto grado de naturalidad, podemos ver a Gamma como un lenguaje intermedio en la derivación de programas a partir de su especificación, permitiendo al programador el diseñar una versión muy abstracta de sus programas (para los cuales es fácil probar la corrección).

- Es más fácil escribir un programa paralelo que un programa secuencial en Gamma.
- Gamma es también conveniente como la base de un *lenguaje de coordinación* para la descripción de las interacciones entre entidades individuales en aplicaciones a gran escala.

Al trabajar con Gamma se han encontrado algunos problemas, entre los más importantes se encuentran:

1. El lenguaje no da facilidades al programador para especificar estrategias de control; no provee apoyo para la programación jerárquica.
2. Dada su semántica hay una explosión combinatoria en las operaciones de los programas y es difícil alcanzar un nivel decente de eficiencia en una implementación del lenguaje de propósito general.

Para lograr la modularidad es deseable que un lenguaje ofrezca un conjunto de operadores para la combinación de programas, así como leyes algebraicas para estos operadores que permitan establecer relaciones entre los programas.

La falta de apoyo para los datos estructurados y la dificultad para plantear estrategias de control se presentan debido a que entre los motivos originales del lenguaje estaba el describir programas con las menores restricciones posibles. Pero una consecuencia de esto es que los programadores en ocasiones tienen que utilizar algunos trucos de codificación para expresar sus algoritmos.

Estos problemas quedan ejemplificados en algunos de los ejemplos anteriores, el programa *max* es un buen ejemplo de un programa en Gamma, pues de manera muy intuitiva presenta la forma de obtener el mínimo de un multiconjunto, pero el programa *sort* puede ya no ser tan claro, pues la forma de representar una lista no es tan obvia y eso afectará la claridad del programa. Pero el programa *factorial* no es nada intuitivo, ya que el cálculo del factorial es secuencial por naturaleza, y como mencionamos en Gamma es más difícil tratar este tipo de problemas.

3.2. Gamma de orden superior

En la definición original de Gamma se manejan dos tipos de términos: los programas y los multiconjuntos. Gamma de orden superior unifica estas dos categorías en un solo elemento: la configuración. Cabe aclarar que en esta parte utilizamos el término configuración como un término que une a los programas y los multiconjuntos y no como se usa en semántica, donde una configuración era un par $\langle e, \sigma \rangle$ con e una expresión y σ un estado.

3.2.1. Sintaxis y semántica de Gamma de orden superior

La sintaxis del lenguaje esta definida como sigue:

$$
\begin{aligned}
Conf &:= Pasiva \mid Activa \\
Pasiva &:= [\emptyset,\ Amb] \\
Activa &:= [Prog,\ Amb] \\
Amb &:= (Var_1 = Multexp_1, \ldots, Var_m = Multexp_n) \\
Prog &:= Par \mid Prog \circ Par \\
Par &:= Sim \mid Par + Sim \\
Sim &:= x_1 : Var_{i_1}, \ldots, x_k : Var_{i_k} \\
&\qquad \rightarrow SecExp_1 : Var_{j_1}, \ldots, SecExp_m : Var_{j_m} \\
&\qquad \Leftarrow C(x_1, \ldots, x_k) \\
SecExp &:= Exp_1, \ldots, Exp_n \mid \emptyset \\
Exp &:= Conf \mid Conf.Var_i \mid x_i \mid ExpArit \mid ExpBool \mid \\
&\quad Multexp \mid Car(Multexp) \mid \ldots \\
Multexp &:= \emptyset \mid Multexp_1 \oplus Multexp_2 \mid Multexp_1 \ominus Multexp_2 \mid \\
&\quad \{SeqExp\} \ldots
\end{aligned}
$$

Una configuración está hecha de un programa (posiblemente vacío) y una colección de multiconjuntos etiquetados. Una configuración con un programa vacío es llamada *pasiva*, en otro caso es *activa*. La colección de multiconjuntos de la configuración puede ser vista como el *ambiente*

del programa. Cada componente del ambiente es un multiconjunto. C es una expresión booleana que representa la condición de la reacción. Los programas pueden ser compuestos usando el operador secuencial $\circ$ y el operador paralelo $+$. $Conf.Var_i$ regresa el valor del campo Var_i en la configuración $Conf$ (cuando $Conf$ ha alcanzado un estado pasivo). $\oplus$ y $\ominus$ son la unión y diferencia de multiconjuntos.

La semántica de Gamma de orden superior está definida de la siguiente manera:

$$
\begin{array}{l}
\textbf{si } P = x_1 : Var_{i_1}, \ldots, x_k : Var_{i_k} \\
\qquad\qquad \rightarrow Seqexp_1 : Var_{j_1}, \ldots, Seqexp_m : Var_{j_m} \\
\qquad\qquad \Leftarrow C(x_1, \ldots, x_k) \\
\quad \wedge\ x_1 \in M_{i_1}, \ldots, x_k \in M_{i_k} \\
\quad \wedge\ C(x_1, \ldots, x_k) \\
\quad \wedge\ \{i_1, \ldots, i_k\} \subseteq \{1, \ldots, n\} \\
\quad \wedge\ \{j_1, \ldots, j_m\} \subseteq \{1, \ldots, n\} \\
\quad \textbf{entonces} \\
\qquad [P, (Var_1 = M_1, \ldots, Var_n = M_n)] \\
\qquad\qquad \rightarrow [P, (Var_1 = M_1', \ldots, Var_n = M_n')] \\
\quad \textbf{donde} \qquad M_i' = M_i \ominus \{x_a | i_a = i\} \oplus \{Seqexp_a | j_a = i\}
\end{array}
$$

$$
\begin{array}{l}
\textbf{si } P = x_1 : Var_{i_1}, \ldots, x_k : Var_{i_k} \\
\qquad \rightarrow Seqexp_1 : Var_{j_1}, \ldots, Seqexp_m : Var_{j_m} \\
\qquad \Leftarrow C(x_1, \ldots, x_k) \\
\quad \wedge \neg \exists x_1 \in M_{i_1}, \ldots, x_k \in M_{i_k} \textbf{ tal que } C(x_1, \ldots, x_k) \\
\quad \wedge \forall i \in \{i_1, \ldots, i_k\}. Inert(M_i) \\
\quad \wedge \{i_1, \ldots, i_k\} \subseteq \{1, \ldots, n\} \\
\quad \wedge \{j_1, \ldots, j_m\} \subseteq \{1, \ldots, n\} \\
\quad \textbf{entonces} \\
\qquad [P, (Var_1 = M_1, \ldots, Var_n = M_n)] \\
\qquad\qquad \rightarrow [\emptyset, (Var_1 = M_1, \ldots, Var_n = M_n)]
\end{array}
$$

$$[\emptyset, (V_1 = M_1, \ldots, V_n = M_n)].V_i \rightarrow M_i$$

$$\frac{[P_2, E] \rightarrow [P_2', E']}{[P_1 \circ P_2, E] \rightarrow [P_1 \circ P_2', E']}$$

$$\frac{[P_2, E] \rightarrow [\emptyset, E]}{[P_1 \circ P_2, E] \rightarrow [P_1, E]} \qquad \frac{[P_1, E] \rightarrow [\emptyset, E] \quad [P_2, E] \rightarrow [\emptyset, E]}{[P_1 + P_2, E] \rightarrow [\emptyset, E]}$$

$$\frac{[P_1, E] \rightarrow [P_1', E']}{[P_1 + P_2, E] \rightarrow [P_1' + P_2, E']} \qquad \frac{[P_2, E] \rightarrow [P_2', E']}{[P_1 + P_2, E] \rightarrow [P_1 + P_2', E']}$$

$$\frac{X \rightarrow X'}{\{X\} \oplus M \rightarrow \{X'\} \oplus M} \qquad \frac{[P_2, E] \rightarrow [P_2', E']}{[P_1 + P_2, E] \rightarrow [P_1 + P_2', E']}$$

$Inert(M) \Leftrightarrow \forall[P, (Var_1 = M_1, \ldots, Var_n = M_n)] \in M, \forall N \in M$
tal que $multiconjunto(N)$
$P = \emptyset \ \wedge \ Inert(M_1) \ \wedge \ \ldots \ Inert(M_n) \ \wedge \ Inert(N)$
y $multiconjunto(N)$ indica si la variable N es un multiconjunto.

La estructura de la semántica operacional del lenguaje muestra que el ambiente regresado por una composición paralela $P_1 + P_2$ debe ser estable para ambos P_1 y P_2.

3.2.2. Programación en Gamma de orden superior

Una de las motivaciones para esta extensión de orden superior fue el proveer al usuario con un poder expresivo más grande para la definición de sus estrategias de control. Las facilidades de control pueden ser ofrecidas como primitivas de un lenguaje o ser definidas dentro del lenguaje mismo.

La mayoría de los programas en Gamma podrían transformarse de manera casi inmediata a Gamma de orden superior, pero no podemos decir lo mismo de los programas de Gamma de orden superior respecto a Gamma. Por ejemplo el programa *max* de Gamma, en Gamma de orden superior sería el siguiente:

$$\begin{aligned} max &= [P, M] \\ &\text{donde:} \\ P &= x, y : M \rightarrow \{]x[\} \Leftarrow x \geq y \end{aligned}$$

El programa *factorial* en Gamma de orden superior es:

$$\begin{aligned} factorial &= [fact, M, M_f = \emptyset] \\ &\text{donde:} \\ fact &= prod + pred + cero \\ cero &= x : M \rightarrow 1 : M_f \Leftarrow x = 0 \\ pred &= x : M \rightarrow x - 1 : M, x : M_f \Leftarrow x > 0 \\ prod &= x, y : M_f \rightarrow xy : M_f \Leftarrow \textbf{verdadero} \end{aligned}$$

Esta versión puede ser más clara que la de Gamma, ya que no usa trucos para el calculo del factorial, lo único que hace es utilizar un multiconjunto auxiliar con los operandos del producto y el multiconjunto original es utilizado para calcular el predecesor de n. Otra ventaja de esta versión es que el producto de los valores puede realizarse en paralelo con el cálculo de los predecesores.

Capítulo 4

Gamma estructurado y Gamma de orden superior

4.1. Gamma estructurado

Como ya se mencionó, la falta de apoyo para datos estructurados produce algunos problemas, principalmente en la claridad de los programas, así como en la implementación de ellos. Para solucionar este problema sin afectar las propiedades básicas del lenguaje, se ha propuesto un formalismo llamado **Gamma estructurado**, el cual permite la definición recursiva de tipos. Gamma estructurado se basa en la noción de *multiconjunto estructurado*, el cual es un conjunto de direcciones que satisfacen relaciones específicas y están asociadas con valores. Las relaciones expresan una forma de vecindad entre moléculas de la solución, las cuales pueden ser usadas en la condición de la reacción o transformadas por la acción de un programa. Consideraremos que un tipo está definido con base en reglas de reescritura sobre las relaciones de un multiconjunto; un multiconjunto estructurado será del tipo T si su conjunto de direcciones satisfacen las invariantes expresadas por las reglas de reescritura que definen a T.

4.1.1. Multiconjuntos estructurados

Un multiconjunto estructurado es un conjunto de direcciones que satisfacen relaciones específicas. Por ejemplo la lista

$$[5,\ 2,\ 7]$$

puede ser representada por un multiconjunto cuyo conjunto de direcciones es

$$\{a_1, a_2, a_3\}$$

y tiene los valores asociados (denotados por $\overline{a_i}$)

$$\overline{a_1} = 5, \quad \overline{a_2} = 2, \quad \overline{a_3} = 7.$$

para este multiconjunto estructurado tenemos definida la relación binaria **next**, la cual establece la relación del siguiente elemento, y la relación **end** que indica el elemento final de la lista. Para este caso las direcciones satisfacen las siguientes relaciones:

$$\textbf{next}\ a_1\ a_2, \qquad \textbf{next}\ a_2\ a_3, \qquad \textbf{end}\ a_3$$

4.1.2. Sintaxis y semántica de Gamma estructurado

En 1998, Fradet y Le Métayer propusieron una sintaxis y semántica para Gamma estructurado. Basándonos en dicha definición consideramos que un programa de Gamma estructurado está definido en términos de pares de una condición y una acción. Esta última puede:

- Verificar y modificar las relaciones sobre las direcciones.
- Verificar y modificar los valores asociados con las direcciones.

Para definir la sintaxis y la semántica de Gamma estructurado, consideramos tres dominios básicos:

- $\mathbb{R}$: el conjunto de símbolos de relación,
- A: el conjunto de direcciones,

- **V**: el conjunto de valores.

La sintaxis de programas de Gamma estructurado esta descrito por la siguiente gramática:

$$\begin{aligned}
\langle Program\rangle &::= \texttt{ProgName} = [\langle Reaction\rangle]^* \\
\langle Reaction\rangle &::= \langle Action\rangle \Leftarrow \langle Condition\rangle \\
\langle Condition\rangle &::= \mathbf{r}x_1 \cdots x_n \mid f^{Bool}(\overline{x_1}, \ldots, \overline{x_n}) \mid \\
&\quad \langle Condition\rangle, \langle Condition\rangle \\
\langle Action\rangle &::= \mathbf{r}x_1 \cdots x_n \mid x := f^{\mathbf{V}}(\overline{x_1}, \ldots, \overline{x_n} \mid \\
&\quad \langle Action\rangle, \langle Action\rangle
\end{aligned}$$

donde **r** es una relación de $\mathbb{R}$ de aridad n, x_i es un dirección de una variable, $\overline{x_i}$ es el valor de la dirección x_i y f^{Bool} es una función definida de V^n a **X**.

En Gamma estructurado al igual que en Gamma el orden de evaluación de las operaciones básicas de una acción no es relevante. Pero los programas en Gamma estructurado deben cumplir dos condiciones sintácticas más:

- Si $\overline{x}$ ocurre en la reacción entonces x ocurre en la condición.
- Una acción no puede incluir dos asignaciones a la misma variable.

Denotamos con $\mathbf{A}(M)$ el conjunto de direcciones presentes en un multiconjunto M y, como se mencionó anteriormente, $\uplus$ es la unión de multiconjuntos. Un multiconjunto estructurado M puede ser visto como $M = Rel \uplus Val$ donde

- *Rel* es un *multiconjunto* de relaciones representadas por n-adas de la forma $(\mathbf{r}, a_1, \ldots, a_n)$, donde $r \in \mathbf{R}$ y $a_i \in \mathbf{A}$.
- *Val* es un *conjunto* de valores representados por ternas de la forma (val, a, v), donde $a \in \mathbf{A}$ y $v \in \mathbf{V}$

De esta forma el ejemplo de la lista puede ser escrito como:

$$\{\ (\mathbf{next}, a_1, a_2), (\mathbf{next}, a_2, a_3), (\mathbf{end}, a_3), (\mathbf{val}, a_1, 5), (\mathbf{val}, a_2, 2), (\mathbf{val}, a_3, 7)\}$$

Una característica más para los multiconjuntos estructurados es que una dirección x no puede tener más de un valor, es decir, ocurre a lo más una vez en Val. Sin embargo puede ocurrir varias veces en Rel, de aquí que es necesario que

$$\mathbf{A}(Rel) \subseteq \mathbf{A}(Val)$$

pero no

$$\mathbf{A}(Val) \subseteq \mathbf{A}(Rel)$$

Así para poder definir la semántica de los programas, definimos tres funciones asociadas con cada reacción $C \Longmapsto A$ de la siguiente manera:

$$\begin{aligned}
\mathcal{T}(a_1,\ldots,a_i,b_1,\ldots,b_j) &= (\mathbf{val},a_1,\overline{a_1}) \in Val \wedge \ldots \wedge \\
&\quad (\mathbf{val},a_i,(\overline{a_i})) \in Val \wedge \\
&\quad (\mathbf{val},b_1,\overline{b_1}) \in Val \wedge \ldots \wedge \\
&\quad (\mathbf{val},b_j,\overline{b_j}) \in Val \wedge \\
&\quad \lfloor C \rfloor \\
\mathcal{C}(C)(a_1,\ldots,a_i,b_1,\ldots,b_j) &= \{(\mathbf{val},a_1,\overline{a_1}),\ldots,(\mathbf{val},a_i,\overline{a_i}), \\
&\quad (\mathbf{val},b_1,\overline{b_1}),(\mathbf{val},b_j,\overline{b_j}) \\
&\quad \uplus \lceil C \rceil \\
\mathcal{A}(A)(a_1,\ldots,a_i,b_1,\ldots,b_j) &= \{(\mathbf{val},a_1,\overline{a_1}),\ldots,(\mathbf{val},a_i,\overline{a_i})\} \\
&\quad \uplus \lceil A \rceil
\end{aligned}$$

donde $\lfloor\ \rfloor$ está definido por

$$\begin{aligned}
\lfloor X_1, X_2 \rfloor &= \lfloor X_1 \rfloor \wedge \lfloor X_2 \rfloor \\
\lfloor \mathbf{r} x_1 \ldots x_n \rfloor &= (\mathbf{r},x_1,\ldots,x_n) \in Rel \\
\lfloor f(\overline{x_1},\ldots,\overline{x_n}) \rfloor &= f(\overline{x_1},\ldots,\overline{x_n})
\end{aligned}$$

y $\lceil\ \rceil$ definida por

$$\begin{aligned}
\lceil X_1, X_2 \rceil &= \lceil X_1 \rceil + \lceil X_2 \rceil \\
\lceil \mathbf{r} x_1 \ldots x_n \rceil &= \{\mathbf{r},x_1,\ldots,x_n\} \\
\lceil f(\overline{x_1},\ldots,\overline{x_n}) \rceil &= \emptyset \\
\lceil x := f(\overline{x_1},\ldots,\overline{x_n}) \rceil &= \{(\mathbf{val},x,f(\overline{x_1},\ldots,\overline{x_n}))\}
\end{aligned}$$

y

- $\{a_1, \ldots, a_i\}$ denota el conjunto de variables no asignadas cuyos valores ocurren en la reacción.
- $\{b_1, \ldots, b_j\}$ denota el conjunto de variables que ocurren en la condición C.
- $\{c_1, \ldots, c_k\}$ denota el conjunto de variables que ocurren solo en la acción A.

La función booleana $\mathcal{T}(C)$ representa la condición de aplicación de una reacción. La función $\mathcal{C}(C)$ representa las n-adas seleccionadas por la condición. La función $\mathcal{A}(A)$ representa las n-adas agregadas por la acción.

La semántica de un programa de Gamma estructurado

$$P = [A_1 \Leftarrow C_1, \ldots, A_m \Leftarrow C_m]$$

aplicado a un multiconjunto M está definido, como el conjunto de formas normales del siguiente sistema de reescritura:

$$M \longrightarrow_P \mathcal{GC}(M)$$
$$\text{si} \quad \forall \{x_1, \ldots, x_n\} \subseteq \mathbf{A}(M) \quad \forall i \in [1, \ldots, m] \quad \neg \mathcal{T}(C_i)(x_1, \ldots, x_n)$$
$$M \longrightarrow_P M - \mathcal{C}(C_i)(x_1, \ldots, x_n) \uplus \mathcal{A}(A_i)(x_1, \ldots, x_n, y_1, \ldots, y_k)$$

donde $y_1, \ldots, y_k \notin \mathbf{A}(M)$ y $\{x_1, \ldots, x_n\} \subseteq \mathbf{A}(M), i \in [1, \ldots, m]$ y $\mathcal{F}(C_i)(x_1, \ldots, x_n)$

Si no hay una n-ada de direcciones que satisfagan alguna condición, entonces se ha encontrado una forma normal. La función $\mathcal{GC}$ elimina de Val las direcciones que no ocurren en Rel. Más formalmente:

$$\mathcal{GC}(Rel + Val) = Rel \uplus \{(\text{val}, a, v) | (\text{val}, a, v) \in Val \wedge a \in \mathbf{A}(Rel)\}$$

Usamos la notación $M \longmapsto_P M'$ para decir que M se reescribe como M' ($M \overset{*}{\longrightarrow}_P$) y M' es una forma normal de P.

Una n-ada de direcciones $(x_1, \ldots, x_n)$ y un par (C_i, A_i) tal que $\mathcal{F}(C_i)(x_1, \ldots, x_n)$ son elegidos no deterministamente. El multiconjunto es transformado eliminando $\mathcal{C}(C_i)(x_1, \ldots, x_n)$, agregando nuevas direcciones $y_1, \ldots, y_k$ y agregando $\mathcal{A}(A_i)(x_1, \ldots, x_n, y_1, \ldots, y_k)$.

El siguiente programa es la versión en Gamma estructurado del programa "exchange sort".

$$Sort = [\ \mathbf{next}\ x\ y,\ \ x := \overline{y},\ \ y := \overline{x} \Leftarrow \mathbf{next}\ x\ y,\ \ \overline{x} > \overline{y}\ ]$$

dos direcciones seleccionadas x y y deben satisfacer la relación **next** $x\ y$ y sus valores $\overline{x}$ y $\overline{y}$ deben ser tales que $\overline{x} > \overline{y}$. La acción intercambia sus valores y la relación permanece sin cambios. Para completar el ejemplo, debemos indicar que el multiconjunto reescrito por *Sort* es de tipo *List* y que la reacción preserva el tipo del multiconjunto. De esta forma tendremos algunas nuevas consideraciones respecto a los programas de Gamma estructurado, a saber, la necesidad de definir los distintos tipos de datos así como de demostrar que las reacciones no modifican la definición del tipo que se esté considerando.

La semántica propuesta anteriormente para Gamma estructurado es una semántica denotacional; sin embargo, podríamos definir una semántica operacional para este formalismo. En general podemos encontrar una correspondencia entre Gamma estructurado y Gamma, considerando como estructura básica de Gamma a los multiconjuntos estructurados e indicando, para un programa dado, todas las posibles relaciones entre las direcciones, incluyendo la relación de cada dirección con su valor; de esta forma el modelo básico de Gamma permanecería sin cambios. En consecuencia podemos considerar que la semántica operacional de Gamma se aplica también a Gamma estructurado.

Utilizando esta idea podemos reescribir el programa de "exchange sort" en Gamma de la siguiente forma:

$$\begin{aligned} sort \ = \ & (\mathbf{next}, x, y), (\mathbf{val}, x, \overline{x}), (\mathbf{val}, y, \overline{y}) \\ & \rightarrow \{|(\mathbf{next}, x, y), (\mathbf{val}, x, \overline{y}), (\mathbf{val}, y, \overline{x})|\} \Leftarrow \overline{x} > \overline{y} \end{aligned}$$

donde **next** y **val** están definidos de la forma antes descrita.

4.1.3. Tipos estructurados

Los multiconjuntos estructurados tienen una ventaja más permiten al programador hacer una organización explícita de los datos. Podemos

dar la noción de tipo basándonos en multiconjuntos estructurados. Definiremos un tipo en términos de reglas de reescritura sobre las relaciones de un multiconjunto. De esta manera diremos que un multiconjunto estructurado es de un tipo si su conjunto de direcciones pueden producir, mediante el sistema de reescritura, la definición del tipo.

Así la sintaxis de tipos esta definida por la siguiente gramática:

$$\begin{aligned} \langle TypeDecl \rangle &::= \texttt{TypeName} = \langle Prod \rangle, [\langle NonTerm \rangle = \langle Prod \rangle]^* \\ \langle NonTerm \rangle &::= \texttt{NonTerminalName}\ x \ldots x_n \\ \langle Prod \rangle &:: \mathbf{r}\ x_1 \ldots x_n | \langle NonTerm \rangle | \langle Prod \rangle, \langle Prod \rangle \end{aligned}$$

donde $\mathbf{r} \in \mathbf{R}$, es de aridad n, y x_i es una variable que denota una dirección.

De esta forma, las listas pueden ser definidas como:

$$\begin{aligned} List &::= L\ x \\ L\ x &::= \mathbf{next}\ x\ y,\ L\ y \\ L\ x &::= \mathbf{end}\ x \end{aligned}$$

4.1.4. Verificación de tipos

Como se mencionó, al manejar el concepto de tipos debemos asegurar que al aplicar un programa a un multiconjunto estructurado de tipo T, dicho programa no modifique el tipo T, pero se ha demostrado que la verificación de tipos es un problema indecidible para tipos definidos por una gramática libre de contexto. Fradet y Le Métayer proponen una subclase de tipos para los cuales existe un algoritmo prácticamente completo [5]. Regresando al caso de Gamma estructurado en [6] Fradet y Le Métayer proponen un algoritmo de verificación de tipos, y demuestran que este algoritmo es correcto pero incompleto.

4.2. Transformación de programas de Gamma estructurado a Gamma de orden superior

Ahora que hemos visto como Gamma estructurado nos proporciona un mejor manejo de las estructuras de datos veremos la forma en que podemos llevar esta idea a Gamma de orden superior para manejar de forma más clara las estructuras de datos en éste último formalismo.

Para transformar programas de Gamma estructurado a Gamma de orden superior modificaremos la sintaxis de Gamma de orden superior para agregar algunos rasgos de Gamma estructurado.

De esta forma consideraremos los mismos tres dominios básicos para Gamma estructurado:

- R: el conjunto de símbolos de relación
- A: el conjunto de direcciones
- V: el conjunto de valores

Y la sintaxis modificada estaría definida como sigue:

$$
\begin{aligned}
Conf &= Pasiva \mid Activa \\
Pasiva &= [\emptyset,\ Amb] \\
Activa &= [Prog,\ Amb] \\
Amb &= (Var_1 = Multiexp_1, \ldots, Var_m = Multiexp_n) \\
Prog &= Par \mid Prog \cdot Par \\
Par &= Sim \mid Par + Sim \\
Sim &= x_1 : Var_{i_1}, \ldots, x_k : Var_{i_k} \\
&\qquad \rightarrow SecExp_1 : Var_{j_1}, \ldots, SecExp_m : Var_{j_m} \\
&\qquad \Leftarrow Condicion \\
Condicion &= \mathrm{r}\ x_1, \ldots, x_n \mid f^{Bool}(\bar{x}_1, \ldots, \bar{x}_n) \mid Condicion, Condicion \\
SecExp &= Exp_1, \ldots, Exp_n \mid \emptyset \\
Exp &= Conf \mid Conf.Var_i \mid x_i \mid ExpArit \mid ExpBool \mid \\
&\quad Multiexp \mid Car(Multiexp) \mid \mathrm{r}\ x_1, \ldots, x_n \\
Multiexp &= \emptyset \mid Multiexp_1 \oplus Multiexp_2 \mid Multiexp_1 \ominus Multiexp_2
\end{aligned}
$$

donde $\mathbf{r}(\in \mathbf{R})$ denota una relación n-aria, x_i es una variable de dirección, $\overline{x_i}$ es el valor de la dirección x_i y f^x es una función de V^n a los booleanos.

Una configuración al igual que en Gamma de orden superior, está hecha de un programa (posiblemente vacío) y una colección de multiconjuntos etiquetados. Una configuración con un programa vacío es llamada *pasiva*, en otro caso es *activa*. La colección de multiconjuntos de la configuración puede ser vista como el *ambiente del programa*. Cada componente del ambiente es un multiconjunto de un tipo específico y tenemos definidas nuevamente las operaciones de composición secuencial $\circ$ y composición paralela $+$

Además, se deben satisfacer dos condiciones adicionales:

- Si $\overline{x}$ ocurre en la reacción entonces x ocurre en la condición
- Una acción no puede incluir dos asignaciones de la misma variable

Podemos considerar que la semántica del lenguaje permanece igual que para Gamma de orden superior ya que solo se hacen restricciones a las reglas sintácticas.

4.2.1. Un ejemplo

Consideramos nuevamente el programa *Sort* estructurado tal que

$$Sort \;=\; [\mathbf{next}\;\; x\; y, \bar{x} > \bar{y} \longmapsto \mathbf{next}\;\; x\; y, x := \bar{y},\; y := \bar{x}]$$

Y sea M el multiconjunto estructurado que estamos considerando tal que :

$$\begin{aligned} M \;=\; \{&(\mathbf{next}, a_1, a_2), (\mathbf{next}, a_2, a_3), \ldots, (\mathbf{next}, a_{n-1}, a_n),\\ &(\mathbf{val}, a_1, x_1), (\mathbf{val}, a_2, x_2), \ldots, (\mathbf{val}, a_n, x_n), (\mathbf{end}, a_n)\} \end{aligned}$$

Sea C la siguiente configuración: $[Sort, M]$.
El programa correspondiente en Gamma de orden superior sería:

$$\begin{aligned} Sort \;=\; &(\mathbf{next}\; x, y) : M, (\mathbf{val}, x, \bar{x}) : M, (\mathbf{val}, y, \bar{y}) : M\\ &\rightarrow (\mathbf{next}\; x, y) : M, (\mathbf{val}, x, \bar{y}) : M, (\mathbf{val}, y, \bar{x}) : M \Leftarrow \bar{x} > \bar{y} \end{aligned}$$

Capítulo 5

Replicación en Lotus Notes

En este capítulo presentaremos una aplicación de Gamma de orden superior: una especificación formal del algoritmo de replicación de objetos en Lotus Notes, para el cual ya existía una especificación informal.

Una de las principales utilidades que se le han dado a los métodos formales es como una herramienta de especificación. En este parte haremos una comparación entre una especificación basada en Gamma de orden superior y una especificación informal. Los sistemas de reglas de reescritura de multiconjuntos de orden superior, como lo es Gamma de orden superior, presentan algunas características muy útiles en la especificación del software moderno, estas características son:

- *Distribución.* La reescritura de multiconjuntos es altamente paralela, esta es una de las características más distintivas de los sistemas distribuidos.
- *Abierta.* El orden superior nos permite manipular las reglas, y con ello su especificación. Esto nos brinda una forma de modularidad, reutilización e intercambio de partes de una especificación. Tales especificaciones son llamadas *abiertas*, en el sentido de que son más manejables, o adaptables con facilidad.

5.1. Lotus Notes y su algoritmo de replicación

Lotus Notes permite el trabajo de grupos de personas sobre un conjunto de documentos. Los documentos son objetos semiestructurados que pueden estar compuestos de gráficas, imágenes, dibujos e información numérica con texto. El conjunto de documentos forma una base de datos.

A diferencia de los sistemas que mantienen a las bases de datos sobre un servidor central, el sistema Notes opta por una arquitectura distribuida en la cual cada participante trabaja sobre réplicas locales de las bases de datos compartidas. En esta arquitectura no hay una réplica maestra de una base de datos. La propagación de los cambios en la base de datos ocurre entre pares de réplicas como una actividad en segundo plano. Los conflictos son resueltos en base a indicadores de tiempo de modificación para cada base de datos .

El algoritmo de replicación garantiza da "consistencia de última instancia" para aplicaciones donde los cambios sobre documentos existentes son menos frecuentes comparados con la creación de nuevos documentos, y donde el tiempo de propagación de los cambios no es considerable. Los sistemas de conferencia asíncronos, como los boletines electrónicos, son ejemplos típicos de estos sistemas.

Concretamente, el algoritmo de replicación de Lotus Notes es un modelo de "one-way pool". Este tipo de algoritmos son llamados así porque son ejecutados en la computadora y solo envían versiones nuevas de los documentos hacia una computadora remota. El algoritmo consiste de tres pasos:

1. Crear una lista de bases de datos (que requieren replicación).
2. Crear una lista de documentos (que requieren replicación).
3. Replicar los documentos listados.

Como la idea original es que los grupos están dispersos geográficamente y los participantes no están conectados regularmente, el algoritmo fue diseñado para el balanceo estático y los respaldos automáticos.

La estrategia de respaldos explota la propiedad de que si una de las réplicas remotas no está disponible, otra réplica puede ser seleccionada, lo cual muestra que el algoritmo es robusto.

De aquí que, en Lotus Notes, haya más de una réplica de un objeto. Otra característica es el control de acceso y bloqueo.

5.2. Especificación del algoritmo de replicación

En esta sección presentaremos una especificación informal junto con una especificación formal para cada uno de los tres pasos del algoritmo de replicación. Daremos una especificación formal basada en reescritura de multiconjuntos.

Como ya mencionamos, los multiconjuntos son contenedores de elementos de un tipo en particular, de esta forma solo debemos especificar como serán los multiconjuntos que utilizaremos para la especificación formal. Los elementos de los multiconjuntos serán de los tipos siguientes: indicador de tiempo, identificadores, cadenas o enteros.

Para mayor comprensión, el algoritmo de replicación será primero propuesto para replicación entre dos máquinas.

5.2.1. Primer paso

Los diseñadores de Lotus Notes escribieron una especificación compacta de su algoritmo de replicación en inglés estructurado. El primer paso fue especificado como sigue:

1. *Crear una lista de bases de datos (que requieren replicación)*:

   Encontrar las bases de datos comunes a ambas computadoras la local y la remota, y para cada base de datos, verificar si la base de datos remota ha sido modificada después de la última replicación con la computadora local.

Para modelar este paso debemos representar tanto la computadora local como la remota junto con sus respectivas bases de datos. Así consideraremos dos multiconjuntos, loc_1 y rem_1. Cada elemento de estos

multiconjuntos representa una base de datos: *id* identifica la base de datos, *docs* representa todos los documentos contenidos en la base de datos, y t indica el tiempo de la última réplica.

$$loc_1 \equiv rem_1 \equiv \{\!|(id, docs, t)|\!\}$$

De esta forma creamos una lista para las bases de datos compartidas entre ambas computadoras la local y la remota. Esta lista es modelada por el multiconjunto $list_1$.

El primer paso del algoritmo puede ser especificado como una regla de reescritura sobre el contenido de los multiconjuntos. La regla $paso_1$ filtra todas las bases de datos que están compartidas entre la computadora local y remota. Consideraremos una base de datos (I_l, D_l, T_l) de la computadora local loc_1 y una base de datos (I_r, D_r, T_r) de la computadora remota rem_1. Si ambas bases de datos tienen el mismo identificador $(I_r = I_l)$, entonces encontramos una base de datos compartida. Si la réplica de la base de datos de la computadora remota ha sido modificada después de la última modificación de la réplica local $(T_r > T_l)$, entonces ponemos toda la información de la base de datos en el multiconjunto intermedio $list_1$ el cual será procesado en los siguientes pasos. Finalmente, debemos asegurar que la réplica remota permanece sin cambio, reinsertando (I_r, D_r, T_r) en rem_1.

$$\begin{aligned} paso_1 \;=\; & (I_l, D_l, T_l) : loc, (I_r, D_r, T_r) : rem_1 \\ & \rightarrow (I_l, D_l, T_l, D_r, T_r) : list_1, (I_r, D_r, T_r) : rem_l \\ & \Leftarrow (I_l = I_r) \;\wedge\; (T_r > T_l) \end{aligned}$$

Cuando la regla $paso_1$ no se puede aplicar más, aseguramos que $list_1$ contiene todas las bases de datos compartidas para la cual las réplicas remotas son más actuales. Además sabemos que loc_1 no tiene disponibles las bases de datos fuera de tiempo. Finalmente podemos asegurar que la réplica remota no fue modificada.

5.2.2. Segundo paso

En el primer paso del algoritmo de replicación encontramos las bases de datos compartidas entre la computadora local y remota. En el

segundo paso identificamos los documentos compartidos entre las bases de datos compartidas. La base de datos será identificada por *id*. Las réplicas local y remota serán modeladas por los multiconjuntos loc_2 y rem_2, respectivamente; *prev* y *pres* representan el tiempo de replicación más reciente de cada multiconjunto.

La estructura de los documentos es análoga a la estructura de las bases de datos. Ambos tienen un identificador y un indicador de tiempo de la modificación más reciente. Las bases de datos son descompuestas en documentos. La estructura interna de los documentos no será detallada. Solo es necesario que tengamos la capacidad de copiar el contenido *cont* de cualquier tipo de documento, sea este texto, imágenes, audio u otra cosa. En la práctica las implementaciones de Lotus Notes descomponen el contenido de un documento recursivamente y presentan algoritmos especiales para el copiado para diferentes tipos de documentos. Sin embargo, el algoritmo abstracto de replicación no llega a este nivel de detalle.

$$docs \equiv loc_1 \equiv list_2 \equiv \{\!|id, cont, t|\!\}$$
$$prev \equiv pres \equiv t$$

La especificación formal del segundo paso del algoritmo de replicación consiste de dos subpasos, uno para la réplica remota de la base de datos y uno para la réplica local.

2. *Crear una lista de documentos (que necesitan replicación)*:
   - *a*) Abrir la base de datos remota y crear una lista de todos los documentos que han sido modificados después que la última replicación. Para cada documento incluimos su identificador de documento y su último tiempo de modificación.
   - *b*) Abrir la base de datos local y crear una lista de todos los documentos.

En el primer subpaso filtramos todos los documentos de la réplica remota, para obtener solo los que sean más recientes que *prev*, la última modificación de la réplica local. Todos los documentos que cumplen esta condición son coleccionados en el nuevo multiconjunto $list_2$. Estos documentos son removidos al mismo tiempo de rem_2. Para evitar

esto se han dado algunas soluciones, una es utilizar un multiconjunto intermedio y una o más reglas para restaurar rem_2 a su estado original.

$$\begin{aligned} paso_{2a} \;=\; & (I, C_r, T_r) : rem_2, T : prev \\ & \rightarrow (I, C_r, T_r) : list_2, T : prev \\ & \Leftarrow T_r > T \end{aligned}$$

Para el segundo subpaso no necesitamos una regla de reescritura porque el multiconjunto loc_2 ya contiene todos los documentos de la base de datos local. En otras palabras $paso_{2b}$ es vacío. Como un resultado de esto la especificación completa del segundo paso consiste de la ejecución secuencial del subpaso $paso_{2b}$ después del subpaso $paso_{2a}$, la cual puede reducirse al primer subpaso solo:

$$paso_2 \equiv (paso_{2b} \circ paso_{2a}) \equiv paso_{2a}$$

5.2.3. Tercer paso

La especificación informal del tercer paso realiza la copia de los nuevos documentos de la base de datos remota a la réplica local. Este compara el indicador de tiempo de ambas versiones y contiene subpasos para cada uno de los posibles casos.

3. *Replicar documentos listados:*

   Para cada elemento en la lista local, encontrar el elemento correspondiente en la lista remota.

   a) Si el documento en la base de datos es una versión más nueva que la versión en la base de datos local, entonces se copia el documento a la base de datos local.

   b) Si el documento en la base de datos remota esta marcado como borrado el documento es borrado en la base de datos local.

   c) Si el documento en la base de datos remota es más viejo que la versión en la base de datos local o el documento no está en la base de datos remota, entonces no se hace nada hasta que la réplica remota copie el documento desde la base de datos local.

d) Para el resto de documentos en la lista de la base de datos remota, se copia este a la base de datos local mientras halla nuevos documentos.

Para determinar cuándo una versión remota es más nueva o más vieja que la versión local comparamos sus respectivos indicadores de tiempo T_r y T_l. En el subpaso $paso_{3a}$ sobreescribimos la versión local en loc_2, mientras que en el caso complementario el subpaso $paso_{3c}$, mantenemos la versión local.

$$\begin{aligned} paso_{3a} &= (I, \_, T_l) : loc_2,\ (I, C_r, T_r) : list_2 \\ &\quad \rightarrow (I, C_r, T_r) : loc_2 \\ &\quad \Leftarrow T_r > T_l \\ paso_{3c} &= (I, C_l, T_l) : loc_2,\ (I, \_, T_r) : list_2 \\ &\quad \rightarrow (I, C_l, T_l) : loc_2 \\ &\quad \Leftarrow T_r < T_l \end{aligned}$$

Es importante notar que utilizamos el guión bajo (_) como un elemento en la regla de reescritura que indica que no importa dicho elemento, es decir, la sustitución de dicha variable es irrelevante para el resto de la regla.

En el caso del subpaso $paso_{3b}$ identificamos a un documento como borrado si su indicador de tiempo tiene el valor *del*.

$$paso_{3b} = (I, C_l, T_l) : loc_2,\ (I, \_, T_r) : list_2 \Leftarrow T_r = del$$

Finalmente en el último caso cubrimos el resto de posibilidades, se aplica a todo el resto de los elementos de $list_2$. Cuando $paso_{3d}$ no se puede aplicar más, $list_2$ será vacío.

$$paso_{3d} = (I, C_r, T_r) : list_2 \rightarrow (I, C_r, T_r) : loc_2$$

De esta forma el paso tres está dado por la composición secuencial de los cuatro subpasos.

$$paso_3 = paso_{3d} \circ paso_{3c} \circ paso_{3b} \circ paso_{3a}$$

5.2.4. Algoritmo completo

La estructura jerárquica de objetos, con las bases de datos en la parte superior y los documentos en la parte inferior, indican la estructura del algoritmo de replicación. El primer paso opera al nivel más alto, mientras que el segundo y tercer paso están situados en el otro. El concepto de configuración nos ayuda a modelar la jerarquía.

Entonces los tres multiconjuntos (loc_1, rem_1 y $list_1$) forman la estructura del nivel superior de nuestra especificación. $prog_1$ es la especificación del programa:

$$nivel_1 = [prog_1,\ loc_1,\ rem_1,\ list_1]$$

Siguiendo esta idea, el nivel dos en la configuración reúne los multiconjuntos y reglas relacionadas con el nivel más bajo:

$$nivel_2 = [prog_2,\ id,\ loc_2,\ rem_2,\ prev,\ rem_2,\ pres,\ list_2]$$

La unión entre los dos niveles esta hecha por la inclusión jerárquica de configuraciones, como se muestre por la modificación del $paso_1$:

$$\begin{aligned} paso_1 \ = \ & (I, D_l, T_l) : loc_1, (I, D_r, T_r) : rem_1 \\ & \rightarrow [prog_2, I, D_l, T_l, D_r, T_r, \emptyset] : list_1, (I, D_r, T_r) : rem_1 \\ & \Leftarrow T_r > T_l \end{aligned}$$

El programa para el nivel inferior $prog_2$, está dado por la composición secuencial de $paso_2$ y $paso_3$:

$$prog_2 = paso_3 \circ paso_2$$

El programa del nivel superior $prog_1$ consiste de $paso_1$ seguido de la regla que recupera las configuraciones del nivel inferior cuando se vuelven pasivas. Esta no tiene una tarea en la descripción informal, pues ésta no indica explícitamente que el segundo y el tercer paso del algoritmo deben de ser repetidos para cada base de datos seleccionada por el primer paso:

$$prog_1 = paso_4 \circ paso_1$$

donde:

$$\begin{aligned} paso_4 \;=\; & [I, D_l, \_, \_, T_r, \emptyset] : list_1 \\ & (I, D_r, T_r) : loc1 \end{aligned}$$

En el caso cuando una computadora local se replica con más de una computadora remota (es decir, su contenido se ha replicará con más de una máquina), las réplicas de las bases de datos (o versiones de documentos) no pueden ser identificadas únicamente por un indicador de tiempo sencillo. El sistema Notes utiliza tablas de replicación que contienen un indicador de tiempo por cada computadora remota. El efecto de esta complicación sobre el algoritmo de replicación es mínimo. Cuando un indicador de tiempo es requerido, este es seleccionado de un multiconjunto que modela la tabla de replicación (usando el identificador de la computadora remota).

En algunas versiones de Notes los documentos son identificados por un indicador de tiempo y número de versión. El número de versión es necesario para resolver los conflictos que puedan ocurrir cuando los relojes de varias computadoras del sistema distribuido no estén suficientemente sincronizados.

5.3. Especificación formal vs. especificación informal

En esta sección presentaremos algunas razones por las cuales es preferible utilizar una especificación formal que una especificación informal.

5.3.1. Detección de errores

Consideremos el tercer paso del algoritmo. En el caso donde la réplica remota y local tengan la misma versión de una base de datos, según el algoritmo no podemos aplicar el primer subcaso ($paso_{3a}$) pues el documento no es más viejo, ni tampoco el segundo ($paso_{3c}$) ya que el documento no es más nuevo y el documento no ha sido marcado como borrado ($paso_{3b}$). Así que recurrimos al último caso ($paso_{3d}$) que

considera las demás posibilidades, por lo tanto debemos considerar al documento como nuevo, y entonces copiarlo a la réplica local. Pero el documento no es nuevo, y ya existe en la réplica local. De esta forma después de una replicación, la réplica local contiene dos copias del mismo documento, luego de n replicaciones, contendrá $n + 1$ copias.

En la especificación formal del tercer paso es necesario hacer una pequeña corrección para no omitir este caso, esto lo haremos modificando a la regla $paso_{3c}$ de la siguiente manera:

$$\begin{aligned} paso_{3c} \;=\; & (I, C_l, T_l) : loc_2,\ (I, \_, T_r) : list_2 \\ & \rightarrow (I, C_l, T_l) : loc_2 \\ & \Leftarrow T_r \leq T_l \end{aligned}$$

Aquí encontramos una ventaja de la especificación formal sobre la otra, pues al escribir una especificación en lenguaje natural, se presentan términos ambiguos como "más viejo" o "más nuevo", mientras que en lenguaje matemático tenemos "mayor que" o "mayor o igual que" los cuales no presentan ambigüedad.

Otro término de este tipo es "para cada", pero este ya fue considerado en las reglas anteriores, con el hecho de que la regla se aplica mientras halla elementos que cumplan la condición especificada por la regla.

5.3.2. Eliminación de listas redundantes

Para cada nivel de la especificación informal las listas fueron utilizadas para contener los conjuntos de datos intermedios. Sin embargo no se indica en la especificación por qué una lista es la estructura más apropiada para este caso. Los conjuntos de datos intermedios no están ordenados. Tampoco hay alguna otra relación entre sus elementos que justifique la linealidad que impone una lista. Por estas razones para la especificación formal es mejor utilizar un tipo de contenedor menos restrictivo: los multiconjuntos.

Analizando las reglas del programa, encontramos que la estructura de lista es redundante. En las reglas, por ejemplo, los elementos que fueron

puestos en $list_1$ en la regla $paso_1$ pueden ser puestos directamente en loc_1. Realizando estos cambios las reglas quedan de al siguiente forma:

$$\begin{aligned} paso_1 \;=\; & (I, D_l, T_l) : loc_1,\ (I, D_r, T_r) : rem \\ & \rightarrow [paso_3 \circ paso_2,\ I,\ D_l,\ T_l,\ D_r,\ T_r] : loc_1, \\ & \quad (I, D_r, T_r) : rem_1 \\ & \Leftarrow T_r > T_l \end{aligned}$$

$$paso_4 \;=\; (I,\ D_l, \_, \_, T_r, \emptyset) : loc_1 \rightarrow (I, D_r, T_r) : loc_1$$

Bajo esta idea, es posible eliminar $list_2$. Los elementos que originalmente se encontraban en $list_2$ serán colocados en rem_2 solo que tendrán un valor reservado $flit$. Al mismo tiempo esto ayuda a recuperar el contenido original de rem_2. Haciendo estas adaptaciones a las reglas $paso_{3a}$ y $paso_{3d}$ tenemos:

$$\begin{aligned} paso_{3a} \;=\; & (I, \_, T_l) : loc_2,\ (filt, I, C_r, T_r) : rem_2 \\ & \rightarrow (I, C_r, T_r) : loc_2, (I, C_r, T_r) : rem_2 \\ & \Leftarrow T_r > T_l \end{aligned}$$

$$\begin{aligned} paso_{3c} \;=\; & (filt, I, C_r, T_r) : rem_2 \\ & \rightarrow (I, C_r, T_r) : loc_2, (flit, I, C_r, T_r) : rem_2 \end{aligned}$$

5.3.3. Paralelismo

Las listas son solo un ejemplo de la secuencialidad impuesta por la especificación informal del algoritmo de replicación. La estructura de dicha especificación es otro ejemplo.

Pero la secuencialidad impuesta por la especificación informal puede ser reducida. En el nivel más alto vemos que el programa $prog_2$ puede copiar documentos entre dos réplicas de la misma base de datos, mientras $prog_1$ busca en las bases de datos los documentos con diferentes versiones.

En el nivel inferior, dentro de $prog_2$, podemos realizar en paralelo $paso_3$ y $paso_2$ para que los documentos de diferentes bases de datos

sean copiados en paralelo.

$$prog_2 = (paso_3 + paso_1)$$

Un refinamiento final introduce el paralelismo entre los subpasos de $paso_3$, ya que la ejecución de los tres casos es irrelevante y solo el $paso_{3d}$ debe ser ejecutado después de las otras reglas:

$$paso_3 = (paso_{3d} \circ (paso_{3c} + paso_{3b} + paso_{3a}))$$

Así con estos refinamientos hemos obtenido una versión más eficiente del algoritmo de replicación.

De esta forma hemos visto algunas de las claras ventajas de una especificación formal respecto a una especificación informal, ya que una especificación formal nos brinda la posibilidad de analizar todos los posibles casos (si han sido bien especificados, cada uno de ellos). También nos permite realizar algunas mejoras al programa especificado.

Capítulo 6

Confiabilidad de las bases de datos distribuidas

En el contexto del desarrollo de software los términos "confiabilidad" y "disponibilidad" de un sistema son mencionados muchas veces sin definirlos precisamente. En este capítulo hablaremos de estos términos junto con la replicación de datos. Al referirnos a los sistemas administradores de bases de datos distribuidas, debemos hacer notar que la distribución de la base de datos y la replicación de los datos no son suficientes para hacer que estos sistemas sean confiables, para esto es necesario implementar un número de protocolos dentro de las SMBD (sistemas manejadores de bases de datos) que, basándose en la distribución y replicación, hagan más confiables sus operaciones.

Diremos que un sistema administrador de bases de datos es confiable si puede procesar las peticiones de los usuarios aun cuando el sistema no sea en sí confiable. En otras palabras, aun cuando los componentes del ambiente distribuido fallen, un SMBD distribuidas debe ser capaz de continuar ejecutando las peticiones del usuario sin violar la consistencia de la base de datos. En esta sección nos referiremos a la confiabilidad respecto a las propiedades de atomicidad y durabilidad de las transacciones, considerando los protocolos de *commit* y recuperación.

6.1. Fallas en los SMBD distribuidas

En general podemos decir que tenemos cuatro tipo de fallas en una base de datos: fallas en las transacciones (aborts), fallas en el sitio (sitio), fallas en el medio (disco) y fallas en las líneas de comunicación.

6.1.1. Fallas en las transacciones

Las transacciones pueden fallar por varias razones. Una de ellas puede ser un dato de entrada incorrecto. Dado que algunos algoritmos de control de concurrencia no permiten que una transacción pueda tener acceso a los datos si otra transacción está haciendo lo mismo, esto es considerado otra falla. El más usual de los casos es cuando una transacción indica la operación de abortar, lo cual indica que las base de datos debe regresar al estado previo antes de iniciar la transacción.

6.1.2. Fallas en el sitio

Las razones por las cuales un sistema falla pueden ser de hardware (procesador, memoria principal, energía, etc) o de software (errores en el sistema operativo o en el código del manejador de la base de datos). Así, una parte de la base de datos que se encuentre en los buffers de la memoria principal se pierde como resultado de una falla en el sistema. Pero como la base de datos está almacenada en un medio secundario, supondremos que esta es segura y correcta. En los sistemas distribuidos las fallas en el sistema son referidas como fallas del sitio y al producirse un falla del sitio éste se vuelve inalcanzable por otros sitios en el sistema distribuido.

Este tipo de fallas es clasificado en dos tipos: un falla total, cuando se producen fallas simultáneas en todos los sitios del sistema distribuido; y las fallas parciales, que indican fallas en algunos sitios mientras que otros permanecen en operación.

6.1.3. Fallas en los medios

Las fallas ocurren en los medios de almacenamiento secundario de la base de datos. Tales fallas pueden deberse a errores en el sistema operativo o en el hardware. El punto importante en este tipo de errores es que parte o toda la base de datos almacenada en el medio secundario es destruida y se vuelve inaccesible.

Las técnicas más comunes para evitar problemas con este tipo de errores es la realización de copias de seguridad. Estas fallas son tratadas como problemas locales en un sitio.

6.1.4. Fallas de comunicación

Los casos anteriores se presentan en los sistemas centralizados y en los distribuidos, pero las fallas de comunicación son solo para los últimos. La fallas más comunes son los errores en los mensajes, un orden incorrecto en los mensajes, pérdida de mensajes y fallas en las líneas. La pérdida de mensajes es generalmente consecuencia de fallas en las líneas de comunicación o de fallas del sitio.

6.2. Protocolos de confiabilidad

La finalidad principal de los protocolos de confiabilidad es el mantener la atomicidad y durabilidad de las transacciones distribuidas que se ejecutan sobre un número de bases de datos. Los protocolos dirigen la ejecución distribuida de los comandos *begin_transaction*, *read*, *write*, *abort*, *commit* y *recover*.

Los comandos de *begin_transaction*, *read* y *write* no causan grandes problemas; el comando *begin_transaction* solo indica como el manejador de la transacción al sitio donde se originó la transacción. El *read* y *write* son ejecutados de acuerdo a las reglas clásicas, es decir, permitiendo solo una escritura a la vez y la posibilidad de varias lecturas. Para una explicación más completa revísese Tamer (1998).

Cada uno de los comandos será ejecutado en cada uno de los sitios. Por facilidad consideraremos que el sitio de donde se origina la transacción tiene un proceso llamado *coordinador* que ejecuta su operación. El

coordinador se comunica con todos los procesos *participantes* en otros sitios que colaboran con la ejecución de la transacción.

Las técnicas de confiabilidad en los sistemas de bases de datos distribuidas consisten de protocolos *commit*, terminación y recuperación. Los protocolos de commit y recuperación especifican cómo los comandos de commit y recuperación (recover) son ejecutados. Asumimos que durante la ejecución de una transacción distribuida, si un sitio involucrado en la ejecución falla, hay otro sitio que puede intentar terminar dicha transacción.

El principal requerimiento de los protocolos de commit es el de mantener la atomicidad de las transacciones distribuidas, esto es, cuando una transacción involucra múltiples sitios, todos los sitios deben realizar las operaciones apropiadas o ninguno de los cambios será permanente.

6.3. Protocolo commit de dos Fases

El protocolo commit de dos fases asegura la atomicidad del commit de las transacciones distribuidas. Extiende los efectos de las acciones locales del commit a todos los sitios involucrados en la aceptación de la ejecución del commit antes de que los cambios sean permanentes.

Inicialmente el coordinador escribe un *begin_commit* en sus registros (logs), envía un mensaje "prepare" a todos los sitios participantes, y entra al estado de **espera**. Cuando un participante recibe un mensaje "prepare", verifica si pudo realizar la transacción. Si fue así, el participante escribe *ready* en sus registros indicando que está listo, y envía un mensaje "vote-commit" al coordinador, y entra al estado de **listo**; en otro caso el participante escribe *abort* en sus registros y envía un mensaje "vote-abort" al coordinador. Si la decisión del sitio es abortar, debe olvidarse de la transacción, pero esta decisión solo sirve como un veto, es decir, es un aborto solo para el sitio. Después de que el coordinador ha recibido una respuesta de todos los participantes decide cuando realizar una acción de commit o de aborto. Si algún participante indica un voto a favor de aborto, el coordinador indicará un aborto para la transacción globalmente, para esto escribe *abort* en sus registros y envía un el mensaje de "global-abort" a todos los sitios participantes, y entra

a un estado de *aborto*, en otro caso, escribe *commit* en sus registros, envía el mensaje "global-commit" de todos los participantes, y entra al estado de **commit**. Los participantes realizan el commit o el aborto de acuerdo a la instrucción del coordinador y envía de regreso una señal de realizado, después de lo cual el coordinador termina la transacción escribiendo un *end_of_transaction* en sus registros.

Las reglas del *commit global* son:

- Si alguno de los participantes vota por el aborto de la transacción, el coordinador toma una decisión de aborto global.
- Si cada uno de los participantes vota por el commit de la transacción, el coordinador toma una decisión de commit global.

El algoritmo 2PC del lado del coordinador es el algoritmo 1. El algoritmo 2PC del lado de los participantes es el algoritmo 2

Especificaremos el comportamiento de los algoritmos anteriores en términos de programas de Gamma de orden superior, para verificar algunas propiedades del algoritmo. No consideraremos los mensajes de *timeout*, así que no es necesario llevar un control del tiempo, tampoco son relevantes las escrituras en los registros (logs).

Los siguientes programas definen el algoritmo 2PC del lado del coordinador: el programa $\mathbf{commit_c}$ será usado cuando el coordinador recibe el mensaje de *commit* de alguno de los sitios participantes de la transacción. En este caso enviaremos a todos los sitios el mensaje de *prepare*.

$$
\begin{aligned}
commit_c \;=\; & (Id_s, Msg\_In_s, Msg\_Out_s) : PL, \\
& (Id_d, Msg\_In_d, Msg\_Out_d) : PL, \\
& global\_dec : STATUS \\
& \quad \rightarrow (Id_s, Msg\_In_s, Msg\_Out_s) : PL, \\
& \quad\;\; (Id_d, \text{“}prepare\text{”}, Msg\_Out_d) : PL, \\
& \quad\;\; \text{“}no\text{-}decision\text{”} : STATUS \\
& \quad \Leftarrow Msg\_Out_s = \text{“}commit\text{”} \\
& \quad\;\; \wedge\; Msg\_In_d = \mathbf{null} \\
& \quad\;\; \wedge\; Msg\_Out_d = \mathbf{null}
\end{aligned}
$$

Algoritmo 1 Algoritmo 2PC-coordinador

```
msg : Mensaje
ev : Evento
PL Lista de Participantes
inicio
  WAIT(ev)
  si ev = Mensaje_Recibido entonces
    msg es el mensaje que llega
    si msg = Commit entonces
      escribir begin_commit en el registro
      enviar el mensaje "prepare" a todos los participantes en PL
      iniciar el timer
    de lo contrario si msg = Vote_Abort entonces
      escribir abort en el registro
      enviar el mensaje "global-abort" a todos los participantes en
      PL
      iniciar el timer
    de lo contrario si msg = Vote_Commit entonces
      actualiza la lista de participantes que han respondido
      si todos los participantes han respondido entonces
        escribir commit en el registro
        enviar el mensaje "global-commit" a todos los participantes
        en PL
        iniciar el timer
      fin
    de lo contrario si msg = Ack entonces
      actualiza la lista de todos los participantes que han enviado el
      mensaje ack
      si todos los participantes han respondido ack entonces
        escribir end_of_transaction en el registro
      de lo contrario
        enviar la decisión global a los participantes que no han res-
        pondido
      fin
    fin
  de lo contrario si ev = TimeOut entonces
    ejecuta el protocolo de terminación
  fin
```

Algoritmo 2 Algoritmo 2PC-participantes

```
  msg : Mensaje
  ev : Evento
inicio
  WAIT(ev)
  si ev = Mensaje_Recibido entonces
    msg es el mensaje que llega
    si msg = Prepared entonces

      si listo para el commit entonces
        escribir ready en el registro
        enviar el mensaje "vote-commit" al coordinador
        iniciar el timer
      de lo contrario
        escribir abort en el registro
        enviar el mensaje "vote-abort" al coordinador
        iniciar el timer
      fin

    de lo contrario si msg = Global_Abort entonces
      escribir abort en el registro
      llamar al proceso local de abortar la transacción
    fin

  de lo contrario si ev = TimeOut entonces
    ejecuta el protocolo de terminación
  fin
fin
```

el programa **vote_abort$_c$** es ejecutado cuando el coordinador recibe el mensaje de *vote_commit* de alguno de los sitios participantes de la transacción, entonces el coordinador enviará a todos los sitios el mensaje *global_abort*.

$$
\begin{aligned}
vote\_abort_c \;=\; & (Id_s, Msg\_In_s, Msg\_Out_s) : PL, \\
& (Id_d, Msg\_In_d, Msg\_Out_d) : PL, \\
& global\_dec : STATUS \\
& \quad \rightarrow (Id_s, Msg\_In_s, Msg\_Out_s) : PL, \\
& \quad\; (Id_d, \text{“global\_abort”}, Msg\_Out_d) : PL, \\
& \quad\; \text{“abort”} : STATUS \\
& \quad \Leftarrow Msg\_Out_s = \text{“global\_abort”} \\
& \quad\; \wedge\; Msg\_In_d = \texttt{null} \\
& \quad\; \wedge\; Msg\_Out_d = \texttt{null}
\end{aligned}
$$

el programa **vote_commit$_c$** se ejecuta cuando el coordinador recibe el mensaje de *vote_commit* de alguno de los sitios participantes de la transacción y enviará a el mensaje de "global_commit" si todos los participantes han votado por el "commit".

$$
vote\_commit_c \;=\; vote\_commit_{c_1} \circ vote\_commit_{c_2}
$$

$$
\begin{aligned}
vote\_commit_{c_1} \;=\; & ((Id_s, Msg\_In_s, Msg\_Out_s) : PL \\
& \quad \rightarrow Id_s : PL_{vc}, (Id_s, \texttt{null}, Msg\_Out_s) : PL \\
& \quad \Leftarrow Msg\_In_s = \text{“vote\_commit”} \\
& \quad\; \wedge\; Msg\_Out_s = \texttt{null} \\
& \quad\; \wedge\; Id_s \notin PL_{vc}) \\
vote\_commit_{c_2} \;=\; & ((Id_s, Msg\_In_s, Msg\_Out_s) : PL, \\
& global\_dec : STATUS \\
& \quad \rightarrow (Id_s, \text{“global\_commit”}, Msg\_Out_s) : PL, \\
& \quad\; \text{“commit”} : STATUS \\
& \quad \Leftarrow Msg\_Out_s = \texttt{null} \\
& \quad\; \wedge\; |PL_{vc}| = |PL|)
\end{aligned}
$$

el programa $\mathbf{ack_c}$ es ejecutado cuando el coordinador recibe el mensaje de *ack* de alguno de los sitios participantes

$$
\begin{aligned}
ack_c &= ack_{c_1} \circ ack_{c_2} \circ ack_{c_3} \\
\\
ack_{c_1} &= ((Id_s, Msg\_In_s, Msg\_Out_s) : PL \\
&\quad \rightarrow Id_s : PL_{ack}, (Id_s, Msg\_In_s, Msg\_Out_s) : PL \\
&\quad \Leftarrow Msg\_Out_s = \text{“}ack\text{”} \\
&\quad\quad \wedge Id_s \notin PL_{vc}) \\
ack_{c_2} &= ((Id_s, Msg\_In_s, Msg\_Out_s) : PL, global\_dec : STATUS \\
&\quad \rightarrow (Id_s, Msg\_In_s, Msg\_Out_s) : PL, \\
&\quad\quad global\_dec : STATUS \\
&\quad \Leftarrow Msg\_In_s = \text{“}ack\text{”} \\
&\quad\quad \wedge Msg\_Out = \texttt{null} \\
&\quad\quad \wedge |PL_{ack}| = |PL|) \\
ack_{c_3} &= ((Id_s, Msg\_In_s, Msg\_Out_s) : PL, global\_dec : STATUS \\
&\quad \rightarrow (Id_s, global\_dec, Msg\_Out_s) : PL, \\
&\quad\quad global\_dec : STATUS \\
&\quad \Leftarrow Msg\_In_s = \texttt{null} \\
&\quad\quad \wedge Id_s \in PL_{ack})
\end{aligned}
$$

Estos programas definen casi completamente el comportamiento del algoritmo 2PC. Solo falta simular el esperar hasta que llegue un mensaje de algún sitio, esto lo podemos hacer con el siguiente programa :

$$
\begin{aligned}
wait &= (Id, Msg\_In, Msg\_Out) : PL \\
&\quad \rightarrow (Id_s, Msg\_In_s, Msg\_Out_s) : PL \\
&\quad \Leftarrow verdadero
\end{aligned}
$$

De esta forma el algoritmo 2PC está definido por la composición paralela de estos programas

$$2pc_c = commit_c + vote\_abort_c + vote\_commit_c + ack + wait$$

El algoritmo 2PC del lado del participante esta definido por la composición paralela de los programas

$$2pc_p \;=\; prepared_p + global\_abort_p + wait$$

donde los programas $\mathbf{prepared_p}$ y $\mathbf{global\_abort_p}$ son: el programa $\mathbf{prepared_p}$ es activado cuando el coordinador envía al participante el mensaje de "prepared".

$$prepared_p \;=\; prepared_{p_1} \circ prepared_{p_2}$$

$$\begin{aligned} prepared_{p_1} \;=\; & ((Id_s, Msg\_In_s, Msg\_Out_s) : PL \\ & \rightarrow (Id_s, \mathtt{null}, \text{"}vote\_commit\text{"}) : PL, \\ & \Leftarrow Msg\_In_s = \text{"}prepare\text{"} \\ & \quad \wedge\, Msg\_Out_s = \mathtt{null}) \\ prepared_{p_2} \;=\; & ((Id_s, Msg\_In_s, Msg\_Out_s) : PL \\ & \rightarrow (Id_s, \mathtt{null}, \text{"}vote\_abort\text{"}) : PL, \\ & \Leftarrow Msg\_In_s = \text{"}prepare\text{"} \\ & \quad \wedge\, Msg\_Out_s = \mathtt{null}) \end{aligned}$$

y el programa $\mathbf{global\_abort_p}$ está definido como:

$$\begin{aligned} global\_abort_p \;=\; & (Id_s, Msg\_In_s, Msg\_Out_s) : PL \\ & \rightarrow (Id_s, \mathtt{null}, \text{"}abort\text{"}) : PL, \\ & \Leftarrow Msg\_In_s = \text{"}global\_abor\text{"} \\ & \quad \wedge\, Msg\_Out_s = \mathtt{null} \end{aligned}$$

En el algoritmo original, no esta completamente clara la forma en que interactúan los algoritmos, así como tampoco el estado en el que se encuentra el coordinador y los participantes entre cada mensaje que se envía o se recibe, pero mediante los programas en Gamma de orden superior que definen dichos algoritmos ($2pc_c$ y $2pc_p$ respectivamente) se define completamente este comportamiento, ya que $2pc_c$ es la composición de los programas que definen cada uno de los casos para los diferentes mensajes y el programa infinito *wait*, el cual permite que el

programa esté listo para recibir nuevos mensajes. Esto mismo se aplica para $2pc_p$.

También en la versión de Gamma de orden superior es más clara la forma en que interactúan los dos programas, pues ambos toman como base a los mismos multiconjuntos y los van modificando, de esta manera los programas se pueden ir comunicando.

Un punto importante en el protocolo es cuando el coordinador o los participantes pueden recibir ciertos mensajes, es decir, el coordinador no puede recibir un mensaje de voto por aborto si no se encuentra en el estado inicial, esto no es muy evidente en los algoritmos propuestos pues solo se indica, para un un mensaje recibido, qué mensajes se deben enviar y a quiénes, sin considerar nunca el estado en que se pueda encontrar el receptor del mensaje. En las reglas de Gamma de orden superior, el estado de los participantes está determinado por los posibles mensajes que recibe y envía, mediante la verificación de estos valores en las condiciones de la reacción podemos asegurar que un participante solo recibe un mensaje en el estado apropiado.

Finalmente para mostrar que el algoritmo funciona debemos ver que el comportamiento del algoritmo refleja el comportamiento descrito al principio de esta sección, para esto es necesario verificar que los mensajes son enviados a los participantes apropiados y que se realizan las acciones apropiadas para cada tipo de mensaje recibido.

El programa $commit_c$ envía a todos los participantes el mensaje de "prepare" pues el multiconjunto PL contiene a todos los participantes con `null` como mensajes que recibe y envía (Msg_In y Msg_Out). De esta forma cuando un participante genere el mensaje de "commit" el resto de los participantes cumplirá la condición de la reacción del programa. Esto mismo puede ser aplicado para el programa $vote\_abort_c$.

Otra importante característica del algoritmo es que un participante no puede cambiar de opinión, es decir, una vez tomado la decisión del votar por un commit o un aborto no puede cambiar, esto se ve reflejado en los programas pues para que esto pasara un participante debería recibir el mensaje de "prepare" en más de una ocasión y entonces tomar una nueva decisión. Pero esto no puede suceder ya que una vez que un participante cumple las condiciones de reacción del programa $commit_c$ contiene siempre un mensaje que se envía o se recibe y entonces no

podrá volver a cumplir dicha condición.

También quedan especificadas las reglas del *commit global*, pues solo enviará a todos los participantes dicho mensaje cuando halla recibido de todos los participantes el voto a favor del commit, esto lo indica el programa $vote\_commit_c$ pues solo enviará el mensaje cuando el número de elementos en el multiconjunto de participantes que votaron por commit sea igual al número de elementos en el multiconjunto de todos los participantes. Y enviará el mensaje de aborto global cuando uno vote por el aborte ($vote\_abort_c$).

De esta forma hemos presentado un algoritmo para el protocolo de dos faces en Gamma de orden superior, del cual podemos verificar que cumple las condiciones especificadas por el algoritmo y esta verificación se puede hacer de una forma más clara que en un algoritmo en pseudocódigo. En el breve análisis anterior hemos tratado la verificación de las principales propiedades del algoritmo. En él se indica como las propiedades se cumplirán al ir aplicando las reglas de reescritura y también se menciona que esta versión es más eficiente que la de pseudocódigo, ya que el hecho de utilizar Gamma de orden superior hace que el algoritmo se pueda ejecutar en paralelo, lo que en principio mejoraría su tiempo de ejecución.

Capítulo 7

Conclusiones

A lo largo de la tesis se presentaron las bases teóricas necesarias para el estudio del modelo de Gamma. Partimos de la sintaxis y semántica de un lenguaje imperativo IMP. La sintaxis utilizada para IMP se basa en reglas que describen la forma en que son evaluadas sus expresiones y son ejecutados sus comandos, por lo tanto estas reglas producen una semántica operacional y esa es la semántica que utilizamos para describir el comportamiento de IMP.

Una vez descrita la sintaxis y semántica de IMP, definimos el comportamiento de un programa determinista y dimos el paso hacia los programas no deterministas y los lenguajes paralelos. Para esto extendimos el lenguaje IMP con una operación paralela, pero con esto surge el problema de las asignaciones sobre una misma variable, es decir, que la asignación llevada a cabo por un programa puede afectar el estado en el que termina la ejecución de la composición de los programas. Esto impide describir el modelo de ejecución de comandos en paralelo como se hace con el resto de los comandos, a saber, usando una relación entre configuraciones de comandos y estados.

Para el estudio del paralelismo, consideramos el modelo propuesto por Dijkstra que se basa en el uso de comandos resguardados. En este modelo la idea básica es considerar la composición paralela como la interacción no determinista de acciones atómicas de los componentes. Existen dos modelos básicos para este enfoque: *comandos resguardados* y *procesos comunicados*. El primero está basado en el uso de memoria compartida y el segundo en un mecanismo de envío de mensajes entre

programas para indicar el intercambio de valores.

Con el análisis de estos modelos obtuvimos una idea global del comportamiento de los lenguajes paralelos. De esta forma continuamos con el estudio de un caso particular de los lenguajes paralelos: Gamma. Se presentó el modelo de Gamma, desarrollado principalmente por Banâtre y Le Métayer [1], para el cual Hernández [7] presentó una semántica denotacional.

Así tomando como referencia a Gamma, presentamos Gamma de orden superior. Gamma de orden superior fue el punto central del desarrollo de la tesis y todos los resultados obtenidos se basan en él. Al igual que Gamma, Gamma de orden superior maneja los multiconjuntos como única estructura de datos lo cual no le resta poder funcional, pero si puede hacer más difícil su comprensión y por tanto limita la aceptación del lenguaje. Este problema fue atacado para Gamma por Fradet y Le Métayer [6], quienes propusieron Gamma estructurado como una solución a este problema.

Uno de los primeros resultados de esta tesis fue adaptar el modelo de Gamma estructurado a Gamma de orden superior. Con éste nuevo modelo obtuvimos un modelo de Gamma de orden superior que nos permite desarrollar programas de manera más sencilla y clara, ya que no necesitamos hacer alguna codificación adicional para los datos que manejará el programa, sino que debemos definir un nuevo tipo de dato que refleje el comportamiento de los datos a manejar. Una ventaja más de este modelo es que para la definición de los tipos de datos se utilizan reglas de reescritura, lo cual permite de manera muy natural la definición recursiva de tipos; el uso de reglas de reescritura también hace que la definición de tipos sea muy similar a la definición de un programa en Gamma. De esta manera los tipos son consistentes con los programas y la idea original de Gamma (y por supuesto Gamma de orden superior) se conserva.

Pero este modelo no es del todo completo ya que, aunque los estudios sobre Gamma estructurado muestran que los programas no modifican el tipo de los multiconjuntos estructurados no hay un algoritmos que pueda asegurar siempre esto, por lo tanto este problema también es heredado por nuestra versión de Gamma de orden superior.

Presentamos una aplicación de Gamma de orden superior, utilizándo-

lo como un lenguaje para la especificación formal de un algoritmo, en este caso el algoritmo de replicación de objetos de Lotus Notes. Esta sección se basó en el artículo de Bourgois [3]. Él hizo una comparación de la especificación propuesta con respecto a la especificación informal existente del algoritmo, aquí se analizarón sus comparaciones y se extendieron los puntos que dejó inconclusos. Esto nos arrojó algunos resultados, uno de los más importantes es que nos liberamos del problema de ambigüedad del lenguaje natural, lo que podría propiciar un funcionamiento del algoritmo distinto al esperado.

Podemos ver que la estructura del algoritmo de replicación se asemeja a una estructura de objetos, donde las bases de datos están en la parte superior y los documentos en la parte inferior, de tal forma que parte del algoritmo (primer paso) solo opera en la parte superior y el resto (segundo y tercer paso) en la parte inferior. Aquí se nota la importancia de utilizar Gamma de orden superior pues el concepto de configuración nos ayudó a modelar esta jerarquía. Esta idea (el modelado de las jerarquía por medio de las configuraciones) junto con la posibilidad de la composición (secuencial y paralela) de programas, nos llevan a pensar que Gamma de orden superior es un medio útil para modelar sistemas basados en objetos (Bourgois [3]). A primera vista esto nos parecería una propuesta muy ambiciosa, ya que uno de los paradigmas más utilizados en la actualidad es el orientado a objetos. Entonces la posibilidad de modelar estos sistemas con Gamma de orden superior (que además nos permite el procesamiento en paralelo) indicaría que en un futuro Gamma de orden superior sería una muy buena opción como lenguaje de programación. Pero aún falta por estudiar más a fondo este tema pues hay algunos aspectos de la orientación a objetos aún no se pueden modelar de forma directa que en Gamma de orden superior.

Las especificaciones formales también permiten hacer refinamientos a los algoritmos sin alterar el funcionamiento básico de los mismos. Esta característica no podía dejar de ser usada aquí, de tal forma que se plantean algunas modificaciones a la primera versión de la especificación del algoritmo de replicación de Lotus Notes. Entre las mejoras que se hicieron tenemos: la consideración de algunas condiciones extras para evitar el duplicado de copias en las máquinas, la detección y la eliminación de listas redundantes en las reglas del algoritmo, y una de

las más importantes: el análisis de los subprogramas que modelan el algoritmo para encontrar los que se puedan ejecutar en paralelo.

La otra aplicación es la versión en Gamma de orden superior del protocolo commit de dos fases (2PC). Como se indicó, la finalidad de este protocolo es mantener la atomicidad y durabilidad de las transacciones distribuidas que se ejecutan sobre diversas de bases de datos. Para el estudio de este problema es importante conocer algunas de las fallas que se presentan comunmente en la bases de datos distribuidas. Por esta razón en la sección 6 se habla brevemente de las bases de datos distribuidas, sus fallas y la confiabilidad en bases de datos distribuidas.

Con esto en mente, abordamos el protocolo de confiabilidad 2PC del cual Tamer ([9]) presenta en una versión semiformal. Luego de explicar el comportamiento de dicho protocolo y mostrarlo en pseudocódigo planteamos una versión de él en Gamma de orden superior. En este protocolo intervienen un coordinador y los participantes, por lo que se requieren dos algoritmos, uno que corre del lado del coordinador y otro que corre del lado los participantes.

Aunque el algoritmo en pseudocódigo muestra un comportamiento que necesariamente debiera ser secuencial, y por el hecho de que en Gamma y Gamma de orden superior es más difícil escribir programas secuenciales que paralelos, parecería que un algoritmo en Gamma de orden superior, más que ser útil para identificar algunas características del algoritmo, sería un conjunto de reglas confusas y rebuscadas. Sin embargo, pudimos definir un conjunto de reglas sencillas que reflejan el comportamiento de cada uno de los posibles casos y mediante la composición de ellas obtenemos la versión completa del protocolo.

Finalmente se hizó una breve verificación de los programas y vimos cómo en la versión de Gamma de orden superior esto se hace con mayor facilidad pues se especifican de manera clara las condiciones necesarias para la aplicación de las reglas, así como las transformaciones que estas realizan. También vímos que al considerar la versión en Gamma de orden superior obtuvimos un enfoque paralelo del protocolo, lo cual nos da ciertas mejoras respecto a la versión secuencial.

En resumen, presentamos un lenguaje de programación orientado al trabajo en paralelo - Gamma - así como algunas de sus variantes. En estos lenguajes vimos las ventajas de tener una semántica formal.

Por ejemplo, pudimos hacer un análisis de los programas de manera más clara y sencilla, así como un proceso de refinamiento más efectivo. Estos lenguajes, aunque en apariencia tienen un comportamiento muy complejo, son muy sencillos y pueden ser la base para el desarrollo de lenguajes más prácticos (e incluso comerciales) en un futuro.

Bibliografía

[1] Banâtre, J. P. y Le Métayer, D. The gamma model and its discipline of programming. En *Science of Computer Programming*, número 15, páginas 55–57. 1990.

[2] Edgar Bermúdez. Sistemas de reducción compuestos: El caso gamma. Tesis de licenciatura, Facultad de Ciencias, UNAM, 2002.

[3] Marc Bourgois. Advantages of formal specifications: A case study of replication in lotus notes. En *Case Studies II*, páginas 231–244.

[4] Chaudron. *The Formal Semantics of Programming Languages an Introduction.* The MIT Press, 1993.

[5] Fradet, Pascal y Le Métayer, Daniel. Shape types. En *Proc. Principies of Programming Languages*, páginas 27–39. ACM Press, Paris, 1997.

[6] Fradet, Pascal y Le Métayer, Daniel. Structured gamma. En *Science of Computer Programming*, número 31, páginas 263–289. 1998.

[7] Francisco Hernández. *A Semantics-Based proof system for Gamma.* PhD thesis, Imperial College of Science of Technology and Medicine, 1999.

[8] Daniel Le Métayer. Higher-order multiset procesing. En *Discrete Mathematics and Theoretical Computer Science*, volumen 18, páginas 179–200. 1994.

[9] Patrick Valduriez M. Tamer. *Principles of Distributed Database Systems.* Prentice Hall, 2 edición, Enero 1999.

[10] David Sands. Composed reduction systems. *Science of Computer Programming*, 1996.

[11] Glynn Winskel. *The Formal Semantics of Programming Languages an Introduction.* The MIT Press, 1993.