Extensión del λ-cálculo para la Modelización de Procesos Concurrentes

Francisco Javier Oliver Villarroya
Departamento de Sistemas Informáticos y Computación
Universidad Politécnica de Valencia

Memoria presentada para optar al título de:
Doctor en Informática

Dirigida por:
Prof. Dr. Isidro Ramos Salavert

Tribunal de lectura:

Presidente:	**Prof. Dr. José María Troya Linero**	**U. Málaga**
Vocales:	**Prof. Dra. María Alpuente Frasnedo**	**U.P. Valencia**
	Prof. Dr. Moreno Falaschi	**U. Udine**
	Prof. Dr. Ernesto Pimentel Sánchez	**U. Málaga**
Secretaria :	**Prof. Dra. Matilde Celma Giménez**	**U.P. Valencia**

Valencia, Septiembre de 1996

Resumen

El λ-cálculo es una teoría sin tipos que interpreta las funciones como reglas, es decir, el proceso de ir de un argumento a un valor, un proceso codificado por una definición. La idea de utilizar el λ-cálculo como un marco matemático para la descripción y el razonamiento acerca de los sistemas computacionales es antigua. De hecho, gracias al análisis realizado por Turing, se puede afirmar que, a pesar de que su sintaxis es muy simple, el λ-cálculo es lo suficientemente potente para describir todas las funciones computables mecánicamente. Pero, como mostró G. Berry, la computación que captura el λ-cálculo es esencialmente secuencial. Un desafío importante que, sobre todo desde finales de los años ochenta, está implicando a muchos investigadores es la construcción de un marco similar al λ-cálculo para la concurrencia y la comunicación entre procesos.

El objetivo fundamental que guiará el trabajo que a continuación presentamos es la formalización de un cálculo que extiende el λ-cálculo para modelizar la concurrencia y la comunicación entre procesos. El lenguaje desarrollado se denomina *λ-cálculo Etiquetado Paralelo (LCEP)*. Su origen está en una propuesta inicial de H. Aït-Kaci (el λ-cálculo Etiquetado) que describe un lenguaje, extensión del λ-cálculo, en el que los argumentos de las funciones se seleccionan mediante etiquetas, incluyendo tanto posiciones numéricas como simbólicas. Esta extensión es *conservativa* en el sentido de que, cuando el conjunto de etiquetas es el conjunto unario $\{1\}$, el λ-cálculo Etiquetado coincide exactamente con el λ-cálculo, condición que no se cumple en las otras propuestas que vamos a estudiar comparativamente como punto de partida. Para describir el nuevo cálculo vamos a utilizar una semántica operacional dada por un sistema de transición, a partir de la cual propondremos diferentes relaciones de equivalencia para modelar los distintos aspectos relacionados con el comportamiento operacional de los procesos.

Por último, ya que la escritura directa de programas en LCEP resulta demasiado compleja en la mayoría de los casos como para pensar en él como recurso expresivo adecuado para la programación, vamos a definir un lenguaje de más alto nivel, ALEPH, que posee los recursos expresivos deseables en programación y permite aprovechar la potencia computacional del sistema. Mostraremos cómo ALEPH se traduce

a LCEP como código máquina y es a través de éste como se realizan las ejecuciones de los programas.

A lo largo del texto es importante descubrir que a través de una sintaxis razonablemente simple es posible expresar todas las características que conlleva la concurrencia dentro de un paradigma funcional.

Agradecimientos

En primer lugar tengo que destacar a Salva. Mucho de lo que aparece aquí es fruto de sus ideas. Su dedicación y consejos han servido para que el trabajo salga a la luz.

Por supuesto a María. Ella siempre está ahí. ¿Qué haríamos nosotros sin sus revisiones constantes y sus esfuerzos por el grupo?

También a mi amigo Germán, a nuestras sesiones de cine, que me sirven para descargar las tensiones del trabajo, y a sus conocimientos y apoyo.

Al tenis, !qué maravilla! A lo mejor, después de todo este tiempo, soy capaz de aprender a jugar bien ya de una vez y ganarle a Juan Ramón. Desde luego que me gustaría.

A Miguel y a Gisela, mis amigos.

A mi familia que, a pesar de todo, sigue atenta a mi futuro.

A Asun, a María José, a Javi, a Vicente, a Majo. Mis amigos de fatigas. Entre todos hemos conseguido que el grupo tenga una solidez cada vez mayor.

A las sesiones de dominó en el Tendur con Juanjo y Germán. Es una experiencia recomendable.

A Albacete, que forma parte de mi historia y que nunca desaparecerá de ella.

A Mati, a Laura, a Encarna, a Juan Carlos, a Paco, a Lidia, a Nati, a Inma, a M. Angeles, a Mabel, a Pietro, a Ferrán, a Carlos, mis amigos de viaje. Nuestras excursiones tienen un encanto que debe permanecer entre nosotros.

A mi amiga, que en la distancia sigue apoyándome y animándome a seguir adelante.

Al Departamento, un lugar confuso y complejo en donde cada día invierto mucho de mí, y a Vicente, por su insistencia.

Por último, y sobre todo, a mi director de Tesis, por su apoyo incondicional.

Índice General

1 Motivaciones **1**

2 Antecedentes **7**

2.1 La Máquina Química Abstracta 7
2.2 El γ-cálculo 10
2.3 El cálculo CHOCS 16
2.4 El π-cálculo 19
2.4.1 El π-cálculo monádico 19
2.4.2 El π-cálculo poliádico 22
2.5 El λ-cálculo Etiquetado 25
2.6 Ejemplos 29
2.6.1 Ejemplo 1: El problema de los Filósofos 29
2.6.2 Ejemplo 2: Teléfonos móviles 34

3 El λ-cálculo Etiquetado Paralelo (LCEP) **39**

3.1 El lenguaje LCEP 40
3.1.1 Interpretación de los símbolos del lenguaje 41
3.1.2 Relación de orden en el conjunto de etiquetas 42
3.2 Interpretación algebraica 43
3.3 Propiedades algebraicas 48
3.4 Axiomas de reducción 49
3.5 Axiomas de reordenación 51
3.6 Reglas de inferencia 52
3.7 El concepto de túnel 53
3.8 Ejemplos 59
3.8.1 Ejemplo 1: El problema de los Filósofos 59
3.8.2 Ejemplo 2: Teléfonos móviles 61
3.9 Comparación con el π-cálculo monádico 62

4 Ejecución de procesos LCEP **67**
4.1 Introducción . . . 67
4.2 Mecanización del π-cálculo . . . 70
4.3 Mecanización de LCEP . . . 72
4.3.1 El sistema de inferencia de LCEP como generador de relaciones 73
4.3.2 Comunicación secuencial y reordenación . . . 76
4.3.3 Comunicación paralela. $\mathcal{T}$-comunicabilidad . . . 77
4.4 Reducción y sistema de inferencia de LCEP . . . 80
4.4.1 Relación generada por el sistema de inferencia . . . 81
4.4.2 Sistema de inferencia asociado a un SRTSC para LCEP . . . 81

5 LCEP como máquina abstracta para ALEPH **85**
5.1 Representación de las construcciones del lenguaje . . . 85
5.1.1 Planteamiento. Modelo computacional y declaraciones . . . 85
5.1.2 Subprogramas . . . 93
5.1.3 Estructura de los procesos asociados a subprogramas . . . 97
5.1.4 Funciones primitivas del lenguaje . . . 101
5.1.5 Diferencias entre funciones y procedimientos . . . 103
5.1.6 Parametrización parcial . . . 104
5.1.7 Definición de la relación de orden $\preceq_{\mathcal{L}}$ . . . 111
5.1.8 Tratamiento de las sentencias de un LMAN como procesos . . 112
5.2 Ejemplos . . . 125
5.2.1 Ejemplo 1: El problema de los Filósofos . . . 125
5.2.2 Ejemplo 2: Teléfonos móviles . . . 127

6 Conclusiones y trabajo futuro **131**

A Un lenguaje con paralelismo explícito **141**
A.1 Esquema de un lenguaje estructurado . . . 141
A.2 Declaraciones . . . 143
A.3 Subprogramas: funciones y procedimientos . . . 144
A.4 Activación de un subprograma . . . 145
A.4.1 Modelos de parametrización . . . 145
A.4.2 Evaluación. Esquemas CBV y CBN . . . 147
A.4.3 Modelo de ejecución de un subprograma . . . 148
A.5 Sentencias básicas . . . 149
A.6 Constructores de sentencias . . . 150
A.7 Extensiones orientadas a la programación paralela . . . 151
A.7.1 Comunicaciones . . . 151
A.7.2 Constructores de paralelismo . . . 152

A.8 Estructura de un programa 152

B El lenguaje de programación ALEPH 155

B.1 Descripción de ALEPH 155

B.1.1 Elementos del lenguaje de programación 155

B.1.2 Sintaxis esquemática 159

B.2 Gramática 164

B.3 Función de traducción 171

B.3.1 Funciones y procedimientos primitivos 179

Índice de Figuras

2.1 El problema de los filósofos . . . 30
2.2 Teléfonos móviles . . . 34
2.3 Simplificación de los Teléfonos Móviles . . . 37

3.1 Arbol sintáctico del proceso $\lambda_P x \parallel \widehat{\ }_P N$ . . . 44
3.2 Arbol sintáctico del proceso $\widehat{\ }_{q_2}(\widehat{\ }_{q_1} A \parallel A_1) \parallel \widehat{\ }_{q_3}(A_2 \parallel \widehat{\ }_{q_4}(A_3 \parallel B))$ . . . 47
3.3 β_P-comunicación paralela . . . 54
3.4 Identificación del túnel en el árbol . . . 56

4.1 Grafo de $\mathcal{T}, F$-comunicabilidad del proceso A . . . 79
4.2 Grafo de $\mathcal{T}, G$-comunicabilidad del proceso A . . . 79
4.3 Grafo de $\mathcal{T}$-comunicabilidad del proceso A . . . 80

5.1 Arbol sintáctico del proceso A . . . 90
5.2 Comunicaciones en la llamada a una función . . . 96
5.3 Comunicaciones tras la instanciación de G . . . 97
5.4 Esquema de la relación de orden . . . 113

Capítulo 1

Motivaciones

A principios de los años treinta, Church construyó el λ-cálculo libre de tipos [Chu32]. Básicamente, el λ-cálculo es una teoría sin tipos que interpreta las funciones como reglas, es decir, el proceso de ir de un argumento a un valor, un proceso codificado por una definición. Los fundadores del λ-cálculo [Chu32] y la teoría de la lógica combinatoria [Cur58, Cur72] (relacionada con este cálculo) tenían dos ideas en mente: desarrollar una teoría general de las funciones computables y extender esta teoría para hacerla servir como un soporte uniforme para la lógica y una parte de las matemáticas. Sin embargo, el descubrimiento de distintas paradojas (Kleene y Rosser [KR35] demostraron que el sistema original de Church era inconsistente) hizo que no tuviera demasiado éxito. A pesar de ello, una parte importante de la teoría ha resultado relevante como base para la teoría de la computación.

La idea de utilizar el λ-cálculo [Bar91] como un marco matemático para la descripción y el razonamiento acerca de los sistemas computacionales es antigua. Ya Church propuso, usando la teoría del λ-cálculo, una formalización de la noción de "efectivamente computable" a través del concepto de "λ-definible" [Chu32]. Kleene mostró que la λ-definibilidad es un concepto equivalente a la recursividad de Gödel-Herbrand [Kle36]. Durante los mismos años, Turing [Tur37] mostró que su noción de "computable" para una "máquina de Turing" es equivalente a la de "λ-definible". Posteriormente, usando las ideas del λ-cálculo, se han obtenido algunos de los teoremas fundamentales de la recursión.

Gracias al análisis realizado por Turing, se puede afirmar que, a pesar de que su sintaxis es muy simple, el λ-cálculo es lo suficientemente potente para describir todas las funciones computables mecánicamente. De hecho, existen diversos lenguajes de programación que presentan características inspiradas por el λ-cálculo. Por otro lado, gracias a las similitudes que aparecen entre el λ-cálculo y algunos lenguajes de programación, las ideas más importantes de su semántica han servido de inspiración para

el desarrollo de las de éstos. Por ejemplo, Landin dio la semántica de ALGOL traduciendo este lenguaje al λ-cálculo y después describiendo una semántica operacional para éste [Lan65].

La sintaxis del λ-cálculo libre de tipos [Bar91] es:

- Los λ-términos son palabras sobre el siguiente alfabeto:
  1. $x, y, \ldots$ variables
  2. λ abstracción
  3. $($, $)$ paréntesis
- El conjunto Λ de los λ-términos se define inductivamente como sigue:
  1. $x \in \Lambda$
  2. $M \in \Lambda \implies (\lambda x.M) \in \Lambda$
  3. $M, N \in \Lambda \implies (MN) \in \Lambda$

La teoría del λ-cálculo tiene como fórmulas:

$$M = N \quad \textit{con} \quad M, N \in \Lambda$$

y aparece axiomatizada por los siguientes axiomas y reglas:
AXIOMA:

- $(\lambda x.M)N \ = \ M[\, x := N \,]$ (β-conversión)

REGLAS:

- $\implies \ M \ = \ M$
- $M \ = \ N \implies N \ = \ M$
- $M \ = \ N \ , \ N \ = \ L \implies M \ = \ L$
- $M \ = \ N \implies MZ \ = \ NZ$
- $M \ = \ N \implies ZM \ = \ ZN$
- $M \ = \ N \implies \lambda x.M \ = \ \lambda x.N$ (ξ-conversión)

La computación que captura el λ-cálculo [Abr89], como mostró Berry, es esencialmente secuencial [Ber78]. Un desafío importante que, sobre todo desde finales de los años ochenta, está implicando a muchos investigadores es la construcción de un marco similar al λ-cálculo para la concurrencia y la comunicación entre procesos [AKG93a, BB93, Bou90, EN86, Hoa85, Mil93d, Mil89, OL94, Oli95, San92]. Unos de los primeros intentos fueron el *Communicating Sequential Processes (CSP)* de

C.A.R. Hoare [Hoa85] y el *Calculus of Communicating Systems (CCS)* de R. Milner [Mil89]. En lo que sigue, vamos a describir algunas de las características esenciales de CCS, pues la propuesta de Milner es la que ha tenido mayor continuación. CCS es un cálculo que está pensado para suministrar un conjunto mínimo de construcciones para la descripción de sistemas concurrentes e indeterministas. Una comunicación en CCS es la acción de pasar un valor, acción que dos procesos pueden ejecutar simultáneamente. Uno de ellos envía un valor a través de un canal etiquetado mientras que el otro recibe este valor en un canal etiquetado con el mismo nombre. De este modo, existen dos primitivas de comunicación en CCS:

- Un constructor de *output* ($\overline{\alpha}e.p$), que representa un proceso que envía e sobre el canal α y entonces se comporta como p.
- Un constructor de *input* ($\alpha x.p$), que representa un proceso que recibe un valor en el canal α.

En el constructor de input, x es una variable ligada y la acción de recibir el valor v lleva a un nuevo proceso $p[x \mapsto v]$, que es igual a p, y en donde se ha sustituido x por v. La comunicación ocurre cuando dos procesos realizan un *emparejamiento* entre acciones de emisión y recepción. Por tanto, podemos escribir la *ley de interacción*, usando un operador de *composición paralela*, $\|$, como:

$$(\alpha x.p \;\|\; \overline{\alpha}e.q) \quad\rightarrow\quad (p[x \mapsto v] \;\|\; q)$$

donde v es el resultado de la evaluación de e. En CCS, esta transición se etiqueta como la acción de comunicación interna τ [Mil89].

Se puede observar que existe una gran semejanza entre esta definición de comunicación y la β-conversión del λ-cálculo. De hecho, la idea original de Milner fue que CCS sirviera como el λ-cálculo de los sistemas concurrentes.

El objetivo fundamental que guía este trabajo es la formalización de un cálculo que extiende el λ-cálculo para modelizar la concurrencia y la comunicación entre procesos.

El lenguaje que desarrollamos se denomina *λ-cálculo Etiquetado Paralelo (LCEP)*. El origen de este lenguaje está en una propuesta inicial de H. Aït-Kaci (el λ-cálculo Etiquetado [AKG93a]) que describe un lenguaje, extensión del λ-cálculo, en el que los argumentos de las funciones se seleccionan mediante etiquetas. El conjunto de etiquetas incluye tanto posiciones numéricas como simbólicas. Esta extensión es *conservativa* en el sentido de que, cuando el conjunto de etiquetas es el conjunto unario $\{1\}$, el λ-cálculo Etiquetado coincide exactamente con el λ-cálculo. Esta idea, fundamental para preservar las buenas propiedades que posee el λ-cálculo tal y como lo concibió Church, no se cumple en otras propuestas que vamos a estudiar comparativamente como punto de partida del trabajo, tomando como base CCS [Mil89] y CSP [Hoa85]. Este estudio incluye la *Chemical Abstract Machine (CHAM)* de G. Berry

y G. Boudol [BB93], el γ-*Calculus* de G. Boudol [Bou89], el *Calculus of Higher Order Communicating Systems (CHOCS)* de B. Thomsen [Tho89] y el *Monadic and Polyadic* π-*Calculus* de R. Milner [Mil93d, Mil92c, MPW92], probablemente la más conocida y extendida de todas ellas.

La estructura de esta tesis es como sigue. En el Capítulo 2, revisamos las propuestas más significativas, citadas anteriormente, mostrando de manera uniforme su sintaxis y su semántica operacional. Vamos a optar por utilizar la propuesta de la mayoría de los investigadores, que pasa por definir la semántica de los sistemas concurrentes por medio del conjunto de experimentos que los sistemas ofrecen a un observador[1]. Usamos el modelo de los sistemas de transición etiquetados de Plotkin [Plo75] como herramienta para definir la semántica operacional de los sistemas concurrentes.

En el Capítulo 3 se presenta la parte más significativa de esta tesis, el λ-cálculo Etiquetado Paralelo (LCEP). Utilizando la semántica operacional, dada por un sistema de transición, proponemos diferentes relaciones de equivalencia para modelar los distintos aspectos relacionados con el comportamiento operacional de los procesos. La relación de equivalencia fundamental está inducida por la noción de *bisimulación*.

El capítulo finaliza presentando una comparación entre el π-cálculo y LCEP, desarrollando la traducción de las diferentes estructuras del π-cálculo como procesos LCEP y viceversa.

En el Capítulo 4 vamos a utilizar una nueva clase de relación de reescritura, la *relación de reescritura sensible al contexto* [Luc95a, Luc95b, Luc96], para dar una caracterización alternativa al comportamiento operacional de los programas. Mostramos cómo se mecaniza la ejecución de procesos LCEP.

La escritura directa de programas en LCEP resulta demasiado compleja, en la mayoría de los casos, como para pensar en él como recurso expresivo adecuado para la programación. Por este motivo, definimos un lenguaje de más alto nivel, al que vamos a llamar ALEPH, que posee los recursos expresivos deseables en programación y permite aprovechar la potencia computacional del sistema [Lan65]. Un análisis de sus características y diversas extensiones realizadas sobre el esquema secuencial para soportar el procesamiento paralelo se presentan en dos apéndices al final de la memoria. En el Capítulo 5 mostramos cómo ALEPH se traduce a LCEP como código máquina y es a través de éste como se realizan las ejecuciones de los programas.

En el último capítulo de la tesis presentamos las conclusiones del trabajo desarro-

[1] La primera teoría general de la concurrencia, que se desarrolló a principios de los años sesenta, fue la teoría de las *redes de Petri* [Pet62]. Esta teoría es una generalización de la teoría de autómatas, que permite la ocurrencia de varias acciones (transiciones de estado) independientemente. Una diferencia importante entre las redes de Petri y las propuestas que vamos a describir es que las redes de Petri prestan especial atención a la *causalidad* entre acciones, mientras que todos los cálculos que veremos son, por el contrario, *observacionales*, es decir, fijan su atención en la observabilidad de los procesos.

llado.

Tras las referencias bibliográficas se han incluido dos apéndices, como hemos mencionado ya. En el Apéndice A se presentan las características que debe presentar un lenguaje de alto nivel con operadores de paralelismo explícitos y en el Apéndice B se describe el lenguaje construido (ALEPH) dando su gramática y la función de traducción correspondiente.

A lo largo del texto es importante descubrir que, a través de una sintaxis razonablemente simple, es posible expresar todas las características que conlleva la concurrencia dentro de un paradigma funcional.

Capítulo 2

Antecedentes

2.1 La Máquina Química Abstracta

La Máquina Química Abstracta (CHAM) se diseñó como un modelo para la computación concurrente asíncrona [BB93, Bou94]. Está basada en el lenguaje Γ de Banâtre y Le Metayer [BM86, BM90]. En ella, los estados son soluciones químicas en donde las moléculas, que flotan, pueden interactuar de acuerdo a una serie de reglas de reacción. Las componentes concurrentes están moviéndose libremente en el sistema y se comunican cuando se ponen en contacto, de una "forma mágica" (sic). En química éste es el resultado del movimiento browniano. Las soluciones pueden estratificarse, encapsulando subsoluciones con membranas que fuerzan a que las reacciones ocurran localmente. Un programa se define por la estructura de las moléculas y por el conjunto de reglas de reacción. Las soluciones son multiconjuntos de moléculas y las reglas de reacción son reescrituras de multiconjuntos. Berry y Boudol presentan las moléculas de un modo sistemático como términos de álgebras.

Las moléculas que pueden interactuar se llaman *iones*. Una solución se puede *calentar* para romper moléculas complejas en sus iones y se puede *enfriar* para reconstruir moléculas más grandes a partir de sus componentes. Por otro lado, una molécula puede contener una subsolución encerrada por una *membrana*, que puede tener algún poro que permita la comunicación entre la solución encapsulada y su entorno. Precisamente, la fuerza del modelo reside en la noción de *membrana*.

Las CHAM's obedecen un conjunto sencillo de leyes estructurales y, cada máquina particular, se define añadiendo un conjunto de reglas que especifican cómo producir nuevas moléculas a partir de las existentes.

Una CHAM se especifica definiendo las moléculas $m, m', \ldots,$ las soluciones S, $S', \ldots,$ y las reglas de transformación. Las moléculas son términos de álgebras, con operaciones específicas para cada máquina concreta. Las soluciones son multiconjun-

tos finitos de moléculas, expresados como $\{\mid m_1, m_2, \ldots, m_k \mid\}$, en donde $\{\mid \cdot \mid\}$ se conoce como el *operador de membrana*. Cada solución puede considerarse como una molécula y, además, puede usarse como una subsolución de otra molécula.

Las reglas de transformación presentan la forma:

$$m_1, m_2, \ldots, m_k \quad \rightarrow \quad m_1^{'}, m_2^{'}, \ldots, m_l^{'}$$

Las reglas se expresan por medio de *esquemas de reglas*. Las reglas reales serán instancias de ellos. La unión multiconjunto de S y S' se escribe como $S \uplus S'$ y $C[\ ]$ va a denotar una molécula con un hueco [ ] en el que se colocará otra molécula.

Las reglas de transformación determinan una relación de transformación $S \rightarrow S'$ entre soluciones, de acuerdo a:

1. *La Ley de Reacción*. Una instancia de la parte derecha de una regla puede sustituir la correspondiente instancia de su parte izquierda. Dada la regla

   $$m_1, m_2, \ldots, m_k \ \rightarrow \ m_1', m_2', \ldots, m_l'$$

   tal que $M_1, M_2, \ldots, M_k$ y $M_1', M_2', \ldots, M_l'$, son instancias de $m_1, m_2, \ldots, m_k$ y de $m_1', m_2', \ldots, m_l'$, respectivamente, entonces:

   $$\{\mid \ M_1, M_2, \ldots, M_k \ \mid\} \quad \rightarrow \quad \{\mid \ M_1', M_2', \ldots, M_l' \ \mid\}$$

2. *La Ley Química*. Las reacciones pueden darse libremente en cualquier solución:

   $$\frac{S \ \rightarrow \ S'}{S \uplus S'' \ \rightarrow \ S' \uplus S''}$$

3. *La Ley de la Membrana*. Una solución puede evolucionar libremente en cualquier contexto:

   $$\frac{S \ \rightarrow \ S'}{\{\mid C[S] \mid\} \ \rightarrow \ \{\mid C[S'] \mid\}}$$

   Algunas CHAM's usan un constructor adicional: la *válvula*. Una válvula es una molécula de la forma $m \triangleleft S$, donde m es una molécula y S es una solución.

4. *La Ley de la Válvula*: $\{\mid m \mid\} \uplus S \ \leftrightarrow \ \{\mid m \triangleleft S \mid\}$

Una CHAM es una máquina intrínsecamente paralela. Se pueden aplicar simultáneamente varias reglas a una solución (supuesto que ninguna molécula aparece implicada en más de una regla) y se pueden transformar subsoluciones en paralelo.

Se distinguen 3 clases de reglas en una CHAM: las reglas de calentamiento $\rightharpoonup$, las reglas de enfriamiento $\rightharpoondown$ y las reglas de reacción $\rightarrow$. Las reglas de calentamiento permiten descomponer una molécula en otras más simples y las reglas de enfriamiento

recomponen una molécula compuesta a partir de sus componentes. Se dice que una solución está caliente (fría) si no se le pueden aplicar reglas de calentamiento (enfriamiento). Berry y Boudol presentan todas las reglas estructurales como reglas de calentamiento, posiblemente emparejadas con reglas de enfriamiento inversas.

La clausura reflexiva, simétrica y transitiva de $(\rightharpoonup \cup \rightharpoondown)$ se expresa como $\stackrel{*}{\rightleftharpoons}$. Usualmente representa una equivalencia estructural. En general, las reglas de reacción involucran a varias moléculas. Se llama *ión* a una molécula que no puede ser calentada más. Una solución es *inerte* si no se le puede aplicar ninguna de las reglas de reacción.

Berry y Boudol llaman γ-cálculo al cálculo construido para la CHAM, utilizando el mismo nombre que el propuesto por Boudol en [Bou89] y que, posteriormente, describiremos. Haciendo la analogía con el λ-cálculo (un receptor en el λ-cálculo es una abstracción $\lambda x.M$), utilizan x^-M para denotar un receptor atómico y M^+ para denotar un emisor atómico enviando el valor M.

La sintaxis del cálculo asociado es la siguiente:

$$M \quad ::= \quad x \mid x^-M \mid (M)^+ \mid (M \mid M) \mid < M >$$

Para formalizar el mecanismo de ejecución, se introduce la noción sintáctica de *estado estable*: aquél que no contiene iones de la misma valencia y que representa una solución inerte.

La sintaxis para los emisores puros, receptores puros y términos estables es:

$$E \quad ::= \quad M^+ \mid (E \mid E) \mid < E >$$
$$R \quad ::= \quad x^-M \mid (R \mid R) \mid < R >$$
$$W \quad ::= \quad E \mid R$$

Las moléculas se describen sintácticamente como:

$$U \quad ::= \quad M \mid S \mid (U \mid U) \mid < U >$$

Por último, las reglas de transformación son:

1. Solución: $U \mid V \ \rightleftharpoons \ U, V$
2. Membrana: $< U > \rightleftharpoons \ \{\mid \ U \ \mid\}$
3. Válvula: $< W > \rightleftharpoons \ W$
4. β-reacción: $x^-M, N^+ \ \rightarrow \ M[N/x]$

Como se puede apreciar, la única regla irreversible es la regla de reacción, ya que es la que expresa la comunicación.

En [HO95] se presenta la construcción en Prolog del prototipo de una CHAM sobre un subconjunto del CCS de R. Milner.

Berry y Boudol muestran que el poder de su cálculo se encuentra esencialmente en las reglas que conciernen al constructor de membranas. Esta construcción es diferente

de la restricción de CCS ya que en la CHAM, si una membrana recoge a un estado estable, la membrana puede desaparecer.

Una crítica importante a la propuesta de Berry y Boudol radica en la utilización de un único canal de comunicación. Esta situación lleva a la imposibilidad de describir sistemas que precisen varios canales como mecanismos de envío de información. Veremos que otras de las propuestas revisadas resuelven este problema.

En [BB93] se propone una semántica basada en las equivalencias de tests. La semántica de los procesos puede derivarse de su observación por medio de los *contextos de programas*, $C[\ ]$. Los contextos de programas pueden verse como tests sobre los procesos y, además, hay una forma natural de definir una equivalencia de tests: dos procesos son *equivalentes* si pasan los mismos tests.

La información operacional más simple es la existencia de una *forma normal* (la *convergencia*): el agente M pasa el test $C[\,]$ si $C[M]$ converge. Formalmente, un agente M *converge* ($M \Downarrow$) si y sólo si existe una solución inerte S tal que $\{|\ M\ |\} \ \rightarrow^* \ S$.

Con esto, es posible definir un preorden de prueba:

$$M \sqsubseteq N \ \stackrel{def}{\Longleftrightarrow} \ \forall C \ \ C[M] \Downarrow \Rightarrow \ C[N] \Downarrow$$

La equivalencia asociada, $\simeq$, se define de manera estándar:

$$M \simeq N \ \stackrel{def}{\Longleftrightarrow} \ M \sqsubseteq N \ \& \ N \sqsubseteq M$$

2.2 El γ-cálculo

Este cálculo es anterior en el tiempo a la construcción de la *Chemical Abstract Machine* (CHAM) que se acaba de describir. La idea de G. Boudol fue extender el λ-cálculo para modelizar procesos concurrentes y comunicación. El mecanismo de comunicación del cálculo [Bou89] es, como en el resto, similar al de CCS. Comunicar consiste en sincronizar el envío y la recepción de un valor a través de un canal (de comunicación) compartido. El cálculo está parametrizado por un conjunto dado de nombres de canales, que se usa en las dos primitivas para *enviar* y *recibir* un valor. Boudol utiliza dos constructores para el paralelismo: el *interleaving*, que no permite comunicación entre los agentes, y la *cooperación*, que fuerza a dos agentes a comunicarse con cualquier nombre de canal.

El constructor del *interleaving* ($p \mid q$) consiste en la yuxtaposición de p y q, sin permitir ninguna comunicación entre ellos. Representa la concurrencia. El constructor de *cooperación* ($p \odot q$), en cambio, consiste en realizar todas las comunicaciones posibles entre p y q hasta la terminación de uno de ellos. Representa la comunicación. El operador $\odot$ no es un operador *estático* ya que los procesos que cooperan ($p \odot q$) no pueden comunicar con otro proceso si p y q no han terminado, pero son libres de

hacerlo si p o q han acabado. El operador $|$ es conmutativo y asociativo; en cambio, el operador $\odot$ no es asociativo.

Dado un conjunto enumerable $\mathcal{X}$ de variables $x, y, z, \ldots$ y un conjunto no vacío $\mathcal{N}$ de nombres de canales $\alpha, \beta, \ldots$, la sintaxis del γ-cálculo se define como sigue.

Primero se definen las *ligaduras*, términos construidos como:

$$\rho \quad ::= \quad \varepsilon \;\mid\; \alpha x \;\mid\; (\rho \cdot \rho) \;\mid\; (\rho \mid \rho)$$

donde ε es una ligadura vacía, $\alpha \in \mathcal{N}$ es un nombre de canal, $x \in \mathcal{X}$ representa una variable, αx representa una recepción en el canal α, $(\rho \cdot \rho)$ la secuenciación de recepciones y $(\rho \mid \rho)$, el interleaving de recepciones.

En [Bou89] se considera a las ligaduras bajo la congruencia $\doteq$, generada por las ecuaciones:

$$(\rho \cdot \varepsilon) = \rho = (\varepsilon \cdot \rho)$$

$$(\rho \mid \varepsilon) = \rho = (\varepsilon \mid \rho)$$

Esta congruencia define la igualdad sobre ligaduras. Cualquier ligadura ρ restringe a las variables que pertenecen al conjunto $var(\rho)$, definido como:

- $var(\varepsilon) = \emptyset$
- $var(\alpha x) = \{x\}$
- $var(\rho \cdot \rho') = var(\rho) \cup var(\rho')$
- $var(\rho \mid \rho') = var(\rho) \cup var(\rho')$

La sintaxis completa del γ-cálculo sigue la siguiente gramática:

$$p \quad ::= \quad 1 \;\mid\; x \;\mid\; \overline{\alpha} p \;\mid\; < \rho > .p \;\mid\; (p \odot p) \;\mid\; (p \mid p)$$

Se llama Γ al conjunto de términos generado por esta gramática. Usamos $p, q, r, \ldots$ para representar a los términos que, en general, llamamos *agentes* o *procesos*. Se define la *ley de interacción* γ como:

$$\begin{aligned} \gamma \quad : \quad & ((\cdots \mid \alpha x.p \mid \cdots) \;\odot\; (\cdots \mid \overline{\alpha} q \mid \cdots)) \quad \rightarrow \\ & \rightarrow \quad ((\cdots \mid p[x \mapsto q] \mid \cdots) \;\odot\; (\cdots \mid 1 \mid \cdots)) \end{aligned}$$

Una variable x está *ligada* si pertenece al ámbito de una ligadura. Entonces, al sustituir q por x en p ($p[x \mapsto q]$) puede ocurrir que haya que renombrar algunas variables ligadas de p. Es importante definir con cuidado la noción de sustitución. Para ello Boudol utiliza las definiciones de [Sto88]. Así, el conjunto de las variables libres del término p, $free(p)$, se define como:

- $free(1) = \emptyset$
- $free(x) = \{x\}$
- $free(\overline{\alpha}p) = free(p)$
- $free(\langle\ \rho\ \rangle.p) = free(p) - var(\rho)$
- $free(p \odot q) = free(p) \cup free(q) = free(p \mid q)$

Se dice que un término p es *cerrado* si $free(p) = \emptyset$.

Como es usual, se dice que dos términos son iguales si solamente difieren en el nombre de sus variables ligadas. Así, la igualdad es la mínima congruencia $\equiv$ generada por las siguientes ecuaciones:

$$(p \odot 1) \;=\; p \;=\; (1 \odot p)$$

$$(p \mid 1) \;=\; p \;=\; (1 \mid p)$$

$$< \varepsilon > .p \;=\; p$$

$$< \rho > .p \;= < \rho' > .p \quad si\ \rho \equiv \rho'$$

$$< \rho > .p \;= < \rho[x \mapsto y] > .p[x \mapsto y] \quad si\ x \in var(\rho)\ \ \&\ \ y \notin free(p) \cup var(\rho)$$

Se puede demostrar que $\equiv$ tiene la propiedad de sustitutividad, esto es,

$$p \equiv q \quad \Rightarrow \quad p[\sigma] \equiv q[\sigma] \quad \text{para toda sustitución} \quad \sigma$$

Un agente p se encuentra en una situación de *acabado* ($p\dagger$) si $p \equiv 1$.

De nuevo, para definir la semántica se usa la técnica de las transiciones etiquetadas. Según [Bou89], es el mejor modo de formalizar la idea de que los procesos no necesitan ser contiguos para comunicarse.

La semántica se da por medio de transiciones etiquetadas $p \stackrel{a}{\rightarrow} p'$, donde la acción a puede ser $\overline{\alpha_p}$, α_p, o la acción de comunicación τ. Se dice que a y b son acciones *complementarias* ($a \frown b$) si $a = \alpha_p$ y $b = \overline{\alpha_p}$, o viceversa.

Para definir la semántica de $< \rho > .p$ se introduce una relación de transición entre ligaduras, $\rho \stackrel{a}{\rightarrow} \rho'$ (donde a tiene la forma $\alpha_{x,p}$), que es la menor relación que satisface las siguientes reglas:

1. $p \in \Gamma \;\; \vdash \;\; \alpha x \stackrel{\alpha_{x,p}}{\rightarrow} \varepsilon$

2. $\rho \stackrel{a}{\rightarrow} \rho' \;\; \vdash \;\; (\rho \cdot \rho'') \stackrel{a}{\rightarrow} (\rho' \cdot \rho'')$

3. $\rho \doteq \varepsilon \,\&\, \rho' \stackrel{a}{\to} \rho'' \;\vdash\; (\rho \cdot \rho') \stackrel{a}{\to} \rho''$

4. $\rho \stackrel{a}{\to} \rho' \;\vdash\; (\rho \mid \rho'') \stackrel{a}{\to} (\rho' \mid \rho'')$

5. $\rho \stackrel{a}{\to} \rho' \;\vdash\; (\rho'' \mid \rho) \stackrel{a}{\to} (\rho'' \mid \rho')$

Por otro lado, la relación de transición $\to$ sobre agentes es el menor subconjunto de $\Gamma \times A \times \Gamma$ que satisface:

1. Salida (introduce la acción de comunicación):

$\vdash\; \overline{\alpha}p \stackrel{\overline{\alpha_p}}{\to} 1$

2. Entrada (introduce la acción de comunicación):

$\rho \stackrel{\alpha_{x,q}}{\to} \rho' \;\vdash\; <\rho>.p \stackrel{\alpha_q}{\to} <\rho'>.p[x \mapsto q]$

3. Entrada (cuando la ligadura está vacía):

$\rho \doteq \varepsilon \,\&\, p \stackrel{a}{\to} p' \;\vdash\; <\rho>.p \stackrel{a}{\to} p'$

4. Comunicación γ (la ley de interacción):

$p \stackrel{a}{\to} p', q \stackrel{b}{\to} q' \,\&\, a \frown b \;\vdash\; (p \odot q) \stackrel{\tau}{\to} (p' \odot q')$

La transición $\stackrel{\tau}{\to}$ es compatible con todos los constructores.

5. Salida:

$p \stackrel{\tau}{\to} p' \;\vdash\; \overline{\alpha}p \stackrel{\tau}{\to} \overline{\alpha}p'$

6. Entrada:

$p \stackrel{\tau}{\to} p' \;\vdash\; <\rho>.p \stackrel{\tau}{\to} <\rho>.p'$

7. Cooperación (izquierda):

$p \stackrel{\tau}{\to} p' \;\vdash\; (p \odot q) \stackrel{\tau}{\to} (p' \odot q)$

8. Cooperación (derecha):

$q \stackrel{\tau}{\to} q' \;\vdash\; (p \odot q) \stackrel{\tau}{\to} (p \odot q')$

9. Interleaving (izquierda):

$p \stackrel{a}{\to} p' \;\vdash\; (p \mid q) \stackrel{a}{\to} (p' \mid q)$

$\stackrel{a}{\rightarrow}$ es compatible con el interleaving para todo $a \in A$.

10. Interleaving (derecha):

$$q \;\stackrel{b}{\rightarrow}\; q' \;\;\vdash\;\; (p \mid q) \;\stackrel{b}{\rightarrow}\; (p \mid q')$$

11. Cooperación (unit izquierda):

$$q \;\stackrel{a}{\rightarrow}\; q', p\dagger \;\;\vdash\;\; (p \odot q) \;\stackrel{a}{\rightarrow}\; q'$$

La cooperación sólo se da cuando uno de los dos procesos participantes ha finalizado; es decir, $p\dagger$ o $q\dagger$.

12. Cooperación (unit derecha):

$$p \;\stackrel{b}{\rightarrow}\; p', q\dagger \;\;\vdash\;\; (p \odot q) \;\stackrel{b}{\rightarrow}\; p'$$

Si $p \not\equiv 1 \not\equiv q$ entonces $(p \odot q)$ solamente puede ejecutar la acción τ. La comunicación entre p y q está prohibida con el constructor $(p \mid q)$. Se denota $p \stackrel{\tau}{\rightarrow} p'$ como $p \rightarrow p'$ y, por definición, se dice que es la γ-reducción entre términos de Γ.

La noción de bisimulación usada por Boudol es una pequeña extensión de la de Milner y Park, cuya definición, a grandes rasgos, es la siguiente:

Se dice que una relación binaria S es una *simulación* [MPW92] si $P\ S\ Q$ implica que, si $P \stackrel{\alpha}{\rightarrow} P'$, entonces existe Q' tal que $Q \stackrel{\alpha}{\rightarrow} Q'$ y $P'\ S\ Q'$; es decir, cualquier transición a partir de P puede ser simulada por una transición a partir de Q tal que la derivación de Q' a partir de P' se mantiene en la simulación. De este modo, una relación binaria S es una *bisimulación* si S y su inversa son simulaciones.

Por otro lado, dos procesos son *iguales*, $M = N$, si y solamente si existe un proceso P tal que $M \stackrel{\alpha}{\rightarrow}^{*} P$ y $N \stackrel{\alpha}{\rightarrow}^{*} P$.

Para Boudol, se pueden ver las acciones bajo bisimulación y, además, se tiene en cuenta la terminación potencial de los agentes. Además, se define directamente la bisimulación para términos no cerrados; por lo tanto, se dice que dos términos p y q son *similares* si todas sus instancias $p[\sigma]$ y $q[\sigma]$ tienen comportamientos similares. Boudol usa dos nociones de *simulación*: fuerte (relativa a la relación de transición $\rightarrow$) y débil.

Sea $R \subseteq \Gamma \times \Gamma$ una relación sobre términos. En [Bou89] se define su extensión $\widehat{R} \subseteq A \times A$ como:

$$a\ \widehat{R}\ b \;\;\stackrel{def}{\Longleftrightarrow}\;\; a = b \;\; o \;\; \exists \alpha \in \mathcal{N}\ \&\ \exists p, q.\ p\ R\ q\ \&\ a = \overline{\alpha_p}\ \&\ b = \overline{\alpha_q}$$

Una relación $R \subseteq \Gamma \times \Gamma$:

1. es una *simulación fuerte*, si satisface

$$S_1: \quad p\ R\ q\ \&\ p[\sigma] \stackrel{a}{\rightarrow} p' \;\;\Rightarrow\;\; \exists b.\ a\ \widehat{R}\ b\ \exists q'.\ p'\ R\ q'\ \&\ q[\sigma] \stackrel{b}{\rightarrow} q'$$

$S_2: \quad p\ R\ q\ \&\ p\dagger \quad \Rightarrow \quad q\dagger$

Es un refinamiento de la definición usual de simulación: agentes "fuertemente" similares deben ejecutar acciones similares. La simulación fuerte preserva la propiedad de terminación.

2. es una *bisimulación fuerte*, si es una simulación fuerte simétrica.

Se puede ver que toda simulación fuerte es sustitutiva.

La congruencia $\equiv$ es una simulación fuerte sobre Γ. Este resultado permite definir la relación de transición $\rightarrow$ sobre $\Gamma/\equiv$.

G. Boudol adopta la equivalencia observacional de Milner como su noción de igualdad [Mil89]. La equivalencia observacional se define con respecto a una relación de transición en la que se abstraen las comunicaciones internas (acciones τ). Esta relación de transición $\Longrightarrow$ es el menor subconjunto de $\Gamma \times A \times \Gamma$ conteniendo $\rightarrow$ y satisfaciendo:

1. $\vdash\ p \stackrel{\tau}{\Longrightarrow} p$

2. $p \stackrel{a}{\Longrightarrow} p''\ ,\ p'' \stackrel{\tau}{\Longrightarrow} p' \quad \vdash \quad p \stackrel{a}{\Longrightarrow} p'$

3. $p \stackrel{\tau}{\Longrightarrow} p''\ ,\ p'' \stackrel{a}{\Longrightarrow} p' \quad \vdash \quad p \stackrel{a}{\Longrightarrow} p'$

Se puede observar que cualquier simulación observacional es sustitutiva.

Una relación $R \subseteq \Gamma \times \Gamma$:

1. es una *simulación observacional* o *débil*, si satisface:

   $W_1: \quad p\ R\ q\ \&\ p[\sigma] \stackrel{a}{\Longrightarrow} p' \quad \Rightarrow \quad \exists b.\ a\ \widehat{R}\ b\ \exists q'.\ p'\ R\ q'\ \&\ q[\sigma] \stackrel{b}{\Longrightarrow} q'$

   $W_2: \quad p\ R\ q\ \&\ p\dagger\ \ \exists q'.\ q[\sigma] \stackrel{\tau}{\Longrightarrow} q'\ \&\ q'\dagger$

2. es una *bisimulación observacional* o *débil*, si es una simulación débil simétrica.

La equivalencia observacional usada como igualdad semántica es la menor bisimulación débil tal que:

$$p \approx q \quad \stackrel{def}{\Longleftrightarrow} \quad \exists R \subseteq \Gamma \times \Gamma \text{ que es una bisimulación débil } \ \&\ \ p\ R\ q$$

Entonces $\approx$ es una bisimulación débil. Además, $\approx$ es una equivalencia.

2.3 El cálculo CHOCS

El *Cálculo para Sistemas de Comunicación de Orden Superior* (CHOCS) [Tho89, Tho93, Tho95] considera los procesos de *envío* y *recepción* al mismo nivel que la composición paralela y el indeterminismo. El cálculo de Ben Thomsen es una extensión de CCS y, con él, es posible simular el λ-cálculo sin tipos [Tho89]. Según Thomsen, "*CCS tiene limitaciones cuando se pretende describir sistemas en expansión. Esta deficiencia proviene del hecho de la naturaleza de primer orden de CCS*". Es usual utilizar construcciones de orden superior en muchas ramas de la teoría de la computación ya que proporciona técnicas de abstracción poderosas y elegantes.

Se da una semántica operacional por medio de un sistema de transición etiquetado, [Plo81], definiendo el conjunto de experimentos que son visibles. A grandes rasgos, el modelo de Plotkin consiste en lo siguiente [Plo81, Tho89]:

Sea P un conjunto de procesos y Ac un conjunto de acciones que pueden ser ejecutadas por los procesos. Una *relación de derivación* $\rightarrow \;\subseteq\; P \times Ac \times P$ define el cambio dinámico de los procesos en función de las acciones ejecutadas. Para expresar que $(p, \alpha, q) \in \rightarrow$ normalmente se suele escribir como $p \overset{\alpha}{\rightarrow} q$, cuyo significado es: "el proceso p puede ejecutar una acción α y, al hacerlo, se convierte en el proceso q". La terna $\mathcal{P} = (P, Ac, \rightarrow)$ constituye el *sistema de transición* de los procesos.

Siguiendo la propuesta de Milner, se considera el sistema de transición etiquetado $\mathcal{P} = (Pr, Act, \longrightarrow)$, donde Pr es un conjunto de procesos y Act es un conjunto de acciones que los procesos pueden ejecutar, y que tiene la forma $Nombres \times \{?, !\} \times Pr \cup \{\tau\}$. $Nombres$ es un conjunto de nombres de canales. En este sistema de transición, $p \overset{a?p'}{\longrightarrow} p''$ puede leerse como que "p puede recibir el proceso p' a través del canal a y haciendo esto se convierte en el proceso p''". $p \overset{a!p'}{\longrightarrow} p''$ puede leerse como que "p puede enviar el proceso p' a través del canal a y haciendo esto se convierte en el proceso p''". La notación ?, ! se utiliza para indicar la dirección (entrada/salida) de la comunicación. Como en CCS, se usa el símbolo especial τ, que no pertenece al conjunto $Nombres$, para simbolizar los movimientos internos y, de este modo, $p \overset{\tau}{\longrightarrow} p'$ puede interpretarse como que "el proceso p puede ejecutar una acción interna y haciéndolo se convierte en el proceso p'". Thomsen utiliza Γ para indicar cualquiera de los posibles tipos de acción: $a?p$, $a!p$ o τ.

En CHOCS se extiende la definición de *sustitución* dada en CCS [AZ84] permitiendo la sustitución en procesos construidos usando los operadores de composición paralela, restricción o renombramiento. Un proceso ligado por un prefijo de entrada tiene la capacidad de recibir cualquier proceso. El proceso recibido se pone en uso sustituyéndolo por la variable ligada.

Los procesos se construyen desde el proceso inactivo *nil*, dos tipos de prefijo (ligadura de *input* $(a?x.p)$ y de *output* $(a!p'.p)$), la elección no determinista $(p + p)$, la

composición paralela ($p \mid p$), la restricción ($p\backslash a$), el renombramiento ($p[S]$) y ciertas variables que están ligadas por un prefijo de entrada.

Esta es la sintaxis:

$$p \quad ::= \quad nil \ \mid \ a \ ? \ x.p \ \mid \ a \ ! \ p'.p \ \mid \ p + p' \ \mid \ p \mid p' \ \mid \ p\backslash a \ \mid \ p[S] \ \mid \ x$$

en donde $a \in Nombres$, $S : Nombres \rightarrow Nombres$ y $x \in V$ (un conjunto de variables).

Hay un orden de precedencia entre los operadores:

$$\backslash \quad > \quad ?, ! \quad > \quad \mid \quad > \quad +$$

Si observamos con detenimiento la sintaxis de CHOCS descubriremos que, a diferencia de otras, no posee ningún constructor de recursión. Según demuestra Thomsen en [Tho89] los comportamientos recursivos pueden simularse usando solamente envío de procesos en la comunicación. Por otro lado, las guardas para la *entrada* son ligaduras de variables; esto implica la noción de variables *libres* y *ligadas*, definidas de la forma habitual.

Por supuesto, se puede conseguir la sincronización pura (como en CCS) ignorando los procesos enviados o recibidos.

La semántica operacional de CHOCS se da, al igual que en el resto de lenguajes, en términos de un sistema de transición de estados:

1. Prefijo:

$$a?x.p \xrightarrow{a?p'} p[p'/x] \qquad a!p'.p \xrightarrow{a!p'} p$$

2. Elección no determinista:

$$\frac{p \xrightarrow{\Gamma} p'}{p + q \xrightarrow{\Gamma} p'} \qquad \frac{p \xrightarrow{\Gamma} p'}{q + p \xrightarrow{\Gamma} p'}$$

3. Paralelismo:

$$\frac{p \xrightarrow{\Gamma} p'}{p \mid q \xrightarrow{\Gamma} p' \mid q} \qquad \frac{p \xrightarrow{\Gamma} p'}{q \mid p \xrightarrow{\Gamma} q \mid p'}$$

$$\frac{p \xrightarrow{a?p'} p'' \quad q \xrightarrow{a!p'} q''}{p \mid q \xrightarrow{\tau} p'' \mid q''} \qquad \frac{p \xrightarrow{a!p'} p'' \quad q \xrightarrow{a?p'} q''}{p \mid q \xrightarrow{\tau} p'' \mid q''}$$

4. Restricción:

$$Para \ \ a \neq b \quad \frac{p \xrightarrow{a?p'} p''}{p\backslash b \xrightarrow{a?p'} p''\backslash b} \quad \frac{p \xrightarrow{a!p'} p''}{p\backslash b \xrightarrow{a!p'} p''\backslash b} \quad \frac{p \xrightarrow{\tau} p''}{p\backslash b \xrightarrow{\tau} p''\backslash b}$$

5. Renombramiento:

$$\frac{p \stackrel{a?p'}{\longrightarrow} p''}{p[S] \stackrel{S(a)?p'}{\longrightarrow} p''[S]} \qquad \frac{p \stackrel{a!p'}{\longrightarrow} p''}{p[S] \stackrel{S(a)!p'}{\longrightarrow} p''[S]} \qquad \frac{p \stackrel{\tau}{\longrightarrow} p''}{p[S] \stackrel{\tau}{\longrightarrow} p''[S]}$$

Basándose en la semántica operacional, Thomsen propone algunas equivalencias y preórdenes para capturar aspectos del comportamiento observacional de los procesos: la bisimulación *de orden superior* ($p \sim q$) y la bisimulación *de orden superior débil* ($p \approx q$).

Una *bisimulación de orden superior* R es una relación binaria sobre el conjunto de procesos Pr tal que, siempre que $p\ R\ q$ y $\Gamma \in Act$, entonces:

1. Si $p \stackrel{\Gamma}{\longrightarrow} p'$, entonces $q \stackrel{\Gamma'}{\longrightarrow} q'$ para algún q', Γ' con $\Gamma\ \hat{R}\ \Gamma'$ y $p'\ R\ q'$.

2. Si $q \stackrel{\Gamma}{\longrightarrow} q'$, entonces $p \stackrel{\Gamma'}{\longrightarrow} p'$ para algún p', Γ' con $\Gamma\ \hat{R}\ \Gamma'$ y $p'\ R\ q'$.

donde $\hat{R} = \{(\Gamma, \Gamma') \mid (\Gamma = a?p'' \ \&\ \Gamma' = a?q'' \ \&\ p''\ R\ q'') \ \vee\ (\Gamma = a!p'' \ \&\ \Gamma' = a!q'' \ \&\ p''\ R\ q'') \ \vee\ (\Gamma = \Gamma' = \tau)\}$.

Se dice que dos procesos p y q son *equivalentes* bajo bisimulación de orden superior si y solamente si existe una bisimulación de orden superior R conteniendo el par (p, q). En este caso, se escribe $p \sim q$. Como Thomsen demuestra en [Tho89], $\sim$ es una relación de equivalencia.

Por otro lado, se define la equivalencia bajo bisimulación de orden superior *débil* o *equivalencia observacional* como sigue:

Una *bisimulación de orden superior débil* R es una relación binaria sobre Pr tal que, siempre que $p\ R\ q$ y $\Phi \in (Act\backslash\{\tau\} \cup \{\varepsilon\})$ entonces:

1. Si $p \stackrel{\Phi}{\Longrightarrow} p'$, entonces $q \stackrel{\Phi'}{\Longrightarrow} q'$ para algún q', Φ' con $\Phi\ \hat{\hat{R}}\ \Phi'$ y $p'\ R\ q'$.

2. Si $q \stackrel{\Phi}{\Longrightarrow} q'$, entonces $p \stackrel{\Phi'}{\Longrightarrow} p'$ para algún p', Φ' con $\Phi\ \hat{\hat{R}}\ \Phi'$ y $p'\ R\ q'$.

donde $\hat{\hat{R}} = \{(\Phi, \Phi') \mid (\Phi = a?p'' \ \&\ \Phi' = a?q'' \ \&\ p''\ R\ q'') \ \vee\ (\Phi = a!p'' \ \&\ \Phi' = a!q'' \ \&\ p''\ R\ q'') \ \vee\ (\Phi = \Phi' = \varepsilon)\}$.

Dos procesos p y q se dice que son equivalentes bajo bisimulación de orden superior débil si y solamente si existe una bisimulación de orden superior débil R conteniendo el par (p, q). En este caso, se escribe $p \approx q$. $\approx$ es una equivalencia.

La equivalencia de bisimulación es más distintiva que la equivalencia observacional, es decir:

$$p \sim p' \ \Rightarrow\ p \approx p'$$

Existen diferentes trabajos que relacionan CHOCS y el π-cálculo (que presentamos a continuación). Entre ellos están los realizados por R. Amadio, [Ama92, Ama93].

2.4 El π-cálculo

El π-cálculo [Mil93d, Mil92c, Mil93c, MPW92, Nie96, Pie93, PS92] surgió como el intento de expresar algebraicamente una movilidad entre procesos completa. Según cuenta Milner en [Mil93d] su deseo fue motivado parcialmente por el sistema Actors [AH86, Agh90] de G. Agha, que no poseía una semántica definida. Años después, Engberg y Nielsen [EN86] consiguieron darle una formulación algebraica. El π-cálculo es una simplificación de ese trabajo.

Su nombre (π) proviene de la idea de Milner de construir un cálculo universal para el paralelismo, del mismo modo que el λ-cálculo lo es para la programación secuencial.

2.4.1 El π-cálculo monádico

El π-cálculo es un modelo para la computación concurrente basado en la noción de *nombramiento*. Su entidad más primitiva es el *nombre* ($\mathcal{X}$), que no tiene estructura: $x, y, \ldots \in \mathcal{X}$. La otra clase de entidad es el *proceso* ($\mathcal{P}$). Los procesos se denotan con $P, Q, \ldots \in \mathcal{P}$, y se construyen a partir de los nombres mediante:

$$P \quad ::= \quad \textstyle\sum_{i \in I} \pi_i.P_i \;\mid\; (P \mid Q) \;\mid\; !P \;\mid\; (\nu x)P$$

donde $\sum$ representa una elección entre varios procesos, $|$ es el operador que permite expresar el *paralelismo* entre dos procesos P y Q, ! es el operador de *replicación*, que representa la definición de procesos infinitos (es decir, que aparecen un número indeterminado de veces), y ν es el operador de restricción, que delimita el espacio de utilización de una variable.

I es un conjunto indexado finito. En un sumando $\pi.P$, el prefijo π representa una acción atómica. Hay dos formas básicas de prefijo:

- $x(y)$: Entra algún nombre y a través del nombre de ligadura x.
- $\overline{x}y$: Sale el nombre y a lo largo del nombre de ligadura x.

Decimos que x es el *sujeto* e y es el *objeto* de la acción. El sujeto es positivo para entradas y negativo para salidas.

La *suma* representa a un proceso disponible para tomar parte en una de varias alternativas de comunicación. La elección no la hace el proceso. A los procesos que tienen esta forma (de suma) se les llama *procesos normales*. Todos los procesos pueden ser convertidos a su forma normal. La sintaxis de los procesos *normales* ($M, N, \ldots \in \mathcal{N}$) es:

$$N \quad ::= \quad \pi.P \;\mid\; 0 \;\mid\; M + N$$

donde 0 representa el proceso nulo.

A través de las dos formas básicas de prefijo, podemos definir los conjuntos $free(P)$ (el conjunto de los nombres libres) y $bn(P)$ (el conjunto de los nombres ligados) de un proceso P, de la manera usual:

$$bn(x(y)) = \{y\} \quad , \quad free(x(y)) = \{x\}$$
$$bn(\overline{x}y) = \emptyset \quad , \quad free(\overline{x}y) = \{x, y\}$$

De la misma forma, definimos el conjunto de los nombres de un proceso P como:

$$n(P) = bn(P) \ \cup \ free(P)$$

La congruencia estructural $\equiv$ es la menor relación de congruencia sobre $\mathcal{P}$ tal que:

1. Los procesos se identifican si solamente difieren en un cambio de nombres ligados.
2. $(\mathcal{N}/\equiv, +, 0)$ es un monoide conmutativo.
3. $(\mathcal{P}/\equiv, |, 0)$ es un monoide conmutativo.
4. $!P \ \equiv \ P \mid !P$.
5. $(\nu x)0 \ \equiv \ 0$, $(\nu x)(\nu y)P \ \equiv \ (\nu y)(\nu x)P$
6. Si $x \notin free(P)$ entonces $(\nu x)(P \mid Q) \ \equiv \ P \mid (\nu x)Q$

La relación de reducción sobre procesos: $P \rightarrow P'$, formaliza que P puede transformarse en P' en un solo paso de computación. La relación se define mediante un axioma (la comunicación) y tres reglas de inferencia:

1. Comunicación:

$$(\cdots + x(y).P + \cdots) \ \mid \ (\cdots + \overline{x}z.Q + \cdots) \ \rightarrow \ P\{z/y\} \ \mid \ Q$$

2. Composición:

$$\frac{P \ \rightarrow \ P'}{P \ \mid \ Q \ \rightarrow \ P' \ \mid \ Q}$$

3. Restricción:

$$\frac{P \ \rightarrow \ P'}{(\nu x)P \ \rightarrow \ (\nu x)P'}$$

Estas dos reglas establecen que la reducción puede ocurrir bajo composición y restricción.

4. Congruencia estructural:

$$\frac{Q \ \equiv \ P \qquad P \ \rightarrow \ P' \qquad P' \ \equiv \ Q'}{Q \ \rightarrow \ Q'}$$

Esta regla muestra que términos congruentes estructuralmente tienen las mismas reducciones.

Es muy importante detectar las operaciones no permitidas:

- No están permitidas las reducciones bajo prefijo o suma.
- Estas reglas no permiten reducción bajo replicación.
- Las reglas no dicen nada acerca de la comunicación potencial de un proceso P con los demás.

En [Mil93d], Milner muestra que una semántica declarativa satisfactoria para el π-cálculo se puede definir a partir de los conceptos de *reducción* y *observabilidad*, [San93a, Liu94, Wal94]. Previamente, introduce los conceptos de *no ligado* y *observable*.

En general, según Milner, un agente B ocurre *no ligado* en A si el agente presenta alguna ocurrencia en A pero no bajo un prefijo α.

Un proceso P es *observable* en α ($P \downarrow_\alpha$) si $\alpha.A$ ocurre no ligado en P, con α no restringido.

De este modo, en [Mil93d] se define la *equivalencia de reducción fuerte* ($\dot{\sim}_r$) como la mayor relación de equivalencia sobre procesos, Ξ, tal que $P \ \Xi \ Q$ implica que:

1. Si $P \rightarrow P'$, entonces $Q \rightarrow Q'$ para todo Q' tal que $P' \ \Xi \ Q'$.
2. Para cada α, si $P \downarrow_\alpha$ entonces $Q \downarrow_\alpha$.

Este concepto es también conocido como *bisimulación "barbed"*.

La equivalencia de reducción es una idea natural pero, desgraciadamente, no se preserva en las construcciones de procesos. Por ello, Milner define la *congruencia de reducción fuerte* ($\sim_r$) como la mayor congruencia incluida en la equivalencia de reducción.

Se puede demostrar que se cumple que $P \sim_r Q$ si y solamente si para todos los contextos de procesos, $C[\ ]$, se cumple que $C[P] \ \dot{\sim}_r \ C[Q]$.

Por otro lado, en [Mil93d], también se define la *equivalencia de reducción débil*, $\dot{\approx}_r$, como la mayor relación de equivalencia sobre procesos, Ξ, tal que $P \ \Xi \ Q$ implica que:

1. Si $P \rightarrow P'$, entonces $Q \rightarrow^* Q'$ para todo Q' tal que $P' \ \Xi \ Q'$.
2. Para cada α, si $P \downarrow_\alpha$ entonces $Q \downarrow_\alpha^*$.

La *congruencia de reducción débil*, $\approx_r$, es la mayor congruencia incluida en la equivalencia de reducción débil.

Estas dos congruencias permiten capturar el comportamiento operacional de los procesos, [BN92].

2.4.2 El π-cálculo poliádico

Una entrada poliádica $x(y_1 \ldots y_n)$ o una salida poliádica $\overline{x}y_1 \ldots y_n$ se puede introducir de una forma sencilla en el π-cálculo monádico. Pero la poliadicidad [Mil93d] no debe reducirse a una abreviación. De hecho, se toma la comunicación poliádica como primitiva.

Una *abstracción* toma la forma $(\lambda x_1 \ldots x_n)P$ (es una abstracción de los nombres de un proceso). Diremos que esta abstracción tiene aridad n. En particular, podemos pensar que un proceso P es una abstracción con aridad cero.

La forma prefija de salida (una *concreción*) se define como $\overline{x}y_1 \ldots y_n.P$ y, por definición, esto es lo mismo que $\overline{x}.[y_1 \ldots y_n]P$. Cada variable y_i es un *nombre dato* de la concreción. Todo proceso P es una concreción de aridad cero.

Las principales diferencias que aparecen al pasar del cálculo monádico al poliádico son:

- Los prefijos $x(\vec{y})$ y $\overline{x}\ \vec{y}$ ya no son primitivos.
- Se añaden las abstracciones $F, G, \ldots$ y las concreciones $C, D, \ldots$, denotándolas colectivamente *agentes* $A, B, \ldots$

Milner utiliza $\alpha, \beta, \ldots$ para nombres y co-nombres y $\vec{x}, \vec{y}, \ldots$ para vectores de nombres, con longitudes $|\vec{x}|, |\vec{y}|, \ldots$

La sintaxis se define como sigue:

1. Procesos normales:

$$N \ ::= \ \alpha.A \ | \ 0 \ | \ M + N$$

2. Procesos:

$$P \ ::= \ N \ | \ (P \mid Q) \ | \ !P \ | \ (\nu x)P$$

3. Abstracciones:

$$F \ ::= \ P \ | \ (\lambda x)P \ | \ (\nu x)P$$

4. Concreciones:

$$C \ ::= \ P \ | \ [x]C \ | \ (\nu x)C$$

5. Agentes:

$$A \ ::= \ F \ | \ C$$

Las leyes 1 a 6 de la congruencia estructural del π-cálculo monádico son válidas aquí también. A las reglas anteriores hay que añadir:

7. $(\nu y)(\lambda x)F \equiv (\lambda x)(\nu y)F \quad (x \neq y)$

8. $(\nu y)[x]C \equiv [x](\nu y)C \quad (x \neq y)$

9. $(\nu x)(\nu y)A \equiv (\nu y)(\nu x)A$, $(\nu x)(\nu x)A \equiv (\nu x)A$

Se define una *pseudo-aplicación* $F \bullet C$ de una abstracción a una concreción (ambas de igual aridad) de la siguiente forma. Sea $F \equiv (\lambda \vec{x}\ P)$ y $C \equiv (\nu \vec{z}\ [\vec{y}]Q)$, donde $\vec{x} \cap \vec{z} = 0$ y $|\vec{x}| = |\vec{y}|$. Entonces:

$$F \bullet C \stackrel{def}{=} (\nu \vec{z})(P\{\vec{y} / \vec{x}\} \mid Q)$$

Así, se define:

1. Comunicación:

$$(\cdots + x.F + \cdots) \mid (\cdots + \overline{x}.C + \cdots) \rightarrow F \bullet C$$

2. Paralelismo:

$$\frac{P \rightarrow P'}{P \mid Q \rightarrow P' \mid Q}$$

3. Restricción:

$$\frac{P \rightarrow P'}{(\nu x)P \rightarrow (\nu x)P'}$$

4. Congruencia estructural:

$$\frac{Q \equiv P \qquad P \rightarrow P' \qquad P' \equiv Q'}{Q \rightarrow Q'}$$

Se puede observar que la reducción se define solamente sobre procesos, no sobre agentes arbitrarios.

Un proceso normal atómico, $\alpha.A$, puede verse como una *acción* α y una *continuación* A. Se puede pensar en α como la localización de una acción. Milner lo denomina un *commitment* (i.e. un proceso "encargado" a actuar en α). Semánticamente, su idea es formalizar que un proceso, en general, no es más que un conjunto de compromisos; es decir, todo proceso es congruente semánticamente con un proceso normal $\sum \alpha_i.A_i$. El modo de formalizar esto es definiendo una relación $P \succ \alpha.A$ entre procesos y compromisos, cuyo significado es que P puede "encargarse" a $\alpha.A$. De esta forma se consigue, aunque con diferente notación, exactamente lo mismo que con el sistema de transición etiquetado de [MPW92]. Así, por ejemplo, la transición etiquetada $P \stackrel{\overline{x}y}{\rightarrow} P'$ puede expresarse como $P \succ \overline{x}.[y]P'$ y, similarmente, la transición $P \stackrel{x(y)}{\rightarrow} P'$ como $P \succ x.(\lambda y)P'$. Esta transformación no representa solamente un cambio en la notación, sino que produce una presentación más satisfactoria de la dinámica del π-cálculo.

Una vez introducido el concepto de *compromiso*, Milner define la semántica operacional en función de él. La relación de compromiso $\succ$ entre procesos y compromisos es la menor relación que satisface las siguientes reglas:

$$SUM: \quad \cdots + \alpha.A \succ \alpha.A$$

$$COM: \quad \frac{P \succ x.F \quad Q \succ \overline{x}.C}{P \mid Q \succ \tau.(F \bullet C)}$$

$$PAR: \quad \frac{P \succ \alpha.A}{P \mid Q \succ \alpha.(A \mid Q)}$$

$$RES: \quad \frac{P \succ \alpha.A}{(\nu x)P \succ \alpha.(\nu x)A} \quad (\alpha \notin \{x, \overline{x}\})$$

$$STRUCT: \quad \frac{Q \equiv P \quad P \succ \alpha.A \quad A \equiv B}{Q \succ \alpha.B}$$

Comparando con la semántica operacional dada en [MPW92], se puede comprobar que ésta es mucho más simple.

En cuanto a la semántica declarativa, redefine la bisimilaridad de una forma mucho más natural, en términos de compromisos y, para ello, previamente se formula el concepto de *respetabilidad*. Sea Ξ una relación binaria arbitraria sobre agentes. En [Mil93d] se dice que la relación Ξ es *respetable* si incluye la congruencia estructural ($\equiv$) y, además, es respetada por la descomposición de concreciones y la aplicación de abstracciones, es decir:

1. Si $C \;\Xi\; D$ entonces tiene formas estándar $C \equiv (\nu \vec{x}\, [\vec{y}]P)$ y $D \equiv (\nu \vec{x}\, [\vec{y}]Q)$ tal que $P \;\Xi\; Q$.

2. Si $F \;\Xi\; G$ entonces sus aridades coinciden, y para todo $\vec{y}$ de longitud igual a la aridad, se tiene que $F\vec{y} \;\Xi\; G\vec{y}$.

Observamos que esta noción es dual a una condición de congruencia; la relación debe preservarse por descomposición en lugar de por composición.

De este modo, en [MPW92] se redefinen las nociones de bisimulación y bisimilaridad para todos los agentes, no sólo para procesos, como sigue:

- Una relación sobre agentes, Ξ, es una *simulación fuerte* si es respetable y, además, se cumple que si $P \;\Xi\; Q$ y $P \succ \alpha.A$, entonces $Q \succ \alpha.B$ para todo B tal que $A \;\Xi\; B$.

- Ξ es una *bisimulación fuerte* si ella y su inversa son simulaciones.
- La *bisimilaridad fuerte*, $\dot{\sim}$, es la mayor de las bisimulaciones.

Se puede demostrar que todo agente, excepto la abstracción, preserva $\dot{\sim}$. Para resolver esto sólo es necesario imponer la clausura bajo sustituciones. P y Q son *fuertemente congruentes*, $P \sim Q$, si $P\sigma \dot{\sim} Q\sigma$ para cualquier sustitución σ. En [MPW92] se demuestra que $\sim$ es una congruencia.

En [BN94] se define una semántica "*fully abstract*" para la causalidad en π-cálculo. Otros artículos interesantes sobre cuestiones semánticas para el π-cálculo son [Hen91, Dam93, Eng93, MP94]. Otra línea abierta por R. Milner es la definición de un *cálculo de acciones*, que se relaciona con el π-cálculo y entre cuyas referencias podemos citar [Mil92a, Mil92b, Mil93a, Mil93b].

2.5 El λ-cálculo Etiquetado

Veamos las características más importantes del λ-cálculo Etiquetado (*label-selective λ-calculus*) de H. Aït-Kaci [AKG93a], cálculo a partir del cual se construirá el λ-cálculo Etiquetado Paralelo. Los λ-términos etiquetados están formados por variables de un conjunto $\mathcal{V}$ y dos construcciones etiquetadas: la *abstracción* y la *aplicación*. El etiquetado se realiza usando etiquetas de un conjunto $\mathcal{L}$ de etiquetas de posición. Este conjunto $\mathcal{L}$ es la unión disjunta de dos conjuntos:

- el conjunto de etiquetas numéricas $\mathcal{N} = \mathbb{N} - \{0\}$
- el conjunto de etiquetas simbólicas $\mathcal{S}$

Cada uno de estos tres conjuntos está totalmente ordenado. $\mathcal{N}$ está ordenado con el orden de los números naturales ($<_{\mathcal{N}}$); $\mathcal{S}$ está ordenado con un orden lineal ($<_{\mathcal{S}}$); y $\mathcal{L}$ está ordenado con un orden $<_{\mathcal{L}}$ tal que $<_{\mathcal{L}} = <_{\mathcal{N}}$ en $\mathcal{N}$, $<_{\mathcal{L}} = <_{\mathcal{S}}$ en $\mathcal{S}$ y $\forall (n,p) \in \mathcal{N} \times \mathcal{S}.\ n <_{\mathcal{L}} p$. Es decir, se asume que todas las etiquetas numéricas son menores que las simbólicas.

Las variables se denotan como $x, y, \ldots$, las etiquetas en $\mathcal{L}$ como $p, q, \ldots$, las etiquetas en $\mathcal{N}$ como $m, n, \ldots$, y las λ-expresiones con mayúsculas.

La sintaxis de los λ-términos etiquetados es:

$$\begin{array}{lcll} M & ::= & x & \textit{variables}, \\ & | & \lambda_p x.M & \textit{abstracciones}, \\ & | & M\widehat{_p}M & \textit{aplicaciones}. \end{array}$$

De un término $\lambda_p x.M$ decimos que "abstrae x a p en M", y de un término $M\widehat{_p}N$ que "aplica M a N a través de p". A menudo, resulta conveniente romper la atomicidad de una abstracción o una aplicación. En la abstracción $\lambda_p x.M$, a la parte $\lambda_p x$ la

llamamos *abstrayente* y a M su *cuerpo*. En la aplicación $M\widehat{_p}N$ a la parte $\widehat{_p}N$ la llamamos *aplicador*. Por *entidad* entendemos tanto a los abstrayentes o aplicadores (en cuyo caso decimos que tenemos entidades etiquetadas) como simplemente a las variables.

Las etiquetas simbólicas se ven como nombres de canales y se usan para la comunicación entre procesos, de manera similar a [Mil93d]. Las etiquetas simbólicas siempre designan posiciones absolutas de los argumentos [ACCL90]. Un proceso es un λ-término etiquetado, donde el *envío* se realiza a través de los aplicadores y la *recepción* a través de los abstrayentes. Si se ejecuta una aplicación a través de dos canales p y q diferentes, no hay ninguna ambigüedad sobre qué abstrayente los recibirá; por lo tanto, estas reducciones (comunicaciones) pueden hacerse en cualquier orden, con el mismo resultado final. Sin embargo, si se da esta situación con $p = q$, el orden en que se ejecuten será importante. En este caso, las reglas nos aseguran que la reducción respeta el orden especificado sintácticamente. En otras palabras, varios argumentos enviados a través del mismo canal se ejecutan en secuencia obligatoriamente. De hecho, la extensión del λ-cálculo motivo de esta tesis permitirá la elección no determinista en esas situaciones con la inclusión de nuevos operadores.

Si las etiquetas numéricas se usan también de forma explícita, el punto de vista que acabamos de describir se traslada de manera similar. Así, por ejemplo, una función de varios argumentos $f(a_1, \ldots, a_n)$ usa sus posiciones como proyecciones cartesianas y puede verse como $f(1 \Rightarrow a_1, \ldots, n \Rightarrow a_n)$. Esa podría ser una posible interpretación de las etiquetas numéricas en el λ-cálculo Etiquetado. Sin embargo, las etiquetas numéricas pueden tener un comportamiento distinto al de las etiquetas simbólicas interpretando que un número puede entenderse implícitamente como la primera posición relativa con respecto a lo que hay a su izquierda [dB72]. Más precisamente, la currificación [Bar91] interpreta cada argumento como el primero relativo a la forma de su izquierda. Esto tiene la ventaja de simplificar la regla de reducción funcional ya que se considera una regla local y no necesita tener en cuenta más que un único argumento cada vez. Vamos a permitir esta situación usando posiciones de argumentos relativas [GS89].

De todas formas, sería probablemente más natural el uso de posiciones absolutas. Por ejemplo, es más fácil escribir $(\lambda(1 \Rightarrow x, 2 \Rightarrow y, 4 \Rightarrow z).M)\widehat{\ }(1 \Rightarrow a, 4 \Rightarrow b)$ que $(\lambda_1 x.\lambda_1 y.\lambda_2 z.M)\widehat{_1}a\widehat{_3}b$. Sin embargo, esta versión totalmente currificada es necesaria para expresar la reducción con reglas locales, uno de los objetivos del cálculo. Afortunadamente, la traslación de la notación con etiquetas absolutas a una totalmente currificada (con etiquetas relativas) se puede realizar de forma sistemática [AKG93a], e.g. restando de cada etiqueta numérica el número de componentes etiquetados numéricamente a su izquierda en el producto cartesiano etiquetado. Por esta razón, la sintaxis del λ-cálculo Etiquetado se limita al etiquetado numérico relativo.

Las sustituciones de variables por λ-expresiones necesitan las mismas precauciones que en el λ-cálculo y obedecen exactamente a las mismas reglas. De manera habitual, se usa el signo igual (=) como igualdad sintáctica módulo α-conversión.

Sea $FV(M)$ el conjunto de variables libres en M, es decir, las variables que no aparecen en los abstrayentes de M. La expresión $[N/x]M$ denota el término que se obtiene sustituyendo todas las ocurrencias libres de la variable x por N en M. Esto es,

$$
\begin{array}{lcll}
[N/x]x & = & N & \\
[N/x]y & = & y & \text{si } y \in \mathcal{V},\, y \neq x \\
[N/x](M_1\widehat{_p}M_2) & = & ([N/x]M_1)\widehat{_p}([N/x]M_2) & \\
[N/x](\lambda_p x.M) & = & \lambda_p x.M & \\
[N/x](\lambda_p y.M) & = & \lambda_p y.[N/x]M & \text{si } y \neq x \text{ y } y \notin FV(N) \\
[N/x](\lambda_p y.M) & = & \lambda_p z.[N/x][z/y]M & \text{si } y \neq x \text{ y } y \in FV(N), \\
 & & & \text{y } z \notin FV(N) \cup FV(M)
\end{array}
$$

Se introducen tres grupos distintos de reglas de reducción: la β-reducción y dos sistemas de reordenamiento. El λ-cálculo Etiquetado es el sistema que combina libremente la β-reducción y el reordenamiento. Intuitivamente, la β-reducción para términos etiquetados puede realizarse en el momento en que una abstracción en la posición p se aplica a través de la misma posición p a un término:

$$(\lambda_p x.M)\widehat{_p}N \;\rightarrow\; [N/x]M \qquad \beta\text{-reducción}$$

Para poder hacer posible la β-reducción, en algunos casos son necesarias algunas reordenaciones de los abstrayentes y los aplicadores. Se distinguen dos conjuntos de ellas:

- las que tratan con, al menos, una etiqueta simbólica, y
- las que tratan solamente con etiquetas numéricas.

La razón para esto es clara: las etiquetas simbólicas son siempre explícitas, es decir, pueden conmutarse libremente con otras, simbólicas o numéricas, simplemente exigiendo que sean distintas; en cambio, las etiquetas numéricas necesitan conservar su coherencia relativa ya que implícitamente representan siempre el primer argumento de lo que aparece a su izquierda (el “currying”).

Las reglas para las etiquetas simbólicas adyacentes son:

$$
\begin{array}{llcll}
(1) & \lambda_p x.\lambda_q y.M & \rightarrow & \lambda_q y.\lambda_p x.M & (\textit{si } p > q) \\
(2) & M\widehat{_p}N\widehat{_q}P & \rightarrow & M\widehat{_q}P\widehat{_p}N & (\textit{si } p > q) \\
(3) & (\lambda_p x.M)\widehat{_q}N & \rightarrow & \lambda_p x.(M\widehat{_q}N) & (\textit{si } p \neq q)
\end{array}
$$

donde al menos una de las etiquetas consideradas es simbólica.

La regla (1) conmuta el orden de los abstrayentes, la regla (2) conmuta el orden de los aplicadores y la regla (3) mueve un aplicador al cuerpo de una abstracción, cuando se cumple que $(p \neq q)$, es decir, cuando no se puede aplicar la β-reducción.

Cuando lo que tenemos son dos entidades etiquetadas adyacentes y numéricas, ocurren intercambios similares a los que acabamos de ver pero preservando la coherencia relativa de las posiciones implícitas. Si tenemos que m y n son dos enteros positivos, las reglas para las etiquetas numéricas son:

$$
\begin{array}{llcll}
(4) & \lambda_m x.\lambda_n y.M & \rightarrow & \lambda_n y.\lambda_{m-1} x.M & (si\ m > n) \\
(5) & M \widehat{\ }_{m} N \widehat{\ }_{n} P & \rightarrow & M \widehat{\ }_{n} P \widehat{\ }_{m-1} N & (si\ m > n) \\
(6) & (\lambda_m x.M) \widehat{\ }_{n} N & \rightarrow & \lambda_{m-1} x.(M \widehat{\ }_{n} N) & (si\ m > n) \\
(7) & (\lambda_m x.M) \widehat{\ }_{n} N & \rightarrow & \lambda_m x.(M \widehat{\ }_{n-1} N) & (si\ m < n)
\end{array}
$$

Las reglas (3), (6) y (7) deben construirse módulo un α-renombramiento apropiado para evitar conflictos. Las reglas (4) y (5) son una traducción directa de (1) y (2).

Veamos algunos ejemplos de reducción usando etiquetas simbólicas y numéricas:

Ejemplo 1 *(Etiquetas simbólicas) Supongamos que $p < q < r < s$.*

$$
\begin{array}{llll}
 & (\lambda_p x.\lambda_q y.\lambda_r z.M)\widehat{\ }_{r} N \widehat{\ }_{s} P \widehat{\ }_{p} Q & \rightarrow_3 & (\lambda_p x.((\lambda_q y.\lambda_r z.M)\widehat{\ }_{r} N))\widehat{\ }_{s} P \widehat{\ }_{p} Q \\
\rightarrow_2 & (\lambda_p x.((\lambda_q y.\lambda_r z.M)\widehat{\ }_{r} N))\widehat{\ }_{p} Q \widehat{\ }_{s} P & \rightarrow_\beta & (\lambda_q y.\lambda_r z.[Q/x]M)\widehat{\ }_{r} [Q/x]N \widehat{\ }_{s} P \\
\rightarrow_3 & (\lambda_q y.((\lambda_r z.[Q/x]M)\widehat{\ }_{r} [Q/x]N))\widehat{\ }_{s} P & \rightarrow_\beta & (\lambda_q y.[Q/x][N/z]M)\widehat{\ }_{s} P \\
\rightarrow_3 & \lambda_q y.([Q/x][N/z]M)\widehat{\ }_{s} P & &
\end{array}
$$

Ejemplo 2 *(Etiquetas numéricas)*

$$
\begin{array}{llll}
 & (\lambda_2 x.\lambda_1 y.\lambda_2 z.M)\widehat{\ }_{4} N \widehat{\ }_{5} P \widehat{\ }_{2} Q & \rightarrow_4 & (\lambda_1 y.\lambda_1 x.\lambda_2 z.M)\widehat{\ }_{4} N \widehat{\ }_{5} P \widehat{\ }_{2} Q \\
\rightarrow_7 & (\lambda_1 y.((\lambda_1 x.\lambda_2 z.M)\widehat{\ }_{3} N))\widehat{\ }_{5} P \widehat{\ }_{2} Q & \rightarrow_5 & (\lambda_1 y.((\lambda_1 x.\lambda_2 z.M)\widehat{\ }_{3} N))\widehat{\ }_{2} Q \widehat{\ }_{4} P \\
\rightarrow_7 & (\lambda_1 y.\lambda_1 x.((\lambda_2 z.M)\widehat{\ }_{2} N))\widehat{\ }_{2} Q \widehat{\ }_{4} P & \rightarrow_\beta & (\lambda_1 y.\lambda_1 x.[N/z]M)\widehat{\ }_{2} Q \widehat{\ }_{4} P \\
\rightarrow_7 & (\lambda_1 y.((\lambda_1 x.[N/z]M)\widehat{\ }_{1} Q))\widehat{\ }_{4} P & \rightarrow_\beta & \lambda_1 y.([Q/x][N/z]M)\widehat{\ }_{4} P \\
\rightarrow_7 & \lambda_1 y.([Q/x][N/z]M \widehat{\ }_{3} P) & &
\end{array}
$$

En [AKG93a] se demuestra que el sistema de reducción construido (la unión de la β-reducción más las reglas de reordenación) es confluente.

En [CO95, LO95, CLO96] se presentan dos prototipos de un intérprete para el λ-cálculo Etiquetado construidos en Prolog y en Pascal.

En el λ-cálculo Etiquetado se refleja una concurrencia implícita entre la ejecución de los diferentes canales pero no existe la posibilidad de elección bajo un mismo canal por falta de operadores específicos para ello. Precisamente, ese será uno de nuestros objetivos, introducir el indeterminismo en la evolución del sistema para reflejar así el comportamiento de los sistemas concurrentes. Obviamente, la incorporación del indeterminismo provoca la pérdida de la propiedad de confluencia.

2.6 Ejemplos

Ilustramos la sintaxis de los diferentes cálculos introducidos mediante la resolución de dos problemas clásicos en el área: el problema de los filósofos y el problema de los teléfonos móviles. Por simplicidad, sólo veremos las soluciones usando la CHAM, el π-cálculo y CHOCS. Como la solución es indeterminista, no es posible darla usando el λ-cálculo Etiquetado de Aït-Kaci. Posteriormente, recuperaremos estos ejemplos para describir su solución en nuestra propuesta, LCEP.

2.6.1 Ejemplo 1: El problema de los Filósofos

Este es un problema clásico propuesto y resuelto inicialmente por Dijkstra, cuyo enunciado aparece en [Dij68].

Hace muchísimos años, un anciano filántropo acomodó en su palacio a cuatro eminentes filósofos. Preparó un salón como comedor para todos ellos, con una gran mesa circular rodeada de cuatro sillas, cada una de ellas etiquetada con el nombre del filósofo que la iba a ocupar. Sobre la mesa situó cuatro palillos, a la derecha de cada uno de los filósofos, pues el menú consistiría en una gran paella.

Los filósofos ocupaban gran parte de su tiempo en pensar, pero, cuando se sentían hambrientos, se acercaban al comedor, se sentaban en su silla, tomaban los palillos situados a izquierda y derecha (en un orden arbitrario) y comenzaban a comer el arroz. Al saciar el hambre dejaban los dos palillos, se levantaban de la silla y volvían a su actividad principal, pensar. Por supuesto, un palillo solamente podía ser usado en un momento determinado por un solo filósofo. Si algún otro filósofo lo estaba utilizando, debía esperar hasta que el palillo se encontrara en disposición de usar de nuevo.

Denotamos a los filósofos por: $Phil_0$, $Phil_1$, $Phil_2$, $Phil_3$. En la Figura 2.1 se muestra un esquema del problema.

En general, estamos buscando una solución simple a este problema. Evidentemente, podemos pensar que solamente necesitamos 8 agentes (los filósofos y los palillos) y cuatro canales (representados a través de los palillos). Encontramos que puede darse un bloqueo si todos los filósofos sienten necesidad de comer a la vez y, por ello, incorporamos otro agente, un camarero (`MAID`) y cuatro nuevos canales (pos_i, con $0 \leq i \leq 3$). La primera solución planteada fuerza a que los filósofos tomen siempre el palillo de su izquierda y después el de su derecha. Posteriormente, mejoraremos la solución permitiendo a los filósofos coger cualquiera de ellos en primer lugar y, lo mismo, para devolverlos a la mesa, sin ningún orden preestablecido.

Utilizamos el subíndice i con dos significados diferentes. Cuando hablamos de $\texttt{PHIL}_i$ o $\texttt{CHOP}_i$, estos son los nombres de cada filósofo y de cada palillo, pero cuando hablamos de $\texttt{MAID}_i$ éste es el número de filósofos. Los primeros son nombres y los

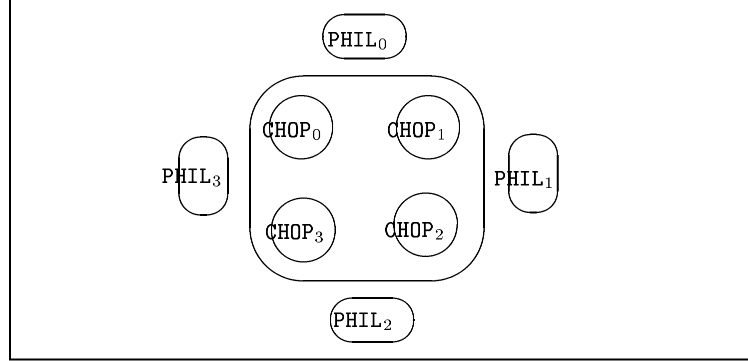


Figura 2.1: El problema de los filósofos

últimos una suma de naturales. Finalmente, definimos $i\underline{+}1$ como $i\underline{+}1 \triangleq (i+1)\ mod\ n$, donde n es el número de filósofos y el símbolo $\triangleq$ se utiliza con el significado de "por definición".

1. Máquina Química Abstracta (CHAM):
   Vamos a modelizar el palacio (COLLEGE) como una molécula formada por ocho agentes (cuatro filósofos: $\texttt{PHIL}_0$, $\texttt{PHIL}_1$, $\texttt{PHIL}_2$, $\texttt{PHIL}_3$ y cuatro palillos: $\texttt{CHOP}_0$, $\texttt{CHOP}_1$, $\texttt{CHOP}_2$, $\texttt{CHOP}_3$). Como ya hemos comentado, este problema puede provocar un bloqueo si los cuatro filósofos quieren comer al mismo tiempo. Para evitarlo, introducimos otro agente: MAID, que controla la situación de los filósofos en la mesa. Entonces, tenemos una molécula (el palacio) con nueve agentes que evoluciona calentando iones indefinidamente. Hay tres clases diferentes de canales: pos_i, $chop_i$, $chop_{i+1}$. Entonces, para definir $\texttt{PHIL}_i$ (permitiendo una elección indeterminista entre los dos posibles palillos a tomar o dejar por cada filósofo) y $\texttt{CHOP}_i$, se puede escribir:

$$\text{PHIL}_i \triangleq \{|\ pos_i.chop_i.chop_{i\underline{+}1}.\overline{chop_i}.\overline{chop_{i\underline{+}1}}.\overline{pos_i}.\text{PHIL}_i\ |\}$$

$$\text{CHOP}_i \triangleq \{|\ \overline{chop_i}.chop_i.\text{CHOP}_i\ |\}$$

   La definición de MAID precisa distinguir tres casos:

$$\begin{aligned}\text{MAID}_0 \triangleq\ & \{|\ \overline{pos_0}.\text{MAID}_1 \oplus \overline{pos_1}.\text{MAID}_1 \oplus \cdots \oplus \\ & \oplus \overline{pos_{n-1}}.\text{MAID}_1\ |\}\end{aligned}$$

$$\begin{aligned}\text{MAID}_{1\leq i\leq n-2} \;\triangleq\; & \{|\; pos_0.\text{MAID}_{i-1} \;\oplus\; pos_1.\text{MAID}_{i-1} \oplus \\ & \oplus \;\cdots\; \oplus\; pos_{n-1}.\text{MAID}_{i-1} \oplus \\ & \oplus\; \overline{pos_0}.\text{MAID}_{i+1} \;\oplus\; \overline{pos_1}.\text{MAID}_{i+1} \oplus \\ & \oplus \;\cdots\; \oplus\; \overline{pos_{n-1}}.\text{MAID}_{i+1} \;\; |\}\end{aligned}$$

$$\begin{aligned}\text{MAID}_{n-1} \;\triangleq\; & \{|\; pos_0.\text{MAID}_{n-2} \;\oplus\; pos_1.\text{MAID}_{n-2} \oplus \\ & \oplus \;\cdots\; \oplus\; pos_{n-1}.\text{MAID}_{n-2} \;\; |\}\end{aligned}$$

Y, por tanto, `COLLEGE` es una molécula como:

$$\begin{aligned}\text{COLLEGE} \;\triangleq\; & \{|\; (\text{PHIL}_0, \;\cdots\;, \text{PHIL}_{n-1}, \text{CHOP}_0, \;\cdots\;, \\ & \text{CHOP}_{n-1}, \text{MAID}_0) \\ & \backslash pos_0 \ldots pos_{n-1} chop_0 \ldots chop_{n-1} \;\; |\}\end{aligned}$$

2. Ejecución usando la CHAM:

   Veamos un ejemplo de ejecución del sistema construido. El sistema parte de la molécula `COLLEGE`. A partir de ella se pueden dar diferentes reacciones eligiendo aleatoriamente entre cualquier $\texttt{PHIL}_i$ y $\texttt{MAID}_0$. Imaginemos que la reacción que se da es entre $\texttt{PHIL}_0$ y $\texttt{MAID}_0$. Entonces, el filósofo que se sienta primero a la mesa es $\texttt{PHIL}_0$ y, por tanto, tenemos:

$$\begin{aligned}& \{|\; (chop_0.chop_1.\overline{chop_0}.\overline{chop_1}.\overline{pos_0}.\text{PHIL}_0, \text{PHIL}_1, \;\cdots\;, \\ & \text{PHIL}_{n-1}, \text{CHOP}_0, \;\cdots\;, \text{CHOP}_{n-1}, \text{MAID}_1) \\ & \backslash pos_0 \ldots pos_{n-1} chop_0 \ldots chop_{n-1} \;\; |\}\end{aligned}$$

   En estos momentos, existe la posibilidad de que se siente cualquier otro filósofo a la mesa o que el filósofo $\texttt{PHIL}_0$ tome el primer palillo. Imaginemos que ocurre esto último. Entonces, la molécula evoluciona a:

$$\begin{aligned}& \{|\; (chop_1.\overline{chop_0}.\overline{chop_1}.\overline{pos_0}.\text{PHIL}_0, \text{PHIL}_1, \;\cdots\;, \text{PHIL}_{n-1}, \\ & chop_0.\text{CHOP}_0, \text{CHOP}_1, \;\cdots\;, \text{CHOP}_{n-1}, \text{MAID}_1) \\ & \backslash pos_0 \ldots pos_{n-1} chop_0 \ldots chop_{n-1} \;\; |\}\end{aligned}$$

   Tras esto, existen de nuevo varias posibles reacciones: se puede sentar cualquier otro filósofo a la mesa o el filósofo $\texttt{PHIL}_0$ puede tomar el siguiente palillo. Supongamos que ocurre esta segunda opción, el filósofo $\texttt{PHIL}_0$ toma el segundo palillo:

$$\begin{aligned}& \{|\; (\overline{chop_0}.\overline{chop_1}.\overline{pos_0}.\text{PHIL}_0, \text{PHIL}_1, \;\cdots\;, \text{PHIL}_{n-1}, \\ & chop_0.\text{CHOP}_0, chop_1.\text{CHOP}_1, \;\cdots\;, \text{CHOP}_{n-1}, \text{MAID}_1) \\ & \backslash pos_0 \ldots pos_{n-1} chop_0 \ldots chop_{n-1} \;\; |\}\end{aligned}$$

A continuación puede ocurrir: que el filósofo $\mathtt{PHIL}_0$ deje el primer palillo que cogió, o que cualquiera del resto de filósofos tome asiento en la mesa. Imaginemos que se sienta el filósofo $\mathtt{PHIL}_1$. Entonces:

$$\{|\ (\overline{chop_0}.\overline{chop_1}.\overline{pos_0}.\mathrm{PHIL}_0,\\ chop_1.chop_2.\overline{chop_1}.\overline{chop_2}.\overline{pos_1}.\mathrm{PHIL}_1,\ \cdots\ ,\mathrm{PHIL}_{n-1},\\ chop_0.\mathrm{CHOP}_0, chop_1.\mathrm{CHOP}_1,\ \cdots\ ,\mathrm{CHOP}_{n-1},\mathrm{MAID}_2)\\ \backslash pos_0 \ldots pos_{n-1} chop_0 \ldots chop_{n-1}\ \ |\}$$

Vemos que ahora las posibilidades son: que $\mathtt{PHIL}_1$ tome un palillo, o que $\mathtt{PHIL}_0$ deje el primer palillo que cogió, o que se siente cualquiera del resto de filósofos. Y así, sucesivamente.

La evolución del sistema puede seguir indefinidamente, de forma similar a como acabamos de mostrar.

3. π-cálculo:

Siguiendo un planteamiento similar al del punto 1, tenemos:

$$\mathrm{PHIL}_i \triangleq pos_i.chop_i.chop_{i\underline{+}1}.\overline{chop_i}.\overline{chop_{i\underline{+}1}}.\overline{pos_i}.\mathrm{PHIL}_i$$

$$\mathrm{CHOP}_i \triangleq \overline{chop_i}.chop_i.\mathrm{CHOP}_i$$

$$\mathrm{MAID}_0 \triangleq \overline{pos_0}.\mathrm{MAID}_1 + \overline{pos_1}.\mathrm{MAID}_1 + \cdots + \\ + \overline{pos_{n-1}}.\mathrm{MAID}_1$$

$$\mathrm{MAID}_{1\leq i\leq n-2} \triangleq pos_0.\mathrm{MAID}_{i-1} + pos_1.\mathrm{MAID}_{i-1} + \\ + \cdots + pos_{n-1}.\mathrm{MAID}_{i-1} + \\ + \overline{pos_0}.\mathrm{MAID}_{i+1} + \overline{pos_1}.\mathrm{MAID}_{i+1} + \\ + \cdots + \overline{pos_{n-1}}.\mathrm{MAID}_{i+1}$$

$$\mathrm{MAID}_{n-1} \triangleq pos_0.\mathrm{MAID}_{n-2} + pos_1.\mathrm{MAID}_{n-2} + \\ + \cdots + pos_{n-1}.\mathrm{MAID}_{n-2}$$

Con estos procesos, el palacio se representa como:

$$\mathrm{COLLEGE} \triangleq \mathrm{PHIL}_0 \mid \cdots \mid \mathrm{PHIL}_{n-1} \mid \\ \mid \mathrm{CHOP}_0 \mid \cdots \mid \mathrm{CHOP}_{n-1} \mid \mathrm{MAID}_0$$

Ahora, vamos a cambiar los procesos $\texttt{PHIL}_i$ para permitir la elección de uno de los palillos sin ningún orden preestablecido.

$$\text{PHIL}_i \triangleq pos_i.\text{PHIL'}_i$$

$$\text{PHIL'}_i \triangleq chop_i.chop_{i\underline{+}1}.\text{PHIL''}_i + chop_{i\underline{+}1}.chop_i.\text{PHIL''}_i$$

$$\text{PHIL''}_i \triangleq \overline{chop_i}.\overline{chop_{i\underline{+}1}}.\overline{pos_i}.\text{PHIL}_i + \overline{chop_{i\underline{+}1}}.\overline{chop_i}.\overline{pos_i}.\text{PHIL}_i$$

4. <u>CHOCS</u>:

$$\text{PHIL}_i \triangleq pos_i?.\text{PHIL'}_i$$

$$\text{PHIL'}_i \triangleq chop_i?.chop_{i\underline{+}1}?.\text{PHIL''}_i + chop_{i\underline{+}1}?.chop_i?.\text{PHIL''}_i$$

$$\text{PHIL''}_i \triangleq chop_i!.chop_{i\underline{+}1}!.pos_i!.\text{PHIL}_i + chop_{i\underline{+}1}!.chop_i!.pos_i!.\text{PHIL}_i$$

$$\text{CHOP}_i \triangleq chop_i!.chop_i?.\text{CHOP}_i$$

$$\text{MAID}_0 \triangleq pos_0!.\text{MAID}_1 + pos_1!.\text{MAID}_1 + \cdots + pos_{n-1}!.\text{MAID}_1$$

$$\text{MAID}_{1\leq i\leq n-2} \triangleq pos_0?.\text{MAID}_{i-1} + pos_1?.\text{MAID}_{i-1} + \cdots + pos_{n-1}?.\text{MAID}_{i-1} + pos_0!.\text{MAID}_{i+1} + pos_1!.\text{MAID}_{i+1} + \cdots + pos_{n-1}!.\text{MAID}_{i+1}$$

$$\text{MAID}_{n-1} \triangleq pos_0?.\text{MAID}_{n-2} + pos_1?.\text{MAID}_{n-2} + \cdots + pos_{n-1}?.\text{MAID}_{n-2}$$

$$\text{COLLEGE} \triangleq \text{PHIL}_0 \mid \cdots \mid \text{PHIL}_{n-1} \mid \text{CHOP}_0 \mid \cdots \mid \text{CHOP}_{n-1} \mid \text{MAID}_0$$

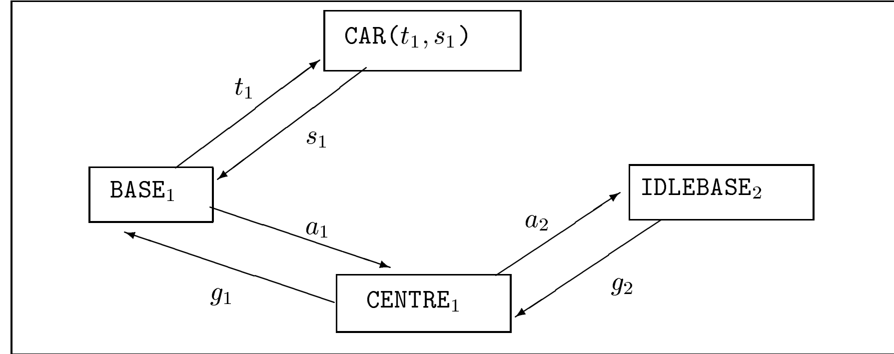


Figura 2.2: Teléfonos móviles

2.6.2 Ejemplo 2: Teléfonos móviles

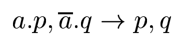
Este ejemplo es una versión simplificada de un sistema usado por Orava y Parrow [OP90, Mil93d, OP92].

Un centro de comunicación (`CENTRE`*) está en contacto permanente con dos estaciones (*`BASE`*), cada una en un lugar diferente de un pais. Un coche (*`CAR`*), que dispone de un teléfono móvil, se mueve a lo largo de ese pais. El coche debe permanecer siempre en contacto con una* `BASE`*. Si en algún momento el coche se encuentra muy alejado del contacto con la* `BASE` *actual, entonces (de una forma que no importa en la modelización) se genera un proceso que da, como resultado, que el coche pierde el contacto con esa* `BASE` *y comienza el contacto con la otra*[1].

En la Figura 2.2 se presenta un esquema de la situación.

1. Máquina Química Abstracta (CHAM)[2]:

   Es imposible modelizar la idea de los canales parametrizados utilizando la CHAM. La regla de reacción trata con *iones* (*valencias*) de los procesos. Siempre que dos iones complementarios flotan en la solución es posible que reaccionen en la solución, pero sin ningún envío de información durante el proceso. Las valencias desaparecen. La regla de reacción, como ya vimos, es:

   
   $$a.p, \overline{a}.q \rightarrow p, q$$

[1] Notación: las variables t_i, s_i, g_i, a_i representan $talk_i$, $switch_i$, $give_i$, $alert_i$, respectivamente.

[2] Para el razonamiento que sigue se parte de la sintaxis de la TCCS CHAM completa ([BB93], pp. 87).

y podemos notar que nada afecta a p o q. Por lo tanto, no existe una solución válida a este problema utilizando la CHAM.

2. <u>π-cálculo</u>[3]:

   En nuestro sistema `SYSTEM` existen cuatro agentes (`CAR`, `BASE`, `IDLEBASE`, `CENTRE`) y cuatro canales (*talk*, *switch*, *give*, *alert*) para cada `BASE`.

   El grafo de la Figura 2.2 muestra el sistema en un estado en el que el coche se encuentra en contacto con la base $\texttt{BASE}_1$ y, por lo tanto, la base $\texttt{BASE}_2$ está desactivada:

$$\begin{aligned}\mathrm{SYSTEM}_1 \;&\triangleq\; (\nu t_1 t_2 s_1 s_2 g_1 g_2 a_1 a_2)\\ &\quad(\mathrm{CAR}(t_1, s_1) \mid \mathrm{BASE}_1 \mid \mathrm{IDLEBASE}_2 \mid \mathrm{CENTRE}_1)\end{aligned}$$

   El coche `CAR` está parametrizado con los canales *talk* y *switch*. En cualquier momento, desde el coche se puede hablar usando un canal o el canal de comunicación puede cambiarse usando el respectivo interruptor:

$$\mathrm{CAR}(t, s) \;\triangleq\; t.\mathrm{CAR}(t, s) \;+\; s(t's').\mathrm{CAR}(t', s')$$

   Una base `BASE` puede comunicarse con el coche, pero si recibe dos nuevos canales a través del canal *switch*, cambia y pasa a estar desactivada:

$$\begin{aligned}\mathrm{BASE}(t, s, g, a) \;&\triangleq\; \overline{t}.\mathrm{BASE}(t, s, g, a) \;+\\ &\quad+\; g(t's').\overline{s}\; t's'\; .\mathrm{IDLEBASE}(t, s, g, a)\end{aligned}$$

   Si una base está desactivada, puede convertirse en activa usando el canal *alert*:

$$\mathrm{IDLEBASE}(t, s, g, a) \;\triangleq\; a.\mathrm{BASE}(t, s, g, a)$$

   Por todo esto, el centro `CENTRE` se define como:

$$\begin{aligned}\mathrm{CENTRE}_1 \;&\triangleq\; \overline{g_1}\; t_2 s_2\; .\overline{a_2}.\mathrm{CENTRE}_2\\ \mathrm{CENTRE}_2 \;&\triangleq\; \overline{g_2}\; t_1 s_1\; .\overline{a_1}.\mathrm{CENTRE}_1\end{aligned}$$

3. <u>Ejecución usando el π-cálculo</u>:

   Veamos cómo es una posible ejecución del sistema. Como acabamos de describir, el sistema de partida es el siguiente:

$$\begin{aligned}\mathrm{SYSTEM}_1 \;&\triangleq\; (\nu t_1 t_2 s_1 s_2 g_1 g_2 a_1 a_2)\\ &\quad(\mathrm{CAR}(t_1, s_1) \mid \mathrm{BASE}_1 \mid \mathrm{IDLEBASE}_2 \mid \mathrm{CENTRE}_1)\end{aligned}$$

[3]Esta solución está tomada de [Mil93d], pp. 11-13.

donde cada uno de los subprocesos que intervienen se ha definido como sigue:

$$\mathrm{CAR}(t_1, s_1) \triangleq t_1.\mathrm{CAR}(t_1, s_1) + s_1(t's').\mathrm{CAR}(t', s')$$

$$\mathrm{BASE}(t_1, s_1, g_1, a_1) \triangleq \overline{t_1}.\mathrm{BASE}(t_1, s_1, g_1, a_1) +$$
$$+ g_1(t's').\overline{s_1}\ t's'\ .\mathrm{IDLEBASE}(t_1, s_1, g_1, a_1)$$

$$\mathrm{IDLEBASE}(t_2, s_2, g_2, a_2) \triangleq a_2.\mathrm{BASE}(t_2, s_2, g_2, a_2)$$

$$\mathrm{CENTRE}_1 \triangleq \overline{g_1}\ t_2 s_2\ .\overline{a_2}.\mathrm{CENTRE}_2$$

Al comenzar la ejecución se dan dos posibles comunicaciones:

(a) De $\mathtt{CAR}(t_1, s_1)$ con $\mathtt{BASE}(t_1, s_1, g_1, a_1)$ a través del canal t_1, lo que se interpreta como que el coche sigue en contacto con la base de partida; o bien

(b) De $\mathtt{BASE}(t_1, s_1, g_1, a_1)$ con $\mathtt{CENTRE}_1$ a través del canal g. Si ocurre esto, el coche dejará de estar en contacto con la base inicial y pasará a estarlo con $\mathtt{BASE}(t_2, s_2, g_2, a_2)$.

Si el sistema selecciona la primera posibilidad, se da la comunicación:

$$t_1.\mathrm{CAR}(t_1, s_1) \mid \overline{t_1}.\mathrm{BASE}(t_1, s_1, g_1, a_1)$$

y el sistema recupera su estado inicial:

$$\mathrm{SYSTEM}_1 \triangleq (\nu t_1 t_2 s_1 s_2 g_1 g_2 a_1 a_2)$$
$$(\mathrm{CAR}(t_1, s_1) \mid \mathrm{BASE}_1 \mid \mathrm{IDLEBASE}_2 \mid \mathrm{CENTRE}_1)$$

Si, por el contrario, se da la otra situación, es decir, la comunicación de $\mathtt{BASE}(t_1, s_1, g_1, a_1)$ con $\mathtt{CENTRE}_1$ a través del canal g, la comunicación en este caso es:

$$g_1(t's').\overline{s_1}\ t's'\ .\mathrm{IDLEBASE}(t_1, s_1, g_1, a_1) \mid$$
$$\mid \overline{g_1}\ t_2 s_2\ .\overline{a_2}.\mathrm{CENTRE}_2$$

y el sistema evoluciona a:

$$(\nu t_1 t_2 s_1 s_2 g_1 g_2 a_1 a_2)$$
$$(\mathrm{CAR}(t_1, s_1) \mid \overline{s_1}\ t_2 s_2\ .\mathrm{IDLEBASE}(t_1, s_1, g_1, a_1) \mid$$
$$\mid \mathrm{IDLEBASE}_2 \mid \overline{a_2}.\mathrm{CENTRE}_2)$$

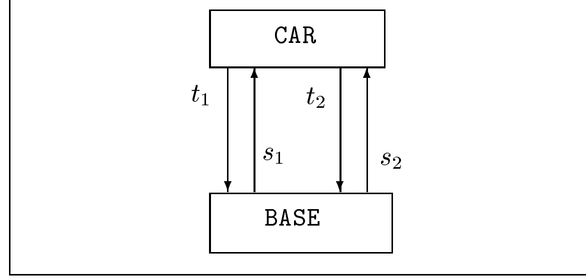


Figura 2.3: Simplificación de los Teléfonos Móviles

Ahora, teniendo en cuenta que la definición de $\texttt{IDLEBASE}_2$ es la siguiente: $a_2.\texttt{BASE}(t_2, s_2, g_2, a_2)$, sólo es posible la comunicación:

$$\overline{a_2}.\mathrm{CENTRE}_2 \ \mid \ a_2.\mathrm{BASE}(t_2, s_2, g_2, a_2)$$

y, por tanto, el sistema evoluciona al estado:

$$(\nu t_1 t_2 s_1 s_2 g_1 g_2 a_1 a_2)$$
$$(\mathrm{CAR}(t_1, s_1) \mid \mathrm{IDLEBASE}_1 \mid \mathrm{BASE}_2 \mid \mathrm{CENTRE}_2)$$

en donde el coche ha pasado a contactar a través de la segunda base, evolucionando la primera a inhabilitada.

Siguiendo la ejecución, el sistema evolucionará indefinidamente conservando activa una de las bases o inhabilitando una de ellas y pasando a activa la otra.

4. CHOCS:

Representamos con una variable una tupla de valores.

Vamos a utilizar una versión simplificada del problema (ver Figura 2.3), donde existe solamente un coche `CAR` y una base `BASE`, con cuatro canales (t_1, s_1, t_2, s_2). De este modo, será más fácil entender la solución planteada aunque seguimos conservando las principales propiedades del problema.

La especificación del sistema simplificado se escribe:

$$\mathrm{SYSTEM} \ \triangleq \ \mathrm{CAR}(t_1, s_1) \ \mid \ \mathrm{BASE}_1$$

La solución general es:

$$\mathrm{CAR}(t, s) \ \triangleq \ t?x.\mathrm{CAR}(x, s) \ + \ s?y.\mathrm{CAR}(y)$$
$$\mathrm{BASE}_i \ \triangleq \ t!t.\mathrm{BASE}_i \ + \ s![t_i, s_i].\mathrm{BASE}_{i+1}$$

Entonces:

$$\begin{aligned} \mathrm{CAR}(t_1, s_1) &\triangleq t_1?x.\mathrm{CAR}(x, s_1) \;+\; s_1?y.\mathrm{CAR}(y) \\ \mathrm{CAR}(t_2, s_2) &\triangleq t_2?x.\mathrm{CAR}(x, s_2) \;+\; s_2?y.\mathrm{CAR}(y) \end{aligned}$$

$$\begin{aligned} \mathrm{BASE}_1 &\triangleq t_1!t_1.\mathrm{BASE}_1 \;+\; s_1![t_2, s_2].\mathrm{BASE}_2 \\ \mathrm{BASE}_2 &\triangleq t_2!t_2.\mathrm{BASE}_2 \;+\; s_2![t_1, s_1].\mathrm{BASE}_1 \end{aligned}$$

Capítulo 3

El λ-cálculo Etiquetado Paralelo (LCEP)

Como primera aportación significativa de esta tesis doctoral se presenta un nuevo cálculo para la modelización de sistemas paralelos, el *λ-cálculo Etiquetado Paralelo* (LCEP) [LO94b], extensión del λ-cálculo Etiquetado de H. Aït-Kaci. Una diferencia esencial con respecto a otras aproximaciones está en el hecho de permitir el uso tanto de etiquetas simbólicas como numéricas para nombrar a los canales de comunicación, situación no permitida hasta ahora en el resto de aproximaciones y que posibilita la utilización de la currificación.

Ejemplo 3 *Una noción introducida en el contexto del λ-cálculo por Haskell Curry es la* currificación *[Bar91]. Por ejemplo, la expresión $x - y$ determina dos funciones h y k de dos variables definidas como $h(x,y) = x - y$ y $k(y,x) = x - y$, que pueden representarse como λ-términos como $h = \lambda x.\lambda y.x - y$ y $k = \lambda y.\lambda x.x - y$.*

Podemos, sin embargo, evitar la necesidad de una notación especial para funciones de varias variables utilizando funciones cuyos valores son, a su vez, otras funciones. Por ejemplo, en lugar de la función de dos parámetros h, se puede considerar la función de un parámetro h^*, definida como $h^* = \lambda x.(\lambda y.x - y)$. Esta función define, para cada valor a, una función especializada de un parámetro $h^* = \lambda y.a - y$, y, por tanto, para cada par de números a, b, $(h^*(a))(b) \ \triangleright_\beta \ a - b \ = \ h(a, b)$.

La utilización de la *currificación* para representar funciones n-arias se apoya en el isomorfismo [AKG93b] existente entre dichas funciones y las funciones cuyo dominio es uno de los conjuntos y cuyo rango es una función de aridad $n-1$ con el mismo rango que la función original, es decir, $A_1 \times A_2 \ldots \times A_n \to B \ \simeq \ A_1 \to (A_2 \times \ldots \times A_n \to B)$.

Ejemplo 4 *Para las funciones binarias, $A \times B \to C \ \simeq \ A \to (B \to C)$, es decir, una función cuyo dominio es $A \times B$ y cuyo rango es C es isomorfa a una función*

cuyo dominio es A y cuyo rango es el conjunto de funciones de B en C. h representa una función $h: A \times B \to C$, mientras que h^* representa una función isomorfa a ésta, $h^*: A \to (B \to C)$. h^* es la versión currificada *de h.*

Ejemplo 5 *Tomemos de nuevo la función definida en el ejemplo 3. Si denotamos la definición formal de h como $h = \lambda x.\lambda y.x - y$, entonces la aplicación de h a un par de parámetros (a, b) se representa como $h(a, b)$ y se interpreta de forma natural como $(h^*(a))(b)$. Implícitamente asumimos que a se asocia con x y b con y y así ocurre de acuerdo con la forma de h y el mecanismo de β-reducción.*

Pero si deseamos que $h(a, b) = h(b \to y, a \to x)$, es decir, queremos hacer explícita *la asociación de los parámetros con las variables, sin cambiar el resultado final, el λ-cálculo no lo permite de forma directa:*

$$((\lambda x.\lambda y.x - y)b)a \;=\; b - a \;\neq\; a - b \;=\; h(a, b)$$

Para resolver esta cuestión, es necesario cambiar la expresión de h o manipular el orden de aplicación de los parámetros. Este problema ya aparece resuelto en el λ-cálculo Etiquetado [AKG93b].

Pero, a pesar de que en [AKG93b] se refleja un paralelismo implícito entre la ejecución de los diferentes canales, no existe la posibilidad de elección bajo un mismo canal porque hacen falta operadores específicos que no existen en el cálculo. LCEP extiende su sintaxis introduciendo el indeterminismo. La inclusión de nuevos operadores (el de paralelismo $\|$, el de secuencialidad $\circ$, el de elección no determinista $+$ y el de replicación $!$) y la introducción de nuevos conceptos (en particular, el concepto de túnel) permite al nuevo cálculo expresar de forma directa el paralelismo.

3.1 El lenguaje LCEP

Sea $\mathcal{V} = \{x, y, z, \ldots\}$ un conjunto de variables, $\mathcal{C} = \{a, b, c, \ldots\}$ un conjunto de constantes, P representa un nombre de canal y pertenece a un conjunto de etiquetas $\mathcal{L} = \mathcal{N} \cup \mathcal{S}$, donde $m, n, \ldots$ denotan etiquetas numéricas tomadas de $\mathcal{N}$ (el conjunto de los números naturales), $\mathcal{S} = \{p, q, \ldots\}$ es un conjunto de etiquetas simbólicas y se cumple que $\mathcal{N} \cap \mathcal{S} = \emptyset$ y 0 representa al proceso nulo. El lenguaje del sistema formal que presentamos es $\mathcal{M}$, y se define inductivamente como sigue:

$$\begin{array}{lcl} M & ::= & 0 \mid a \mid x \mid \lambda_P x.M \mid \widehat{P} M \mid \\ & & (M \parallel M) \mid (M \circ M) \mid (M + M) \mid !M \end{array}$$

$$P \quad ::= \quad \textit{número} \mid \textit{símbolo}$$

De forma general, denominamos λ_{EP}-términos a los términos del lenguaje así construidos.

La utilización de etiquetas tanto simbólicas como numéricas permite resolver el problema de la especificación etiquetada de parámetros, facilidad que presentan muchos lenguajes de programación. Mediante la especificación etiquetada de parámetros, independizamos el orden de escritura de los parámetros reales para una función del orden de presentación de los parámetros formales en la definición de la misma.

3.1.1 Interpretación de los símbolos del lenguaje

En lo que sigue, interpretamos que los términos del lenguaje $\mathcal{M}$ representan *procesos* [San93b]. Por lo tanto, a los símbolos $\circ, \|, +, !$ los consideramos como *constructores de procesos*. También los operadores de comunicación $\lambda_P x$ (abstracción o *input*) y $\widehat{P}$ (*output*) los vemos como *constructores de procesos*, cuyo efecto sobre un proceso M es definir un nuevo proceso ($\lambda_P x.M$ y $\widehat{P}M$, respectivamente) capaz de establecer una *comunicación* bajo determinadas condiciones. Las constantes, variables y el proceso nulo (0) constituyen los elementos *atómicos* del lenguaje.

Informalmente, la interpretación de los distintos constructores de procesos es la siguiente:

1. Constructor de *secuencialidad* ($\circ$): Crea un proceso $M \circ N$ a partir de los procesos M y N. El proceso N se ejecuta a continuación de M.

2. Constructor de *paralelismo* ($\|$): Define un nuevo proceso $M \| N$ a partir de los procesos M y N. Ambos procesos se ejecutan simultáneamente.

3. Elección *no determinista* ($+$): $M + N$ expresa la posibilidad de que M o N (uno de los dos) tome parte en una operación de comunicación o reordenación. Una comunicación puede concretarse en cualquiera de los procesos que lo integran, desapareciendo el resto.

4. Operador de *replicación* ($!$): $!M$ define un proceso que permite la aparición repetida de M. Constituye una especie de "almacén" del cual se puede tomar, de modo indefinido, un proceso M.

Los operadores obedecen el siguiente orden de prioridad, de mayor a menor:

$$! \quad > \quad \widehat{P} \quad > \quad \circ \quad > \quad \lambda_P x \quad > \quad + \quad > \quad \|$$

Una ocurrencia de la variable x aparece *ligada* en un proceso M si y solamente si se encuentra en un término como $\lambda_P x.M$ (para todo $P \in \mathcal{L}$). En cualquier otro caso, la ocurrencia de x es *libre*. Si x tiene al menos una ocurrencia libre en M decimos que x es una *variable libre* de M. Designamos como $FV(M)$ al conjunto de variables libres

del proceso M. Decimos que un proceso es *cerrado* si es un término sin variables libres.

Las relaciones entre los procesos vienen descritas en términos de *comunicación*. Los constructores $\lambda_P x$ y $\widehat{P}$ dan a dos procesos M y N la posibilidad de comunicarse a través de un canal etiquetado con $P \in \mathcal{L}$, de la siguiente forma:

1. Constructor de *input* ($\lambda_P x$): El proceso $\lambda_P x.M$ puede *recibir* por el canal P un proceso que sustituye en M las ocurrencias libres de x por el proceso entrante a través del canal.

2. Constructor de *output* ($\widehat{P}$): El proceso $\widehat{P}N$ puede *enviar* por el canal P un proceso N.

3.1.2 Relación de orden en el conjunto de etiquetas

En el conjunto de etiquetas $\mathcal{L}$ se establece una relación de *orden parcial* [Pra86], que denotamos como $\preceq_{\mathcal{L}}$. Se exige como condición para la relación que la etiqueta numérica 0 sea el mínimo del conjunto ordenado. Empleamos la relación de orden parcial en el conjunto de etiquetas, entre otras cosas, para representar el orden de aplicación de los parámetros reales a los parámetros formales de una función, análogamente a como se hace en el λ-cálculo para tratar el problema de la parametrización de llamadas a función. Sin embargo, es mucho más importante para establecer las relaciones de comunicabilidad al extender la comunicación paralela mediante el concepto de túnel (ver Sección 3.7).

En LCEP podemos reescribir la función h del ejemplo 3 como:

$$h(p,q) \quad = \quad \lambda_p x.\lambda_q y.x - y$$

Ahora hemos *dado nombre* a los parámetros formales de la función h mediante las etiquetas $p, q \in \mathcal{S}$. Podemos especificar la aplicación de h a dos parámetros a, b particularizando el destino de dichos parámetros. Por ejemplo,

$$h(a,b) = (\lambda_p x.\lambda_q y.x - y) \circ \widehat{q}b \circ \widehat{p}a = h(b \to q, a \to p)$$

Para que el axioma de comunicación secuencial (β_{Sec}, ver Sección 3.4) permita obtener el resultado deseado: $h(a,b) = a - b$, tenemos que utilizar los axiomas de reordenación (Sección 3.5), y definir $p \preceq q$.

3.2 Interpretación algebraica

Podemos presentar el formalismo de manera explícita en un contexto algebraico [BK85, HM85] interpretando los constructores de proceso como operaciones de una signatura homogénea (*one-sorted*):

1. Los constructores de procesos $\circ, \|, +, !$ son símbolos de operación.

2. Se define $\mathcal{P} = \{\widehat{P} : P \in \mathcal{L}\}$, un conjunto de símbolos de operación de output indexado por $\mathcal{L}$.

3. Se define $\Lambda = \{\lambda_P x : (P \in \mathcal{L}) \wedge (x \in \mathcal{V})\}$, un conjunto de símbolos de operación de input indexado por $\mathcal{L} \times \mathcal{V}$.

La *signatura* definida es $\Sigma_{\mathcal{M}} = \{0\} \cup \mathcal{C} \cup \mathcal{P} \cup \Lambda \cup \{!, \circ, \|, +\}$.

El perfil de los diferentes símbolos de operación viene definido por:

$$
\begin{array}{lll}
0 & : \ \to \mathcal{M} & \\
a & : \ \to \mathcal{M} & \forall a \in \mathcal{C} \\
\chi\ _ & : \ \mathcal{M} \to \mathcal{M} & \forall \chi \in \mathcal{P} \\
\kappa\ _ & : \ \mathcal{M} \to \mathcal{M} & \forall \kappa \in \Lambda \\
!\ _ & : \ \mathcal{M} \to \mathcal{M} & \\
_ \circ _ & : \ \mathcal{M} \times \mathcal{M} \to \mathcal{M} & \\
_ \| _ & : \ \mathcal{M} \times \mathcal{M} \to \mathcal{M} & \\
_ + _ & : \ \mathcal{M} \times \mathcal{M} \to \mathcal{M} &
\end{array}
$$

La *aridad* de los símbolos de operación viene determinada por la siguiente función:

$$ar : \Sigma_{\mathcal{M}} \to \mathbb{N}$$

El conjunto de términos para $\Sigma_{\mathcal{M}} \cup \mathcal{V}$, que llamamos $\mathcal{T}_{\Sigma_{\mathcal{M}} \cup \mathcal{V}}$, está en biyección con el lenguaje $\mathcal{M}$, es decir, $\mathcal{T}_{\Sigma_{\mathcal{M}} \cup \mathcal{V}} \simeq \mathcal{M}$. La función (biyectiva) $T : \mathcal{M} \to \mathcal{T}_{\Sigma_{\mathcal{M}} \cup \mathcal{V}}$ asocia a cada proceso M el término correspondiente $T(M)$ en $\mathcal{T}_{\Sigma_{\mathcal{M}} \cup \mathcal{V}}$.

Para trabajar con variables de proceso introducimos un conjunto (no vacío) de variables $\mathcal{W} = \{L, M, N, \ldots\}$, con $\mathcal{W} \cap \mathcal{V} = \emptyset$. El conjunto de términos con variables sobre $\Sigma_{\mathcal{M}} \cup \mathcal{V} \cup \mathcal{W}$ se denota como $\mathcal{T}_{\Sigma_{\mathcal{M}} \cup \mathcal{V}}(\mathcal{W})$. Lógicamente, $\mathcal{T}_{\Sigma_{\mathcal{M}} \cup \mathcal{V}} \subset \mathcal{T}_{\Sigma_{\mathcal{M}} \cup \mathcal{V}}(\mathcal{W})$.

Podemos entonces vincular a cada proceso $M \in \mathcal{M}$ el árbol sintáctico del término $T(M)$ que lo representa en $\mathcal{T}_{\Sigma_{\mathcal{M}} \cup \mathcal{V}}$. Un término del conjunto $\mathcal{T}_{\Sigma_{\mathcal{M}} \cup \mathcal{V}}(\mathcal{W})$ es un esquema de proceso. En general, no distinguimos entre proceso y esquema de proceso más que cuando es necesario y manejamos únicamente procesos. Cuando hablamos del *árbol sintáctico* de M nos referimos en realidad al árbol sintáctico de $T(M)$. Por ejemplo, al proceso $\lambda_P x.M \;\|\; \widehat{_P}N$ se le asocia el árbol sintáctico que se muestra en la Figura 3.1:

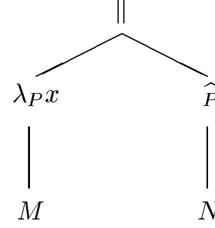


Figura 3.1: Arbol sintáctico del proceso $\lambda_P x \;\|\; \widehat{_P}N$

A partir de aquí podemos utilizar las nociones habituales de *ocurrencia*, *conjunto de ocurrencias*, *subtérmino*, ... para referirnos a procesos (términos) y subprocesos (subtérminos), [Sco90].

Para formalizar la noción de árbol sintáctico (en el conjunto $\mathcal{T}_{\Sigma_{\mathcal{M}} \cup \mathcal{V}}(\mathcal{W})$) recordamos las siguientes definiciones:

Definición 3.2.1 (Ocurrencia) *Una ocurrencia $u \in (\mathbb{N} - \{0\})^*$ es una secuencia de números naturales. El conjunto $\Theta = \{u \mid u \in (\mathbb{N} - \{0\})^*\}$ es el lenguaje generado mediante la operación de concatenación de símbolos cuando se utiliza el conjunto de los números naturales no nulos como alfabeto. Denotamos la operación de concatenación en forma infija, como es habitual, pero empleamos una forma visible del operador: ".", es decir, escribimos 1.2.3 en lugar de 123. ϵ representa la cadena vacía. Una ocurrencia es, por tanto, una palabra del lenguaje Θ.*

Extendemos la operación $. : L \times L \rightarrow L$, de concatenación de cadenas en un lenguaje L, a una operación (con el mismo nombre) $. : L \times 2^L \rightarrow 2^L$, de cadenas con subconjuntos de cadenas de L en subconjuntos de cadenas de L, como:

$$\alpha.\mathrm{L} = \{\alpha.\beta : \beta \in \mathrm{L}\} \quad \text{siendo } \mathrm{L} \in L$$

Definición 3.2.2 (Orden parcial sobre ocurrencias) *Se define una relación de orden parcial $\trianglelefteq$ sobre Θ como el cierre reflexivo y transitivo de la siguiente relación:*

$$R_{\trianglelefteq} = \{\langle u, u.i\rangle : i \in (\mathbb{N} - \{0\})^* \;\wedge\; u \in \Theta\}$$

o, de forma equivalente, como el orden de prefijos en el lenguaje Θ:

$$u \trianglelefteq v \Leftrightarrow \exists w \in \Theta : u.w = v$$

Definición 3.2.3 (Conjunto de ocurrencias de un término) *Sea una signatura homogénea Σ (provista de una función de aridad $ar : \Sigma \rightarrow \mathbb{N}$), un conjunto de variables $V = \{x, y, \ldots\}$ y $T_\Sigma(V)$ el conjunto de términos con variables generado libremente por la signatura. Se define el conjunto de ocurrencias $O(t)$ de un término $t \in T_\Sigma(V)$ mediante una función $O : T_\Sigma(V) \rightarrow 2^\Theta$ del conjunto de términos con variables $T_\Sigma(V)$ en el conjunto de partes de Θ.*

La función queda definida inductivamente como sigue:

1. $O(x) = \{\epsilon\} \quad x \in V$.
2. $O(f(t_1, t_2, \ldots, t_k)) = \{\epsilon\} \cup \bigcup_{i \in \{1,2,\ldots,k\}} i.O(t_i)$

$i.O(t_i)$ representa la extensión de la concatenación de ocurrencias a concatenación de ocurrencias y conjuntos de ocurrencias, como en la Definición 3.2.1, tomando i como una cadena de un único símbolo.

Definición 3.2.4 (Subtérmino a una ocurrencia) *Dado un término $t \in T_\Sigma(V)$ y una ocurrencia $u \in O(t)$, se define el subtérmino de t a la ocurrencia u, escrito t/u, como:*

1. $t/\epsilon = t$
2. $f(t_1, t_2, \ldots, t_i, \ldots, t_k)/i.u = t_i/u \quad$ *si* $1 \leq i \leq k$

Definición 3.2.5 (Arbol sintáctico de un término) *Se define el árbol sintáctico (abreviado: AS) $\mathcal{A}(t)$ asociado a un término $t \in T_\Sigma(V)$ como un* grafo *$\mathcal{A}(t) = (N(t), A(t))$, siendo $N(t)$ un conjunto de* nodos *y $A(t)$ un conjunto de* arcos, *tomados del producto cartesiano $N(t) \times N(t)$ [Ber83].*

Para cada término t podemos establecer ambos conjuntos como sigue:

1. *El conjunto de nodos $N(t) = O(t)$; es decir, el conjunto de nodos viene definido por el* conjunto de ocurrencias *del término.*
2. *El conjunto de arcos $A(t)$ se define también en función de $O(t)$:*

$$A(t) \quad = \quad \{\langle u, u.i\rangle : \; i \in \mathbb{N} - \{0\} \; \wedge \; u, u.i \in O(t)\}$$

El grafo asociado con un término constituye un grafo *dirigido acíclico*, y lo denominaremos simplemente árbol, [Ber83]. La ocurrencia ϵ define la raíz de dicho árbol. En ocasiones, consideramos los nodos del árbol unidos por arcos bidireccionales, dando lugar a la siguiente definición.

Definición 3.2.6 (Arbol sintáctico simétrico de un término) *El árbol sintáctico simétrico (abreviado: AS°) $\mathcal{A}^\circ(t) = (N(t), A^\circ(t))$ de un término $t \in T_\Sigma(V)$ es el árbol obtenido a partir del árbol sintáctico de t, $\mathcal{A}(t)$, completando todo arco $\langle u, v\rangle$ con un arco simétrico $\langle v, u\rangle$: $A^\circ(t) = A(t) \; \cup \; \{\langle v, u\rangle : \; \langle u, v\rangle \in A(t)\}$.*

Definición 3.2.7 (Función de determinación simbólica) *Dado un término cualquiera $t \in T_\Sigma(V)$ y su conjunto de ocurrencias $O(t)$, la función de determinación simbólica de t es una función $\omega_t : \; O(t) \rightarrow (\Sigma \cup V)$ definida como:*

1. $\omega_x(\epsilon) = x \qquad$ *para* $x \in V$
2. $\omega_{f(t_1,t_2,\ldots,t_k)}(\epsilon) = f$
3. $\omega_t(u) = \omega_{t/u}(\epsilon)$

Esta función nos permite obtener el símbolo de la signatura Σ que corresponde a la etiqueta que aparece en una ocurrencia determinada u del conjunto de ocurrencias de un término t. Extendemos de manera natural esta función obteniendo un homomorfismo ϖ_t respecto a la *concatenación de cadenas de ocurrencias* en $O(t)^*$ y a la *concatenación de cadenas de símbolos* en $(\Sigma \cup V)^*$:

$$\varpi_t : \; O(t)^* \rightarrow (\Sigma \cup V)^*$$

1. $\varpi_t(\epsilon) = \epsilon$
2. $\varpi_t(u.v) = \omega_t(u).\varpi_t(v)$ con $u \in O(t), v \in O(t)^*$

En lo que sigue, las definiciones dadas en términos de una signatura homogénea Σ y un conjunto de variables V se particularizan a la signatura $\Sigma_{\mathcal{M}\cup\mathcal{V}}$ y a los conjuntos de variables $\mathcal{V}$ y $\mathcal{W}$, es decir, al conjunto de términos $\mathcal{T}_{\Sigma_\mathcal{M}\cup\mathcal{V}}(\mathcal{W})$. Todas ellas tendrán gran importancia en la formalización del concepto de *túnel*.

Al hablar de *árbol sintáctico* de un término (o de un proceso) nos ceñimos al concepto expresado anteriormente. Sin embargo, al realizar la *representación gráfica* de dicho árbol, en lugar de presentar los nodos (ocurrencias del término) presentaremos su *determinación simbólica*. El árbol presentado es isomorfo al árbol sintáctico.

Dado $G = (N, A)$, un grafo donde N es un conjunto de nodos y A un conjunto de arcos $A \subseteq N \times N$, representamos un *camino* en el grafo G como una cadena $\gamma \in N^+$ de nodos del grafo, de manera que si $\gamma = n_1 n_2 \cdots n_k$ con $k > 0$, entonces $\forall i : \; 1 \leq i < k. \; \langle n_i, n_{i+1}\rangle \in A$. A los nodos n_1 y n_k se les denomina *extremos* del camino γ. Si, además, se verifica que $\forall i, j : \; 1 \leq i, j \leq k. \; i \neq j \implies n_i \neq n_j$, es decir, no se pasa dos veces por el mismo nodo, entonces el camino se denomina *camino elemental* [Ber83].

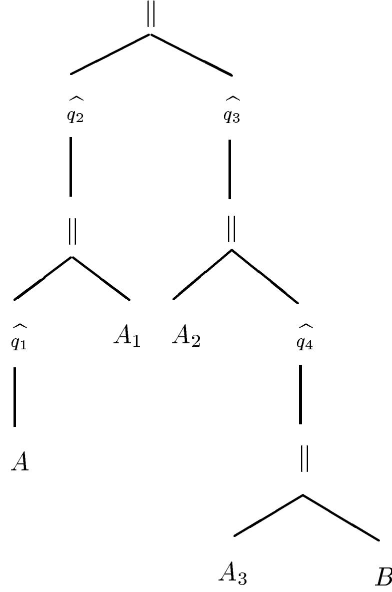


Figura 3.2: Arbol sintáctico del proceso $\widehat{\ }_{q_2}(\widehat{\ }_{q_1} A \parallel A_1) \parallel \widehat{\ }_{q_3}(A_2 \parallel \widehat{\ }_{q_4}(A_3 \parallel B))$

Definición 3.2.8 (Camino entre dos subprocesos) *Sea un proceso A. Sean M y N subprocesos disjuntos de A, esto es:*

$$\exists u, v \in O(A) : \; A/u = M, A/v = N \;, \; u \ntrianglelefteq v \;\; y \;\; v \ntrianglelefteq u$$

Entonces, un camino elemental $\gamma = n_1 n_2 \ldots n_k$ *con* $k > 0$ *en el árbol sintáctico simétrico* $\mathcal{A}^{\circ}(A)$ *de A define un camino entre los subprocesos M y N si:*

1. $\{u, v\} \cap \{n_1, n_2, \ldots n_k\} = \emptyset$*, es decir, u y v no están en el camino elemental.*

2. $\langle u, n_1 \rangle \in \mathcal{A}^{\circ}(A)$ *y* $\langle n_k, v \rangle \in \mathcal{A}^{\circ}(A)$.

Es decir, podemos completar el camino elemental con los nodos del árbol que representan las ocurrencias de los subprocesos en cuestión, resultando un nuevo camino elemental cuyos extremos son los nodos de interés.

Ejemplo 6 *Consideremos el proceso* $C \equiv \widehat{\ }_{q_2}(\widehat{\ }_{q_1} A \parallel A_1) \parallel \widehat{\ }_{q_3}(A_2 \parallel \widehat{\ }_{q_4}(A_3 \parallel B))$. *Como podemos apreciar en la Figura 3.2, el camino elemental entre los subprocesos A y B viene definido como:*

$$\gamma \;\; = \;\; < \widehat{\ }_{q_1}, \parallel, \widehat{\ }_{q_2}, \parallel, \widehat{\ }_{q_3}, \parallel, \widehat{\ }_{q_4}, \parallel >$$

3.3 Propiedades algebraicas

El símbolo $\equiv$ representa una congruencia sobre $\mathcal{M}$, que se define como la menor congruencia que satisface las siguientes propiedades [Mil93d]:

1. Dos procesos son equivalentes si sólo difieren en un cambio de variables ligadas (α-conversión del λ-cálculo).

2. Propiedades del monoide conmutativo $(\mathcal{M}/\equiv, \|, 0)$:

$\pi_{\|1}: M \| 0 \equiv 0 \| M \equiv M$	*Elemento neutro*
$\pi_{\|2}: M \| N \equiv N \| M$	*Conmutativa*
$\pi_{\|3}: M \| (N \| P) \equiv (M \| N) \| P$	*Asociativa*

3. Propiedades del monoide conmutativo $(\mathcal{M}/\equiv, +, 0)$:

$\pi_{+1}: M + 0 \equiv 0 + M \equiv M$	*Elemento neutro*
$\pi_{+2}: M + N \equiv N + M$	*Conmutativa*
$\pi_{+3}: M + (N + P) \equiv (M + N) + P$	*Asociativa*
$\pi_{+4}: M + M \equiv M$	*Idempotente*

4. Propiedades del operador de composición secuencial:

$\pi_{\circ 1}: M \circ 0 \equiv 0 \circ M \equiv M$	*Elemento neutro*

5. Propiedades del operador de replicación:

$\pi_{!1}: !M \equiv M \| !M$	*Replicación*

6. Propiedades del operador de emisión:

$\pi_{\widehat{\ }1}: \widehat{\ }_0 M \equiv M$	*Operador identidad*

3.4 Axiomas de reducción

Para ilustrar las definiciones que aparecen a continuación vamos a usar un proceso LCEP ejemplo, que describe un proceso compuesto formado por cuatro subprocesos en paralelo:

$$\lambda_P x.M \;\|\; (\widehat{_P}N \circ L + \widehat{_P}R \circ S) \;\|\; (\lambda_Q y.T + \lambda_P z.J) \;\|\; \widehat{_Q}U \circ V$$

Definición 3.4.1 (Proceso comunicante elemental) *Un proceso de la forma $\lambda_P x.M$ o bien $\widehat{_P}N \circ L$, con $P \in \mathcal{L}$, $x \in \mathcal{V}$, $L, M, N \in \mathcal{M}$ es un proceso comunicante elemental sobre el canal P. $\lambda_P x.M$ es un* receptor elemental *sobre P. $\widehat{_P}N \circ L$ es un* emisor elemental *sobre P.*

En el ejemplo, $\lambda_P x.M$ y $\lambda_P z.J$ son receptores elementales sobre P, $\lambda_Q y.T$ es un receptor elemental sobre Q, $\widehat{_P}N \circ L$ y $\widehat{_P}R \circ S$ son emisores elementales sobre P y $\widehat{_Q}U \circ V$ es un emisor elemental sobre Q.

Definición 3.4.2 (Par complementario) *Dado un canal $P \in \mathcal{L}$, si un proceso M es un receptor elemental sobre P y un proceso N es un emisor elemental sobre P, entonces (M, N) es un* par complementario.

En el ejemplo, $(\lambda_P x.M, \widehat{_P}N \circ L)$, $(\lambda_P x.M, \widehat{_P}R \circ S)$, $(\lambda_P z.J, \widehat{_P}N \circ L)$ y $(\lambda_P z.J, \widehat{_P}R \circ S)$ son pares complementarios a través del canal P y $(\lambda_Q y.T, \widehat{_Q}U \circ V)$ es un par complementario a través de Q.

Definición 3.4.3 (Proceso comunicante) *Un proceso M de la forma $M \equiv M_1 + M_2 + \ldots + M_n$, $n \geq 1$ donde alguno de los M_i es un proceso comunicante elemental, es un proceso comunicante. A los procesos comunicantes elementales M_i de un proceso comunicante $M \equiv M_1 + M_2 + \ldots + M_n$, con $1 \leq i \leq n$, se les llama* subprocesos comunicantes de M.

En el ejemplo, $\widehat{_P}N \circ L + \widehat{_P}R \circ S$ es un proceso comunicante y $\widehat{_P}N \circ L$ y $\widehat{_P}R \circ S$ son subprocesos comunicantes de él.

Definición 3.4.4 (Par comunicante) *Dados dos procesos comunicantes $M \equiv M_1 + M_2 + \ldots + M_m$ y $N \equiv N_1 + N_2 + \ldots + N_n$, el par (M, N) es un* par comunicante *si $\exists P \in \mathcal{L}$, $\exists i$ $1 \leq i \leq m$ y $\exists j$ $1 \leq j \leq n$ tales que (M_i, N_j) o (N_j, M_i) son un par complementario.*

En el ejemplo, $(\lambda_P x.M, \widehat{_P}N \circ L + \widehat{_P}R \circ S)$ es un par comunicante.

Definición 3.4.5 (Canales soporte de un proceso comunicante) *Sea un proceso comunicante $M \equiv M_1 + M_2 + \ldots + M_n$, $n \geq 1$. Entonces se definen los siguientes conjuntos:*

- $\Gamma^+(M) = \{P \in \mathcal{L} : \exists i\ 1 \leq i \leq n,\ \exists M'_i \in \mathcal{M},\ \exists x \in V.\ M_i \equiv \lambda_P x.M'_i\}$ *es el conjunto de canales de* entrada *en* M.

- $\Gamma^-(M) = \{P \in \mathcal{L} : \exists i\ 1 \leq i \leq n,\ \exists N_i, L_i \in \mathcal{M}.\ M_i \equiv \widehat{_P} N_i \circ L_i\}$ *es el conjunto de canales de* salida *en* M.

- $\Gamma(M) = \Gamma^+(M) \cup \Gamma^-(M)$ *es el conjunto de* canales soporte *de* M.

En el ejemplo, en $\lambda_Q y.T + \lambda_P z.J$, $\Gamma^+ = \{P, Q\}$ es el conjunto de canales de entrada, $\Gamma^- = \{\emptyset\}$ es el conjunto de canales de salida y $\Gamma = \{P, Q\}$ es el conjunto de canales soporte.

Definición 3.4.6 (Canales activos en un par comunicante) *En un par comunicante* (M, N) *pueden encontrarse varios pares complementarios. El conjunto de* canales activos *del par* (M, N), *denotado* $\Gamma(M, N)$, *es el conjunto de etiquetas* $P \in \mathcal{L}$ *para las cuales existe un par complementario* (M_i, N_j) *o* (N_j, M_i).

En el ejemplo, $\Gamma(\lambda_P x.M, \lambda_Q y.T + \lambda_P z.J) = \{P\}$.

De acuerdo con la definición de *canales de entrada y salida*, es evidente que:

$$\Gamma(M, N) = (\Gamma^+(M) \cap \Gamma^-(N)) \ \cup\ (\Gamma^+(N) \cap \Gamma^-(M))$$

Definimos dos axiomas de reducción (β-comunicación) que describen la evolución de un par comunicante cuando los procesos integrantes del par se componen secuencialmente o en paralelo: β_{sec} y β_P.

1. β_{sec}: β-comunicación secuencial:

$$(\ldots + \lambda_P x.M) \circ (\widehat{_P} N \circ L + \ldots) \ \longrightarrow_{\beta_{sec}}\ M[N/x] \circ L \qquad P \in \mathcal{L} - \{0\}$$

2. β_P: β-comunicación paralela, a través del canal P. Como más tarde veremos, al introducir el concepto de túnel (en la Sección 3.7), precisamos indexar la β-comunicación paralela, pues, dependiendo del orden de las etiquetas de los canales, será posible definir comunicaciones paralelas en diferentes niveles del árbol sintáctico. Por tanto, definimos β_P como sigue:

$$(\ldots + \lambda_P x.M) \parallel (\widehat{_P} N \circ L + \ldots) \ \longrightarrow_{\beta_P}\ M[N/x] \parallel L \qquad P \in \mathcal{L} - \{0\}$$

   donde β_P es una familia de axiomas indexada por $\mathcal{L} - \{0\}$.

3.5 Axiomas de reordenación

1. Reordenación con etiquetas simbólicas [AKG93b, AKG93c]:

$$\rho_1 \quad : \quad (\lambda_p x.\lambda_q y.M) \longrightarrow_{\rho_1} (\lambda_q y.\lambda_p x.M) \quad \text{si } p \succ_{\mathcal{L}} q$$

$$\rho_2 \quad : \quad \widehat{_p}M \circ \widehat{_q}N \longrightarrow_{\rho_2} \widehat{_q}N \circ \widehat{_p}M \quad \text{si } p \succ_{\mathcal{L}} q$$

$$\rho_{sec_3} \quad : \quad (\ldots + \lambda_p x.M) \circ (\widehat{_q}N \circ L + \ldots) \longrightarrow_{\rho_{sec_3}} \lambda_p x.(M \circ \widehat{_q}N) \circ L \quad \text{si } (p \succ_{\mathcal{L}} q) \vee (p \prec_{\mathcal{L}} q)$$

Los dos primeros axiomas indican que, dados dos procesos de input (output) etiquetados con canales simbólicos p y q, si se cumple que $p \succ_{\mathcal{L}} q$ entonces es posible intercambiar el orden de esos dos procesos de input (output). El tercero permite introducir un proceso dentro de una estructura más interna si se cumplen las condiciones de disparo del axioma, que son que $(p \succ_{\mathcal{L}} q)$ o $(p \prec_{\mathcal{L}} q)$.

2. Reordenación con etiquetas numéricas ($m, n \in \mathbb{N} - \{0\}$) [AKG93b, AKG93c]:

$$\eta_1 \quad : \quad (\lambda_m x.\lambda_n y.M) \longrightarrow_{\eta_1} (\lambda_n y.\lambda_{m-1} x.M) \quad \text{si } m \succ_{\mathcal{L}} n$$

$$\eta_2 \quad : \quad \widehat{_m}M \circ \widehat{_n}N \longrightarrow_{\eta_2} \widehat{_n}N \circ \widehat{_{m-1}}M \quad \text{si } m \succ_{\mathcal{L}} n$$

$$\eta_{sec_3} \quad : \quad (\ldots + \lambda_m x.M) \circ (\widehat{_n}N \circ L + \ldots) \longrightarrow_{\eta_{sec_3}} \lambda_{m-1} x.(M \circ \widehat{_n}N) \circ L \quad \text{si } m \succ_{\mathcal{L}} n$$

$$\eta_{sec_4} \quad : \quad (\ldots + \lambda_m x.M) \circ (\widehat{_n}N \circ L + \ldots) \longrightarrow_{\eta_{sec_4}} \lambda_m x.(M \circ \widehat{_{n-1}}N) \circ L \quad \text{si } m \prec_{\mathcal{L}} n$$

Estos cuatro axiomas están relacionados con la noción de currificación [Bar91]. Los dos primeros son análogos a los correspondientes a las etiquetas simbólicas y los dos últimos son similares al tercero. Estos axiomas nos permiten dar soporte formal a la instanciación etiquetada de parámetros para una función sin preocuparnos del orden de escritura de éstos (ver Subsección 3.1.2).

3.6 Reglas de inferencia

Las propiedades algebraicas y los axiomas de reducción anteriores constituyen los *axiomas* de una teoría formal que permite definir una relación de reducción $\rightarrow$ sobre los procesos en $\mathcal{M}$. Las reglas de inferencia de ese sistema formal son las siguientes:

$$\mu_{\circ_1}: \frac{M \rightarrow M'}{M \circ N \rightarrow M' \circ N} \qquad \mu_{\circ_2}: \frac{M \rightarrow M'}{N \circ M \rightarrow N \circ M'}$$

$$\mu_{\lambda}: \frac{M \rightarrow M'}{\lambda_p x.M \rightarrow \lambda_p x.M'} \qquad \mu_{\frown}: \frac{M \rightarrow M'}{\widehat{p}M \rightarrow \widehat{p}M'}$$

$$\mu_{\|}: \frac{M \rightarrow M'}{M \parallel N \rightarrow M' \parallel N} \qquad \mu_{Struct}: \frac{M \equiv N \quad N \rightarrow N' \quad N' \equiv M'}{M \rightarrow M'}$$

Estas reglas de inferencia recogen el significado computacional de los operadores del lenguaje LCEP y son análogas a las reglas de inferencia del π-cálculo [Mil93d].

Las reglas permiten la reducción de determinados λ_{EP}-subtérminos en función de la estructura del λ_{EP}-término donde se encuentran. Las reglas de inferencia gobiernan este proceso de reducción de (sub)términos.

Observamos la ausencia de reglas de este tipo para los operadores de replicación (!) y de elección indeterminista (+). En el primer caso, esta decisión refleja el papel computacional asignado al operador: construir una especie de "almacén" del cual podemos tomar, de modo indefinido, un proceso; por tanto, no parece tener mucho sentido que el proceso que guardamos ahí cambie.

En cuanto al operador de elección indeterminista, su significado computacional viene descrito en términos de los axiomas de reducción y reordenación en el contexto de los procesos comunicantes. Básicamente, recoge la idea de que una comunicación puede concretarse en cualquiera de los procesos comunicantes elementales integrantes del proceso comunicante, haciendo entonces que los restantes procesos desaparezcan. Por lo tanto, no tiene sentido que los integrantes (sumandos) de un proceso comunicante desarrollen actividad computacional aparte de la derivada de la aplicación de las reglas mencionadas, dado que, de entre ellos, solamente uno de los procesos experimentará una reducción y continuará la ejecución con posteriores operaciones de reducción.

La regla μ_{Struct} permite dar significado computacional a las clases de congruencia definidas en $\mathcal{M}$ por la congruencia $\equiv$ (ver la Sección 3.3). Su presencia entre las reglas de inferencia equivale a considerar los axiomas de reducción y reordenación definidos sobre el conjunto cociente $\mathcal{M}/\equiv$. La presencia de esta regla, junto con las propiedades algebraicas del operador $\|$ (ver la Sección 3.3), permiten prescindir de una regla análoga a la regla $\mu_{\circ 2}$ para el operador $\|$, como mostramos seguidamente:

(1) $N \to N'$	*por hipótesis*
(2) $N \parallel M \;\equiv\; M \parallel N$	*por* $\pi_{\parallel 2}$
(3) $N \parallel M \to N' \parallel M$	*por* $\mu_{\parallel}$ *sobre* (1)
(4) $N' \parallel M \;\equiv\; M \parallel N'$	*por* $\pi_{\parallel 2}$
(5) $M \parallel N \to M \parallel N'$	*por* μ_{Struct} *sobre* (2), (3), (4)

Es decir, las reglas anteriores y las propiedades algebraicas de los operadores establecen que:

$$\frac{N \to N'}{M \parallel N \to M \parallel N'}$$

3.7 El concepto de túnel

En el mundo real, la idea de comunicación se encuentra ligada de alguna manera al deseo de reducir distancias entre personas, entidades, ... Los modelos computacionales basados en la noción de *proceso* emplean la comunicación como abstracción de la interacción que existe entre los procesos presentes en el sistema. El concepto de *canal* de comunicación sirve para representar la distancia que media entre los participantes en una comunicación, ya que el canal significa, para cada participante en el enlace, su interlocutor respectivo.

Los modelos formales de comunicación entre procesos incorporan el canal como base de la representación del mecanismo de comunicación. En el π-cálculo [Mil93d], por ejemplo, la acción básica que dos (sub)procesos pueden realizar es la comunicación, representada por el axioma de comunicación, como ya hemos visto. Dicha comunicación se puede llevar a cabo cuando dos (sub)procesos *normales* [Mil93d] con ocurrencias complementarias de un *nombre* se encuentran relacionados mediante el operador de paralelismo ($|$).

La regla análoga en LCEP es la β_P-comunicación paralela, que permite ésta entre dos procesos comunicantes enlazados por el constructor de paralelismo ($\parallel$). Se puede interpretar que el constructor de paralelismo representa la *distancia (sintáctica)* que los procesos deben salvar para establecer la comunicación. Ello significa entender la estructura sintáctica del proceso como el medio "físico" a través del cual pueden o no establecerse comunicaciones entre los subprocesos existentes en él. Bajo esta óptica, la regla β_P describe cómo debe ser el camino que se establece entre un proceso emisor y uno receptor para que entre los subprocesos ocurra una comunicación efectiva.

Por ejemplo, en el proceso $A \equiv \lambda_P x.M \parallel \widehat{P} N$ se puede dar una β_P-comunicación.

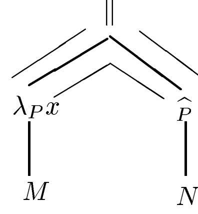


Figura 3.3: β_P-comunicación paralela

Esta afirmación es inmediata, ya que:

$$\lambda_P x.M \parallel \widehat{_P}N \;\equiv\; \lambda_P x.M \parallel \widehat{_P}N \circ 0 \;\equiv\; (0+\lambda_P x.M) \parallel (\widehat{_P}N \circ 0 + 0)$$

$$(0+\lambda_P x.M) \parallel (\widehat{_P}N \circ 0 + 0) \;\longrightarrow_{\beta_P}\; M[N/x] \parallel 0$$

En la Figura 3.3 se ha encerrado el *camino sintáctico* que se debe "recorrer" para establecer la comunicación entre los procesos. Observamos que, por ejemplo, en un proceso como $B \equiv \lambda_P x.M \parallel y \circ \widehat{_P}N$, no sería posible realizar una β_P-comunicación. La regla establece las condiciones mínimas para que dicha comunicación pueda darse. Define también, bajo la interpretación métrica sintáctica esbozada, la *distancia mínima* que puede recorrerse mediante una β_P-comunicación.

El concepto de *túnel* permite extender el alcance de una β_P-comunicación. La siguiente regla describe cómo se consigue:

$$\mu_{\mathcal{T}} \quad : \quad \frac{M \parallel N \;\longrightarrow_{\beta_P}\; M' \parallel N'}{\widehat{_{Q_1}}M \parallel \widehat{_{Q_2}}N \;\longrightarrow_{\beta_P}\; \widehat{_{Q_1}}M' \parallel \widehat{_{Q_2}}N'}$$

para $Q_1, Q_2 \in \mathcal{L}$ si $P \succ_{\mathcal{L}} Q_1, Q_2$.

Informalmente, $\mu_{\mathcal{T}}$ expresa la posibilidad de establecer una β_P-comunicación entre dos procesos $\widehat{_{Q_1}}M$ y $\widehat{_{Q_2}}N$, sujeta a una doble condición:

- M y N deben ser capaces de establecer una β_P-comunicación.
- La etiqueta $P \in \mathcal{L}$ debe ser *mayor* que las etiquetas Q_1 y Q_2.

Podemos distinguir dos cuestiones en relación con la regla $\mu_{\mathcal{T}}$:

1. Por un lado, la regla expresa una *condición sintáctica* sobre la forma de los procesos que pueden establecer una β_P-comunicación *extendida*. Dichos procesos se pueden crear aplicando operadores de output $\widehat{_{Q_1}}$ y $\widehat{_{Q_2}}$ a procesos que sabemos capaces de comunicarse, y, a continuación, aplicando el operador de construcción de paralelismo.

2. Por otro lado, expresa una *restricción* sobre los operadores $\widehat{_{Q_1}}$ y $\widehat{_{Q_2}}$, que podemos aplicar para conservar la β_P-comunicación establecida. Dicha restricción se da

en términos de la relación de orden existente entre P, el canal a través del cual se establece la comunicación primitiva, y las etiquetas Q_1 y Q_2 asociadas a los constructores de proceso empleados.

Por tanto, independientemente de la posibilidad real de una β_P-comunicación debida a la relación de orden parcial establecida en $\mathcal{L}$, podemos hablar de la existencia de ciertas estructuras sintácticas entre dos procesos comunicantes que hacen "físicamente" posible esa comunicación. Esta estructura sintáctica es lo que entendemos por *túnel*, y puede describirse como *un camino en el árbol sintáctico de un proceso, entre dos subprocesos comunicantes.*

Ejemplo 7 *Consideremos el proceso C del Ejemplo 6, $C \equiv \widehat{_{q_2}}(\widehat{_{q_1}}A \parallel A_1) \parallel \widehat{_{q_3}}(A_2 \parallel \widehat{_{q_4}}(A_3 \parallel B))$. Vamos a comprobar que si $A \equiv \lambda_p x.M$ y $B \equiv \widehat{_p}N$ entonces $A \parallel B \longrightarrow_{\beta_p} A' \parallel B'$ y puede producirse una β_p comunicación entre A y B, en el seno de C. Suponemos que $p \succ q_i$ para $i = 1..4$.*

$A \parallel B \longrightarrow_{\beta_p} A' \parallel B'$	*por la estructura de A y B*
$A \parallel A_3 \parallel B \longrightarrow_{\beta_p} A' \parallel A_3 \parallel B'$	*por* $\mu_{\parallel}$
$\widehat{_{q_1}}A \parallel \widehat{_{q_4}}(A_3 \parallel B) \longrightarrow_{\beta_p} \widehat{_{q_1}}A' \parallel \widehat{_{q_4}}(A_3 \parallel B')$	*por* $\mu_{\mathcal{T}}$
$\widehat{_{q_1}}A \parallel A_1 \parallel A_2 \parallel \widehat{_{q_4}}(A_3 \parallel B) \longrightarrow_{\beta_p}$	
$\longrightarrow_{\beta_p} \widehat{_{q_1}}A' \parallel A_1 \parallel A_2 \parallel \widehat{_{q_4}}(A_3 \parallel B')$	*por* $\mu_{\parallel}$
$\widehat{_{q_2}}(\widehat{_{q_1}}A \parallel A_1) \parallel \widehat{_{q_3}}(A_2 \parallel \widehat{_{q_4}}(A_3 \parallel B)) \longrightarrow_{\beta_p}$	
$\longrightarrow_{\beta_p} \widehat{_{q_2}}(\widehat{_{q_1}}A' \parallel A_1) \parallel \widehat{_{q_3}}(A_2 \parallel \widehat{_{q_4}}(A_3 \parallel B'))$	*por* $\mu_{\mathcal{T}}$

En el árbol sintáctico del proceso C podemos identificar el túnel que aparece dibujado en la Figura 3.4.

Observamos que, independientemente de qué procesos sean A y B, el túnel existe como entidad independiente. En el árbol sintáctico del proceso C, el túnel existe como *un camino entre los nodos A y B*. No obstante, no es un túnel cualquier camino entre dos nodos del árbol sintáctico de un proceso. Un túnel es un camino constituido por constructores de output y constructores de paralelismo.

Formalmente, la definición de túnel es la siguiente:

Definición 3.7.1 (Túnel y s-túnel) *Sea $M \in \mathcal{M}$ un proceso. Un camino elemental $\gamma_\tau \in O(M)^+$ en el árbol sintáctico simétrico $\mathcal{A}^\circ(M)$ del proceso M, cuya* determinación simbólica $\tau = \varpi_M(\gamma_\tau)$ *es de la forma $\tau = \beta_1 \parallel \beta_2$, donde $\beta_1, \beta_2 \in (\mathcal{P} \cup \{\parallel\})^*$, es un* túnel *en el árbol sintáctico de M. A la determinación simbólica de γ_τ, τ, se le llama* s-túnel.

Definición 3.7.2 (Localización de un túnel) *La* localización *de un túnel γ_τ es la menor ocurrencia ν_{γ_τ} que participa en el camino elemental γ_τ.*

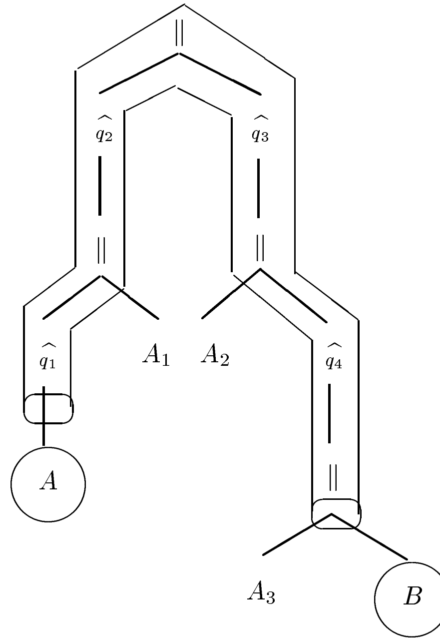


Figura 3.4: Identificación del túnel en el árbol

Muchas propiedades pueden definirse únicamente en función del s-túnel asociado a un túnel. Es importante observar que, puesto que la función ϖ_t no es en general inyectiva, un mismo s-túnel puede asociarse con varios túneles, si éstos ocupan localizaciones diferentes en el árbol sintáctico del proceso.

El significado de las definiciones de túnel y s-túnel se pondrá de manifiesto en el Teorema 3.7.1.

Definición 3.7.3 ($\mathcal{T}$-comunicación potencial) *Sea un proceso A. Si M y N son subprocesos de A, existe una $\mathcal{T}$-comunicación potencial entre M y N si:*

1. *M y N son procesos comunicantes,*

2. *existe un túnel entre M y N.*

Sin embargo, como ya se apuntó, la presencia de un túnel y de dos procesos comunicantes M y N a sus extremos no es suficiente para que se de una comunicación. Se puede establecer una comunicación entre dichos procesos en función de la *composición* del túnel, la relación de orden establecida sobre $\mathcal{L}$ y el conjunto de canales activos $\Gamma(M, N)$ entre M y N.

Definición 3.7.4 (Conjunto de etiquetas de un túnel) *Sea un túnel γ_τ y el correspondiente s-túnel τ. Definimos un homomorfismo $h : (\mathcal{P} \cup \{\|\})^* \to 2^{\mathcal{L}}$ respecto*

a la concatenación de símbolos en $(\mathcal{P} \cup \{\|\})^*$ *y respecto a la unión de conjuntos en* $2^{\mathcal{L}}$*, como sigue:*

$$h(\|) = h(\epsilon) = \emptyset$$

$$h(\widehat{P}) = \{P\}$$

de manera que $h(a\beta) = h(a) \cup h(\beta)$*, con* $a \in \mathcal{P} \cup \{\|\}$*,* $\beta \in (\mathcal{P} \cup \{\|\})^*$*.*

Entonces, el conjunto de etiquetas de un túnel γ_τ *(y del s-túnel* τ*) es* $h(\tau)$*.*

Definición 3.7.5 (Túnel P-transparente) *Un túnel* γ_τ *(o su s-túnel* τ*) es* P*-transparente si* $P \in \mathcal{L}$ *es una cota superior* estricta *de* $h(\tau)$*, es decir, si* $\forall Q \in h(\tau)\ P \succ_{\mathcal{L}} Q$*, o* $h(\tau) = \emptyset$*.*

Haciendo uso del concepto de transparencia de un túnel, vamos a formalizar las propiedades que se deben cumplir para que sea posible una comunicación sobre un canal determinado.

Definición 3.7.6 ($\mathcal{T}$-comunicabilidad sobre un canal) *Sea un proceso* A*. Si* M *y* N *son subprocesos de* A*, entonces* M *y* N *son* $\mathcal{T}$*-comunicables sobre un canal* P *si:*

1. (M, N) *es un par comunicante,*
2. $P \in \Gamma(M, N)$*,*
3. *existe un túnel* P*-transparente* γ_τ *entre* M *y* N *localizado en* ν_{γ_τ}*, y*
4. $\forall w \in O(A):\ w \trianglelefteq \nu_{\gamma_\tau} \ \Rightarrow\ \omega_A(w) \in \{\|, \circ\} \cup \Lambda \cup \mathcal{P}$*.*

De las definiciones anteriores se desprende que la $\mathcal{T}$-comunicabilidad generaliza la idea de la β-comunicación paralela, siendo ésta un caso particular de $\mathcal{T}$-comunicabilidad en la que el s-túnel entre los procesos comunicantes es el más sencillo posible: $\tau = \|$. De la definición de transparencia de un túnel se sigue que para todo $P \in \mathcal{L}$, un s-túnel como τ siempre es P-transparente, ya que siempre es factible la comunicación. Esto viene establecido por el siguiente teorema:

Teorema 3.7.1 ($\mathcal{T}$-comunicabilidad) *Sea un proceso* A*. Sean* M *y* N *dos subprocesos de* A *de manera que* $M = A/u$ *y* $N = A/v$ *para* $u, v \in O(A)$*.*

Si M *y* N *son* $\mathcal{T}$*-comunicables sobre un canal* $P \in \mathcal{L}$*, siendo* M' *y* N' *subprocesos comunicantes de* M *y* N*, respectivamente, y* (M', N') *un par complementario, entonces, asumiendo que* $M' \equiv \lambda_P x.M''$ *y* $N' \equiv \widehat{P} N'' \circ L$*, se cumple que:*

$$A \ \rightarrow\ A' \quad \text{donde } A' = A[u \leftarrow M''[N''/x]][v \leftarrow L]$$

DEMOSTRACIÓN.

Por hipótesis, M y N son dos subprocesos de A que son $\mathcal{T}$-comunicables sobre un canal $P \in \mathcal{L}$. Por la Definición 3.7.6, (M, N) es un par comunicante. Por ser (M, N) un par comunicante, por la Definición 3.4.4, M y N son de la forma $M \equiv M_1 + M_2 + \ldots + M_m$ y $N \equiv N_1 + N_2 + \ldots + N_n$, en donde, al ser (M, N) un par comunicante, $\exists P \in \mathcal{L}$, $\exists i\ 1 \leq i \leq m$ y $\exists j\ 1 \leq j \leq n$ tales que (M_i, N_j) o (N_j, M_i) son un par complementario. En este caso, un par complementario es (M', N'). Por hipótesis, $M' \equiv \lambda_P x.M''$ y $N' \equiv \widehat{_P} N'' \circ L$. Por tanto, se dan las condiciones necesarias para que se pueda producir una β-comunicación, o bien secuencial o bien paralela, en función de cómo estén compuestos entre sí los subprocesos:

$$\begin{array}{l}(\ldots + \lambda_P x.M'') \circ (\widehat{_P} N'' \circ L + \ldots) \ \longrightarrow_{\beta_{sec}} \ M''[N''/x] \circ L \\ \qquad P \in \mathcal{L} - \{0\}\end{array}$$

o

$$\begin{array}{l}(\ldots + \lambda_P x.M'') \parallel (\widehat{_P} N'' \circ L + \ldots) \ \longrightarrow_{\beta_P} \ M''[N''/x] \parallel L \\ \qquad P \in \mathcal{L} - \{0\}\end{array}$$

En ambos casos, M se transforma en $M''[N''/x]$. Por tanto, el proceso A se reduce al proceso A', con $A' = A[u \leftarrow M''[N''/x]][v \leftarrow L]$.

□

La aplicación de un operador $\widehat{_P}$ a un proceso A provoca el *aislamiento* de los procesos comunicantes M en A respecto a las comunicaciones *externas* a $\widehat{_P} A$, a través de los canales $Q \in \Gamma(M)$ que no sean estrictamente superiores a P, como podemos comprobar al intentar aplicar la regla del túnel. Esta observación motiva la siguiente definición:

Definición 3.7.7 (P-filtro de un proceso) *Sea A un proceso y $P \in \mathcal{L}$. El proceso $\widehat{_P} A$ se denomina un P-*filtro *para el proceso A.*

Si M es un subproceso comunicante en A, entonces, para que un proceso N *externo* a $\widehat{_P} A$ sea $\mathcal{T}$-comunicable sobre cualquier $Q \in \Gamma(M, N)$, debe cumplirse como condición necesaria, aunque no suficiente, que P sea una *cota inferior estricta* de $\Gamma(M, N)$. La noción de *filtro* presenta características similares a las del operador de restricción ν del π-cálculo de Milner [Mil93d], ya que restringe la capacidad de comunicación con el exterior del filtro. Sin embargo, en el π-cálculo, ν restringe el ámbito de uso de los nombres (su localización) y esa idea no se traslada al P-filtro, puesto que en LCEP hay dos conjuntos disjuntos: el de las etiquetas y el de los procesos, mientras que en el π-cálculo los nombres se usan tanto para canales como para generar los procesos.

Ejemplo 8 *Consideremos el proceso $C \equiv \widehat{_{q_2}}(\widehat{_{q_1}} A \parallel A_1) \ \parallel \ \widehat{_{q_3}}(A_2 \parallel \widehat{_{q_4}}(A_3 \parallel B))$ de los ejemplos 6 y 7. Sean los procesos $A \equiv \lambda_p x.M$, $A_2 \equiv \lambda_p x.M'$ y $B \equiv \widehat{_p} N$. El subproceso*

$\widehat{q_3}(A_2 \parallel \widehat{q_4}(A_3 \parallel B))$ *es un* q_3*-filtro para* $A_2 \parallel \widehat{q_4}(A_3 \parallel B)$. *Si* $p \succ q_i$ *para* $i = 1, 2, 4$ *pero* $p \preceq q_3$ *o simplemente* p *y* q_3 no son comparables, *entonces el túnel entre* A *y* B no es p-transparente, *y* A *y* B *no son* $\mathcal{T}$*-comunicables. Sin embargo, existe un túnel* p*-transparente entre* A_2 *y* B*, pudiéndose establecer una* $\mathcal{T}$*-comunicación entre ellos. Esto significa que las comunicaciones sobre* p *quedan* confinadas *al interior del* q_3*-filtro.*

A continuación vamos a recuperar los ejemplos desarrollados al final del Capítulo 2 utilizando en esta ocasión LCEP como lenguaje de programación.

3.8 Ejemplos

Revisamos los problemas resueltos en el Capítulo 2, adaptándolos a la sintaxis de LCEP.

3.8.1 Ejemplo 1: El problema de los Filósofos

1. La especificación LCEP de este problema es:

$$\text{PHIL}_i \triangleq \lambda_{pos_i} x \circ \text{PHIL'}_i$$

$$\begin{aligned}\text{PHIL'}_i \triangleq\ & \lambda_{chop_i} y \circ \lambda_{chop_{i\underline{+}1}} z \circ \text{PHIL''}_i + \\ & + \lambda_{chop_{i\underline{+}1}} y \circ \lambda_{chop_i} z \circ \text{PHIL''}_i\end{aligned}$$

$$\begin{aligned}\text{PHIL''}_i \triangleq\ & \widehat{chop_i} \circ \widehat{chop_{i\underline{+}1}} \circ \widehat{pos_i} \circ \text{PHIL}_i + \\ & + \widehat{chop_{i\underline{+}1}} \circ \widehat{chop_i} \circ \widehat{pos_i} \circ \text{PHIL}_i\end{aligned}$$

$$\text{CHOP}_i \triangleq \widehat{chop_i} \circ \lambda_{chop_i} x \circ \text{CHOP}_i$$

La definición de `MAID`[1] necesita:

$$\begin{aligned}\text{MAID}_0 \triangleq\ & \widehat{pos_0} \circ \text{MAID}_1 + \widehat{pos_1} \circ \text{MAID}_1 + \\ & + \cdots + \widehat{pos_{n-1}} \circ \text{MAID}_1\end{aligned}$$

$$\begin{aligned}\text{MAID}_{1 \le i \le n-2} \triangleq\ & \lambda_{pos_0} x \circ \text{MAID}_{i-1} + \lambda_{pos_1} x \circ \text{MAID}_{i-1} + \\ & + \cdots + \lambda_{pos_{n-1}} x \circ \text{MAID}_{i-1} + \\ & + \widehat{pos_0} \circ \text{MAID}_{i+1} + \widehat{pos_1} \circ \text{MAID}_{i+1} + \\ & + \cdots + \widehat{pos_{n-1}} \circ \text{MAID}_{i+1}\end{aligned}$$

[1]El significado del subíndice de `MAID` es el número de filósofos que están comiendo en estos momentos. El símbolo + se interpreta como la suma de la forma habitual.

$$\begin{aligned}\text{MAID}_{n-1} &\triangleq \lambda_{pos_0}x \circ \text{MAID}_{n-2} + \lambda_{pos_1}x \circ \text{MAID}_{n-2} + \\ &\cdots + \lambda_{pos_{n-1}}x \circ \text{MAID}_{n-2}\end{aligned}$$

$$\begin{aligned}\text{COLLEGE} &\triangleq \text{PHIL}_0 \parallel \cdots \parallel \text{PHIL}_{n-1} \parallel \\ &\parallel \text{CHOP}_0 \parallel \cdots \parallel \text{CHOP}_{n-1} \parallel \text{MAID}_0\end{aligned}$$

Podemos ver que la modelización del problema haciendo uso de la CHAM, CHOCS, el π-cálculo y el λ-cálculo Etiquetado Paralelo es similar. Las diferencias son mínimas porque la solución a este problema no precisa enviar ninguna información a través de los canales, es solamente un problema de sincronización de procesos y, en cualquiera de ellos, entonces, las construcciones usadas son análogas.

2. Primeros pasos de la ejecución:

   Vamos a plantearnos la ejecución del proceso construido. Vemos que se parte de un proceso compuesto constituido por diferentes subprocesos en paralelo:

$$\begin{aligned}\text{COLLEGE} &\triangleq \text{PHIL}_0 \parallel \cdots \parallel \text{PHIL}_{n-1} \parallel \\ &\parallel \text{CHOP}_0 \parallel \cdots \parallel \text{CHOP}_{n-1} \parallel \text{MAID}_0\end{aligned}$$

   Observando la definición de cada uno de esos subprocesos tenemos que, al comenzar, son posibles varias comunicaciones: entre $\texttt{MAID}_0$ y cualesquiera de los procesos asociados a los filósofos, y el significado informal de la comunicación seleccionada será que el filósofo elegido en la comunicación se sentará en la mesa. Imaginemos que es el filósofo $\texttt{PHIL}_0$. Entonces, el sistema evoluciona a:

$$\begin{aligned}&\text{PHIL'}_0 \parallel \text{PHIL}_1 \parallel \cdots \parallel \text{PHIL}_{n-1} \parallel \\ &\parallel \text{CHOP}_0 \parallel \cdots \parallel \text{CHOP}_{n-1} \parallel \text{MAID}_1\end{aligned}$$

   En estos momentos son posibles diversas comunicaciones: cualquier filósofo diferente a $\texttt{PHIL}_0$ se puede sentar a la mesa, o bien el filósofo $\texttt{PHIL}_0$ puede tomar al azar uno de los dos palillos que se encuentran a su lado.

   Supongamos que la comunicación que se selecciona permite que el filósofo $\texttt{PHIL}_0$ tome el palillo $\texttt{CHOP}_1$. Entonces tenemos:

$$\begin{aligned}&\lambda_{chop_0}z \circ \text{PHIL"}_0 \parallel \text{PHIL}_1 \parallel \cdots \parallel \text{PHIL}_{n-1} \parallel \text{CHOP}_0 \parallel \\ &\parallel \lambda_{chop_1}x \circ \text{CHOP}_1 \parallel \cdots \parallel \text{CHOP}_{n-1} \parallel \text{MAID}_1\end{aligned}$$

   De nuevo, son posibles varias comunicaciones: que el filósofo $\texttt{PHIL}_0$ tome el palillo $\texttt{CHOP}_0$, que deje en la mesa el palillo $\texttt{CHOP}_1$ o que se siente en la mesa

cualquiera del resto de filósofos. Supongamos que la comunicación que se da permite que se siente en la mesa el filósofo $\texttt{PHIL}_1$. La situación del sistema pasa a ser:

$$\lambda_{chop_0} z \ \circ \ \mathrm{PHIL"}_0 \parallel \mathrm{PHIL'}_1 \parallel \mathrm{PHIL}_2 \parallel \cdots \parallel \mathrm{PHIL}_{n-1} \parallel$$
$$\parallel \mathrm{CHOP}_0 \parallel \lambda_{chop_1} x \ \circ \ \mathrm{CHOP}_1 \parallel \cdots \parallel \mathrm{CHOP}_{n-1} \parallel \mathrm{MAID}_2$$

Y el sistema puede, de una forma similar a la vista anteriormente, continuar su evolución indefinidamente.

3.8.2 Ejemplo 2: Teléfonos móviles

1. Utilizando la versión simplificada del problema introducida en el Capítulo 2, el programa LCEP que lo resuelve es:

$$\mathrm{SYSTEM} \ \triangleq \ \mathrm{CAR}(t_1, s_1) \parallel \mathrm{BASE}_1$$

Vamos a usar canales numéricos como mecanismo para construir las tuplas de los canales y aplicaremos los axiomas de reordenación definidos anteriormente. Esta solución que planteamos ilustra la importancia de poder disponer de etiquetas numéricas como una particularidad a destacar del λ-cálculo etiquetado paralelo.

La solución general es, por tanto:

$$\mathrm{CAR}(t,s) \ \triangleq \ \lambda_t x.\mathrm{CAR}(t,s) \ + \ \lambda_s x.\lambda_1 T.\lambda_1 S.\mathrm{CAR}(T,S)$$

$$\begin{aligned}\mathrm{BASE}_i \ \triangleq \ & \widehat{\ }_{t_i}\ t \circ \mathrm{BASE}_i \ + \\ & + \widehat{\ }_{s_i}\ s \circ (\widehat{\ }_1\ t_{i+1} \circ \widehat{\ }_1\ s_{i+1} \circ \mathrm{BASE}_{i+1})\end{aligned}$$

Entonces:

$$\begin{aligned}\mathrm{CAR}(t_1,s_1) \ \triangleq \ & \lambda_{t_1} x.\mathrm{CAR}(t_1,s_1) \ + \\ & + \ \lambda_{s_1} x.\lambda_1 T.\lambda_1 S.\mathrm{CAR}(T,S)\end{aligned}$$

$$\begin{aligned}\mathrm{CAR}(t_2,s_2) \ \triangleq \ & \lambda_{t_2} x.\mathrm{CAR}(t_2,s_2) \ + \\ & + \ \lambda_{s_2} x.\lambda_1 T.\lambda_1 S.\mathrm{CAR}(T,S)\end{aligned}$$

$$\mathrm{BASE}_1 \ \triangleq \ \widehat{\ }_{t_1}\ t_1 \circ \mathrm{BASE}_1 \ + \ \widehat{\ }_{s_1}\ s_1 \circ (\widehat{\ }_1\ t_2 \circ \widehat{\ }_1\ s_2 \circ \mathrm{BASE}_2)$$
$$\mathrm{BASE}_2 \ \triangleq \ \widehat{\ }_{t_2}\ t_2 \circ \mathrm{BASE}_2 \ + \ \widehat{\ }_{s_2}\ s_2 \circ (\widehat{\ }_1\ t_1 \circ \widehat{\ }_1\ s_1 \circ \mathrm{BASE}_1)$$

2. Primeros pasos de la ejecución:

   El sistema está formado por dos procesos compuestos en paralelo. El comienzo de la ejecución presenta dos posibles comunicaciones paralelas, a través de los canales t_1 y s_1. Supongamos que se da la comunicación por t_1. El sistema, entonces, evoluciona a:

   $$\mathrm{CAR}(t_1, s_1) \parallel \mathrm{BASE}_1$$

   Esta posibilidad puede repetirse indefinidamente. Si el sistema selecciona el canal s_1, la evolución es a:

   $$\lambda_1 T.\lambda_1 S.\mathrm{CAR}(T,S) \parallel \widehat{1}\ t_2 \circ \widehat{1}\ s_2 \circ \mathrm{BASE}_2$$

   Ahora, la única posible comunicación es a través del canal `1`:

   $$\lambda_1 S.\mathrm{CAR}(t_2, S) \parallel \widehat{1}\ s_2 \circ \mathrm{BASE}_2$$

   De nuevo, la única comunicación posible es a través del canal `1`:

   $$\mathrm{CAR}(t_2, s_2) \parallel \mathrm{BASE}_2$$

   Y podemos apreciar que el sistema ha evolucionado cambiando los procesos de partida $\mathtt{CAR}(t_1, s_1)$ y $\mathtt{BASE}_1$ a $\mathtt{CAR}(t_2, s_2)$ y $\mathtt{BASE}_2$, como se pretendía.

En el Capítulo 5, por último, veremos la solución a estos dos mismos problemas utilizando el lenguaje de más alto nivel ALEPH.

3.9 Comparación con el π-cálculo monádico

Para finalizar el presente capítulo, vamos a construir una función de traducción de las estructuras básicas del π-cálculo a sus correspondientes en LCEP, y viceversa. Para desarrollar esta función, debemos imponer diferentes restricciones sobre la sintaxis de LCEP. Esta función demuestra que el π-cálculo puede interpretarse como un subconjunto de LCEP ya que todas sus estructuras son representables en LCEP sin utilizar la expresividad que introduce en el cálculo el uso de las etiquetas numéricas como canales de comunicación.

Existe una línea de investigación que trata formalmente la comparación entre diferentes lenguajes haciendo uso del "*embedding*" [dBP91a, dBP91b]. Nuestro objetivo aquí es, simplemente, mostrar las semejanzas y diferencias entre ambos cálculos y, mediante la función de traducción, trasladar las construcciones de un cálculo al otro. Sin embargo, como trabajo futuro, ésta es una idea muy interesante a desarrollar.

Vamos a comenzar por describir las diferencias que existen entre ambos cálculos y que exigen limitaciones a la capacidad expresiva de LCEP en la traducción:

- Hacemos que en LCEP el conjunto de etiquetas y el de variables se correspondan (en definitiva, son conjuntos de símbolos). Evidentemente, el π-cálculo es más simple en su definición, pero la posibilidad de distinguir entre las etiquetas y la información a trasmitir por ellas permite acercar más nuestra propuesta al λ-cálculo, que ha sido nuestro punto de partida.

- El conjunto de etiquetas numéricas puede considerarse con el mismo tratamiento que el de las simbólicas, es decir, se pueden considerar como el mismo conjunto de símbolos, con el orden lexicográfico (la noción de la currificación se pierde en el π-cálculo, como ya sabíamos, puesto que el cálculo es incapaz de expresarla).

- Los canales numéricos de LCEP no tienen una correspondencia directa en el π-cálculo, como acabamos de citar en el punto anterior.

- En el π-cálculo no existen las reglas de reordenación, heredadas en LCEP de la propuesta de Aït-Kaci en el λ-cálculo Etiquetado.

- Las reglas de inferencia $\mu_{\circ_2}, \mu_\lambda, \mu_\frown$ no se pueden utilizar.

- El operador de restricción (ámbito de las variables) presenta ciertas similitudes con la noción de *filtro* en LCEP (ver la Definición 3.7.7).

- El operador de secuenciación de LCEP se puede corresponder con el "punto" del π-cálculo.

Asumiendo todas estas simplificaciones en LCEP, todas las estructuras básicas del π-cálculo monádico se pueden definir en este subconjunto de LCEP. Veamos, por tanto, la función de traducción del π-cálculo de R. Milner a nuestra propuesta LCEP, definida como:

$$[\![\cdot]\!] \;:\; \pi\text{-cálculo} \;\rightarrow\; LCEP$$

- El proceso nulo existe en ambas propuestas:

$$[\![0]\!] \;=_{[\![\cdot]\!]}\; 0$$

- Una variable en el π-cálculo se corresponde con una variable en LCEP:

$$[\![x]\!] \;=_{[\![\cdot]\!]}\; x$$

- El prefijo de entrada en el π-cálculo se traduce a una abstracción etiquetada con el mismo canal en LCEP:

$$[\![p(x).M]\!] \;=_{[\![\cdot]\!]}\; \lambda_p x.[\![M]\!]$$

- El prefijo de salida del π-cálculo se traduce a un aplicador cuyo canal de comunicación coincide y cuyo proceso asociado es la información trasmitida por el canal de salida en el π-cálculo, todo ello seguido secuencialmente del resto del proceso:

$$[\![\overline{p}x.M]\!] =_{[\![\cdot]\!]} \widehat{_p}x \circ [\![M]\!]$$

- El operador de paralelismo del π-cálculo tiene su operador correspondiente en LCEP:

$$[\![M \mid N]\!] =_{[\![\cdot]\!]} [\![M]\!] \parallel [\![N]\!]$$

- En ambos cálculos existe un operador de elección indeterminista con un significado similar:

$$[\![M + N]\!] =_{[\![\cdot]\!]} [\![M]\!] + [\![N]\!]$$

- El operador de replicación (!) existe con el mismo significado en los dos cálculos:

$$[\![!M]\!] =_{[\![\cdot]\!]} ! [\![M]\!]$$

- La relación de reducción sobre procesos en el π-cálculo está formada por un axioma (la comunicación) y tres reglas de inferencia (composición, restricción y congruencia estructural). Veamos cómo se traducen todos ellos en LCEP:
  - La comunicación del π-cálculo tiene una correspondencia directa con el significado de la comunicación paralela (β_P) de LCEP:

$$[\![(\cdots + x(y).M + \cdots)|(\cdots + \overline{x}z.N + \cdots) \ \to\ M\{z/y\}|N]\!] =_{[\![\cdot]\!]}$$
$$(\cdots + \lambda_x y.[\![M]\!]) \parallel (\widehat{_x}z \circ [\![N]\!] + \cdots) \ \to_{\beta_x}\ [\![M]\!][z/y] \parallel [\![N]\!]$$

  - Existe una regla de inferencia en LCEP similar a la de composición en el π-cálculo.
  - No hay una correspondencia directa a la regla de inferencia para la restricción en el π-cálculo. La localidad de las variables en LCEP sigue la del λ-cálculo y, por tanto, la regla relacionada con la restricción no es necesaria en este caso.
  - Por último, hay en LCEP una regla de inferencia similar a la de congruencia estructural del π-cálculo.

Con todas las restricciones que hemos impuesto a LCEP:

- asumiendo que no son representables las etiquetas numéricas,

- que las reglas de reordenación no tienen su análogo en el π-cálculo,
- que alguna de las reglas de inferencia de LCEP no se puede representar en el π-cálculo, y
- que tampoco existe la β-comunicación secuencial,

plantearse la función de traducción de LCEP al π-cálculo nos lleva a definir la siguiente función de traducción:

$$[\![\cdot]\!] \;:\; LCEP \;\rightarrow\; \pi\text{-cálculo}$$

Para cada elemento sintáctico de los cálculos, esta función queda definida como sigue:

- El proceso nulo es un elemento de ambas propuestas:

$$[\![0]\!] \;=_{[\![\cdot]\!]}\; 0$$

- La noción de variable tiene una correspondencia entre ellas:

$$[\![x]\!] \;=_{[\![\cdot]\!]}\; x$$

- Una abstracción etiquetada con el canal P en LCEP se corresponde a un prefijo de entrada en el π-cálculo, cuyo sujeto es el canal P y cuyo objeto es la misma variable ligada:

$$[\![\lambda_p x.M]\!] \;=_{[\![\cdot]\!]}\; p(x).[\![M]\!]$$

- Un aplicador con canal de comunicación P en LCEP es equivalente a un prefijo de salida en el π-cálculo, con sujeto el canal P:

$$[\![\widehat{_p}M \circ N]\!] \;=_{[\![\cdot]\!]}\; \overline{p}M.[\![N]\!]$$

- El operador de paralelismo tiene un significado similar en ambos cálculos y, por tanto, tiene una correspondencia directa:

$$[\![M \;\parallel\; N]\!] \;=_{[\![\cdot]\!]}\; [\![M]\!] \mid [\![N]\!]$$

- En los dos cálculos existe un operador de elección indeterminista:

$$[\![M \;+\; N]\!] \;=_{[\![\cdot]\!]}\; [\![M]\!] \;+\; [\![N]\!]$$

- El operador de replicación de LCEP es similar al del π-cálculo, puesto que es un operador heredado de él:

$$[\![!M]\!] \;=_{[\![\cdot]\!]}\; !\,[\![M]\!]$$

- Por último, existe una correspondencia inmediata solamente entre la β-comunicación paralela de LCEP y la β-comunicación del π-cálculo, pero no con respecto a la β-comunicación secuencial de LCEP:

$$\begin{array}{c} [\![(\cdots + \lambda_p x.M) \parallel (\widehat{p} N \circ L + \cdots) \ \rightarrow_{\beta_P} \ M[N/x] \parallel L]\!] \ =_{[\![\cdot]\!]} \\ (\cdots + p(x).[\![M]\!]) \mid \overline{p}[\![N.L]\!] + \ldots) \ \rightarrow \ [\![M]\!]\{N/x\}|[\![L]\!] \end{array}$$

Capítulo 4

Ejecución de procesos LCEP

Los sistemas de inferencia (SI) son un modo usual de describir y analizar modelos computacionales. La reescritura de términos juega un papel importante en el área de la computación funcional ya que facilita una mecanización del proceso de inferencia que hace ejecutables las especificaciones. Vamos a utilizar un nuevo tipo de relación de reescritura, la *reescritura sensible al contexto* [Luc95a, Luc95b, Luc96, LO94a], en la que se muestra cómo podemos asociar un sistema de reescritura de términos a cada elemento de una clase particular de sistemas de inferencia que define la misma relación de reducción. Aplicaremos este nuevo marco construyendo un sistema de reescritura que puede verse como una mecanización del π-cálculo de R. Milner, descrito en un capítulo anterior y haciendo posteriormente lo mismo con LCEP.

4.1 Introducción

Vamos a comenzar este capítulo describiendo las principales aportaciones de la *reescritura sensible al contexto* (SRSC), de Salvador Lucas. Para mayor detalle ver [Luc95b, Luc95a, Luc96].

Los sistemas de inferencia (SI) se utilizan ampliamente para capturar y modelizar las propiedades abstractas de la computación. Los SI nos permiten expresar las propiedades estáticas y el comportamiento dinámico de los sistemas representados [Smu71]. La mayoría de los SI producen un gran espacio de búsqueda y por ello se han desarrollado estrategias para reducir ese espacio eliminando algunas derivaciones inútiles. La teoría de los sistemas de reescritura de términos (SRT) se ha mostrado muy útil para mecanizar la ejecución de programas funcionales [EM85], imponiendo direccionalidad en el uso de las ecuaciones en las pruebas [DJ90, Der93]. Se usan ecuaciones dirigidas, llamadas "reglas de reescritura", para reemplazar iguales por iguales, pero sólo en la dirección indicada. En este esquema, la computación consiste

en reescribir términos a una forma normal, es decir, a una expresión que no puede reescribirse más [Der93]. La noción de secuencia de reescrituras tradicionalmente define una visión operacional de la computación. El origen ecuacional de las técnicas de reescritura de términos lleva a considerar una expresión $t \to s$ como "*t igual a s*" en lugar de "*t se convierte en s*" [Mes92]. Como una consecuencia de esto, la propiedad de "*ser cerrado bajo aplicación de contexto*" (una forma particular de regularidad) es esencial en la definición de la relación de reescritura. La definición de la *reescritura sensible al contexto* está basada en la propiedad algebraica de la regularidad. Esta propiedad puede particularizarse a cada símbolo en la signatura Σ del programa por medio de una función $\mu : \Sigma \to 2^{\mathbb{N}}$ y se relaciona con una clase particular de SI. Esto permite formular una relación de reescritura, que llamamos sensible al contexto, que puede verse como una mecanización de los SI.

La idea en que se basa el concepto de reescritura sensible al contexto es imponer una condición de reemplazamiento que puede impedir alguno de los posibles pasos de reducción.

Informalmente, dado un término t, el subtérmino de t a la ocurrencia u es un "*redex*" de la relación de reescritura sensible al contexto si los símbolos en t que etiquetan las ocurrencias superiores a u satisfacen una condición particular que se llama *condición de reemplazamiento*. La condición de reemplazamiento se da en términos de la función μ.

Definición 4.1.1 (Relación de reescritura en un paso [DJ90]) *Dado un SRT $\mathcal{R} = (\Sigma, R)$ y los términos $t, s \in \mathcal{T}(\Sigma \cup V)$, decimos que t se reescribe a s, y lo denotamos por $t \to_{\mathcal{R}} s$, si $t|_u = l\sigma$, $s = t[r\sigma]_u$, $l \to r$ es una regla de $\mathcal{R}$, u es una ocurrencia de t y σ es una sustitución. $\to_{\mathcal{R}}$ es la relación de reescritura en un paso para $\mathcal{R}$ y $\to^*_{\mathcal{R}}$ es el cierre reflexivo y transitivo de dicha relación. Un subtérmino $t|_u$ que es reducible por la regla $l \to r$ con sustitución σ se dice que es un* redex *y a $r\sigma$ se le llama el* contracto *de $t|_u$.*

Definición 4.1.2 (Función de reemplazamiento) *Sea Σ una signatura. Una función $\mu : \Sigma \to 2^{\mathbb{N}}$ es una función de reemplazamiento (o Σ-función) para la signatura Σ si y solamente si para toda $f \in \Sigma$ se cumple que $\mu(\mathrm{f}) \subseteq \mathbb{N}^+_{ar(\mathrm{f})}$.*

Definición 4.1.3 (Condición de reemplazamiento) *Sea Σ una signatura y sea μ una Σ-función. Sea $t \in \mathcal{T}(\Sigma \cup V)$ un término. La condición de reemplazamiento ζ_t con respecto a μ es una relación ζ_t definida sobre el conjunto de ocurrencias $O(t)$ como sigue:*

$$\zeta_t(\varepsilon).$$

$$\zeta_{\mathrm{f}(t_1,\ldots,t_i,\ldots,t_k)}(i.u) \Leftrightarrow (i \in \mu(\mathrm{f})) \wedge \zeta_{t_i}(u).$$

Decimos que la ocurrencia u de un término t satisface la condición de reemplazamiento ζ_t si y solamente si $\zeta_t(u)$.

Informalmente hablando, la condición de reemplazamiento para un término t con respecto a μ impone una restricción sobre los símbolos de función que etiquetan el camino entre la raíz de t y el subtérmino a la ocurrencia u. Si $u = i.j.\cdots.k$, entonces necesitamos observar en las ocurrencias $\varepsilon < i < i.j < \cdots < i.j.\cdots.k = u$ sobre u para determinar si se cumple o no la condición de reemplazamiento $\zeta_t(u)$.

Definición 4.1.4 (Relación sensible al contexto en un paso) *Sea $\mathcal{R} = (\Sigma, R)$ un SRT. Sea μ una Σ-función. Un término $t \in \mathcal{T}(\Sigma \cup V)$ se μ-reescribe a un término $s \in \mathcal{T}(\Sigma \cup V)$, escrito $t \hookrightarrow_{\mathcal{R}(\mu)} s$, si $t \to_{\mathcal{R}} s$ y la ocurrencia seleccionada u satisface ζ_t con respecto a μ. $\hookrightarrow_{\mathcal{R}(\mu)}$ es la relación de reescritura sensible al contexto en un paso de $\mathcal{R}$ con respecto a μ. $\hookrightarrow^*_{\mathcal{R}(\mu)}$ es el cierre reflexivo y transitivo de $\mathcal{R}$ con respecto a μ.*

Existen algunas conexiones entre la relación de reescritura sensible al contexto para un SRT $\mathcal{R} = (\Sigma, R)$ y una Σ-función μ, y una clase particular de sistemas de inferencia.

Consideremos el SI $\text{SI}^\diamond = \langle K, \mathcal{V}, \mathcal{P}, \mathsf{A}, \mathsf{R}\rangle$ con $K = \Sigma \cup V$. Sea $\mathcal{V} = \mathcal{V}_V \biguplus \mathcal{V}_I$ el conjunto de $\text{SI}^\diamond$-variables, donde el conjunto $V = \{x_1, \ldots, x_n\}$ de variables de $\mathcal{R}$ está en una correspondencia "uno a uno" con el conjunto $\mathcal{V}_V = \{X_1, \ldots, X_n\}$ de variables de $\text{SI}^\diamond$. Formalmente, $\nu : V \to \mathcal{V}_V$ y $X_i = \nu(x_i)$. El conjunto de variables en $\text{SI}^\diamond$ que no están en $\mathcal{V}_V$ se denota como $\mathcal{V}_I$. $\mathcal{P}$ contiene un único símbolo de predicado binario $\diamond$, que se usa como un functor infijo. Entonces, el conjunto Φ de fórmulas bien formadas de $\text{SI}^\diamond$ es $\Phi = \{T \diamond S \mid T, S \in \mathcal{T}(\Sigma \cup V \cup \mathcal{V})\}$.

El conjunto A de (esquemas de) axiomas de $\text{SI}^\diamond$ se denota como $\{\mathsf{a}, \mathsf{b}, \ldots\} \subseteq \Phi$. Escribimos $(\mathsf{a})\dfrac{}{T \diamond S}$ cuando $\mathsf{a} = T \diamond S \in \mathsf{A}$, como es usual. El esquema de regla de inferencia de $\text{SI}^\diamond$ es una relación k-aria sobre Φ, con $k > 1$. Escribimos $(\mathsf{r})\dfrac{T_1 \diamond S_1, \ldots, T_{k-1} \diamond S_{k-1}}{T \diamond S}$ cuando $\mathsf{r} = (T_1 \diamond S_1, \ldots, T_{k-1} \diamond S_{k-1}, T \diamond S)$ es un esquema de regla de inferencia de $\text{SI}^\diamond$.

De acuerdo a la definición de axioma como una instancia de un esquema de axioma, una regla de inferencia como la instancia de un esquema de regla, y el concepto de demostración, todos los teoremas son fórmulas bien formadas "básicas" (con respecto a $\mathcal{V}$) $t \diamond s$ con $t, s \in \mathcal{T}(\Sigma \cup V)$.

El conjunto de todos los teoremas de $\text{SI}^\diamond$ puede verse como una relación $\diamond$ sobre $\mathcal{T}(\Sigma \cup V)$. Se define la relación (simbólica) $\diamond \subseteq \mathcal{T}(\Sigma \cup V) \times \mathcal{T}(\Sigma \cup V)$ inducida por $\text{SI}^\diamond$ como:

$$t \diamond s \Longleftrightarrow \vdash_{IS^\diamond} t \diamond s$$

Definición 4.1.5 (SI básico) *Sea* $\mathcal{R} = (\Sigma, R)$ *un SRT. Sea* μ *una* Σ*-función. El sistema de inferencia básico asociado a* $\mathcal{R}$ *y* μ*, denotado como* $I(\mathcal{R}, \mu) = \langle \Sigma \cup V, \mathcal{V}, \{\diamond\}, \mathsf{A}_R, \mathsf{R}_\mu \rangle$*, se define de la siguiente forma:*

1. $\alpha : l \to r \in R$ *si y solamente si*

$$(\mathsf{a}_\alpha)\frac{}{T_l \diamond T_r} \in \mathsf{A}_R$$

2. *Para cada* $\mathrm{f} \in \Sigma$*,* $k = ar(\mathrm{f})$*,* $i \in \mu(\mathrm{f})$ *y* $X_1, \ldots, X_k, Y_i$ *variables distintas en* $\mathcal{V}$

$$(\mathsf{r}[\mathrm{f}, i])\frac{X_i \diamond Y_i}{\mathrm{f}(X_1, \ldots, X_i, \ldots, X_k) \diamond \mathrm{f}(X_1, \ldots, Y_i, \ldots, X_k)} \in \mathsf{R}_\mu$$

La relación definida por $I(\mathcal{R}, \mu)$ *en* $\mathcal{T}(\Sigma \cup V)$ *se denota como* $\to_{I(\mathcal{R},\mu)}$*.*

Esta definición formaliza la clase de SI que mecanizan la reescritura sensible al contexto. Informalmente hablando, las reglas de inferencia necesitan acomodarse a los patrones $\mathsf{r}[\mathrm{f}, i]$ y los axiomas deben satisfacer las condiciones $Var(T_r) \subseteq Var(T_l)$ y $T_l \notin \mathcal{V}_V$, que hacen posible reformularlos como reglas de reescritura.

Definición 4.1.6 (SI transitivo) *Sea* $\mathcal{R} = (\Sigma, R)$ *un SRT. Sea* μ *una* Σ*-función. Sea* $I(\mathcal{R}, \mu)^* = \langle \Sigma \cup V, \mathcal{V}, \{\diamond\}, \mathsf{A}_R \cup \{\rho\}, \mathsf{R}_\mu \cup \{\tau\} \rangle$*. El sistema de inferencia* $I(\mathcal{R}, \mu)^*$ *es el* sistema de inferencia transitivo *asociado a* $\mathcal{R}$ *y a* μ*.*

Informalmente hablando, $I(\mathcal{R}, \mu)^*$ extiende $I(\mathcal{R}, \mu)$ añadiendo el axioma de reflexividad y la regla de transitividad. La menor relación sobre $\mathcal{T}(\Sigma \cup V)$ generada por $I(\mathcal{R}, \mu)^*$ se denota como $\to_{I(\mathcal{R},\mu)^*}$.

En [Luc95b] se demuestra que una relación de reescritura sensible al contexto se puede construir a partir de un sistema de inferencia de acuerdo a la Definición 4.1.6.

Proposición 4.1.1 (Generando la relación en un paso [Luc95b]**)**
Sea $\mathcal{R} = (\Sigma, R)$ *un SRT y* μ *una* Σ*-función. Sea* $I(\mathcal{R}, \mu) = \langle \Sigma \cup V, \mathcal{V}, \{\diamond\}, \mathsf{A}_R, \mathsf{R}_\mu \rangle$ *el sistema de inferencia básico asociado a* $\mathcal{R}$ *y* μ*. Sea* $\hookrightarrow_{\mathcal{R}(\mu)}$ *la relación de reescritura sensible al contexto en un paso para* $\mathcal{R}$ *y* μ*. Entonces,* $\to_{I(\mathcal{R},\mu)} = \hookrightarrow_{\mathcal{R}(\mu)}$*.*

4.2 Mecanización del π-cálculo

A continuación, vamos a mostrar brevemente una primera aplicación interesante de la relación de reescritura sensible al contexto. Concretamente, la mecanización del π-cálculo utilizando este nuevo marco.

El π-cálculo de R. Milner [Mil93d, MPW92, Pie93], como ya hemos visto, es un esquema computacional que modeliza la concurrencia. En el π-cálculo, el conjunto $\mathcal{P}$ de procesos $P \in \mathcal{P}$ se define por:

$$P \quad ::= \quad \sum_{i \in I} \pi_i.P_i \ \mid \ (P \mid Q) \ \mid !P \ \mid \ (\nu x)P$$

donde $\pi_i ::= x(y) \ \mid \ \overline{x}y$, para $x, \overline{x}, y \in \mathcal{X}$ (un conjunto de *nombres*), representa las acciones de comunicación básicas: *input* ($x(y)$) y *output* ($\overline{x}y$).

Los constructores de procesos son:
| (paralelismo), ! (replicación), νx (restricción) y + (elección indeterminista).

Para simplificar el sistema de reducción se identifican algunas expresiones a través de una congruencia $\equiv$ sobre $\mathcal{P}$. La *relación de transición* $\rightarrow \subseteq \mathcal{P} \times \mathcal{P}$ definida a continuación formaliza el proceso de reducción [Mil93d].

Axiomas:

$$\text{COM:} \quad (\cdots + x(y).P + \cdots) \mid (\cdots + \overline{x}z.Q + \cdots) \rightarrow P[z/y] \mid Q$$

Reglas de inferencia:

$$\text{PAR:} \qquad \frac{P \rightarrow P'}{P \mid Q \rightarrow P' \mid Q}$$

$$\text{STRUCT:} \quad \frac{Q \equiv P, P \rightarrow P', P' \equiv Q'}{Q \rightarrow Q'}$$

$$RES: \qquad \frac{P \rightarrow P'}{(\nu x)P \rightarrow (\nu x)P'}$$

Notemos que la regla STRUCT no empareja con el esquema de regla considerado en la Definición 4.1.5. Podemos transformar $\rightarrow$ en una relación "equivalente" considerando (clases de) procesos (módulo $\equiv$) dados por el cociente $\mathcal{P}/\equiv$. Entonces, podemos trabajar con reescritura ecuacional (módulo $\equiv$), ver [DJ90, Mes92]. Con $[t] \rightarrow [s]$ queremos expresar que podemos reducir (módulo $\equiv$) cualquier término t' que verifique $t' \equiv t$ a cualquier término s' que verifique $s' \equiv s$. Es sencillo demostrar que el nuevo sistema de inferencia:

$$[COM]: \ [(\cdots + x(y).P + \cdots) \mid (\cdots + \overline{x}z.Q + \cdots)] \rightarrow [P[z/y] \mid Q]$$

$$[\text{PAR}]: \ \frac{[P] \rightarrow [P']}{[P \mid Q] \rightarrow [P' \mid Q]} \qquad [\text{RES}]: \ \frac{[P] \rightarrow [P']}{[(\nu x)P] \rightarrow [(\nu x)P']}$$

induce la misma relación de reducción que el original, en el sentido de que $P' \rightarrow Q' \Longleftrightarrow [P] \rightarrow [Q]$ y $P' \equiv P$, $Q' \equiv Q$.

El siguiente ejemplo ilustra que un uso arbitrario (independiente del contexto) del axioma de comunicación como una regla de reescritura (módulo $\equiv$) conduce a secuencias de ejecución incorrectas.

1. $P = u(v).(x(y) \mid \overline{x}(z))$. Una primera interpretación de la regla de reescritura [COM] podría llevarnos a $u(v).(0 \mid 0) \equiv u(v)$, que no se corresponde con ninguna reducción en el π-cálculo.

2. Asumiendo que $P \rightarrow P'$, entonces $!P$ se reduce a $!P'$, que tampoco es una reducción en el π-cálculo.

A continuación vamos a definir el SRT $\mathcal{R}_\pi = (\Sigma_\pi, \{[\mathrm{COM}_\pi]\})$ y la Σ-función μ_π:

- Σ_π: Queremos formular una descripción precisa de los procesos a través de términos de la signatura Σ_π. Primero remarquemos los siguientes hechos:

  – No podemos considerar el nombre x en el axioma [COM] como un símbolo de variable, ya que no lo es.

  – No podemos considerar x como un símbolo de función arbitrario, ya que tiene un dominio de "nombres".

  – No es posible considerar . como un símbolo de función arbitrario, ya que es asimétrico (la posición de la parte izquierda solamente puede ser instanciada con comunicaciones π_i's).

  En este caso, se cumple que podemos considerar los procesos de comunicación $x(y)$ y $\overline{x}y$ como funciones unarias i_{xy} y o_{xy}, respectivamente. Entonces, definimos la signatura $\Sigma_\pi = \Pi \cup \overline{\Pi} \cup \nabla \cup \{|, !, +, 0\}$, donde $\Pi = \{i_{xy} \mid x, y \in \mathcal{X}\}$ es el conjunto de las funciones de *input*, $\overline{\Pi} = \{o_{xy} \mid x, y \in \mathcal{X}\}$ es el conjunto de las funciones de *output* y $\nabla = \{\nu_x \mid x \in \mathcal{X}\}$ es el conjunto de las funciones de *restricción*. Las aridades de las funciones son las siguientes: $ar(i_{xy}) = ar(o_{xy}) = ar(\nu_x) = 1$, $ar(0) = 0$, $ar(|) = ar(+) = 2$, $ar(!) = 1$. Así, $[\mathrm{COM}_\pi]$ es la reformulación de [COM] usando Σ_π.

- μ_π: De la Definición 4.1.5, se puede deducir que las únicas reglas de inferencia son $\mathsf{r}[\mathrm{f}, i]$ (para alguna f e i). Así, dado un símbolo de función $\mathrm{f} \in \Sigma_\pi$, si existe una regla de inferencia $\mathsf{r}[\mathrm{f}, i]$, entonces i pertenece a $\mu_\pi(\mathrm{f})$. Viceversa, cada índice $i \in \mu_\pi(\mathrm{f})$ produce una correspondiente regla $\mathsf{r}[\mathrm{f}, i]$ en el SI. En otras palabras, μ_π asigna un conjunto vacío a todo constructor de proceso en $\Pi \cup \overline{\Pi} \cup \{!, +, 0\}$ ya que no hay ninguna regla de inferencia $\mathsf{r}[\mathrm{f}, i]$ asociada. En relación con [RES] y [PAR], $\mu_\pi(\nu_x) = \{1\}$ y $\mu_\pi(|) = \{1\}$, respectivamente.

4.3 Mecanización de LCEP

Como acabamos de ver, se pueden utilizar los sistemas de reescritura sensibles al contexto [Luc95b] para mejorar la eficiencia de la mecanización de un sistema definido mediante una serie de reglas de inferencia. Examinando los axiomas de comunicación

y reordenación para LCEP (ver el Capítulo 3 de la tesis) podemos observar que, considerando los axiomas de comunicación y reordenación y las reglas de inferencia que hacen referencia a los operadores de construcción de procesos, utilizando la Definición 4.1.5, se puede construir un sistema de reescritura sensible al contexto cuyo sistema de inferencia asociado sea el de partida. Esta será una manera de dar un mecanismo operacional de ejecución de procesos LCEP mediante reemplazamientos en el sistema de reescritura sensible al contexto.

Sin embargo, esta forma de proceder no proporciona aún un mecanismo completo de ejecución puesto que no se están considerando:

- las propiedades algebraicas de los operadores,
- la congruencia estructural expresada por la regla μ_{Struct},
- la regla $\mu_{\mathcal{T}}$ que describe la construcción de túneles.

Es evidente que estas dos últimas reglas de inferencia no tienen la forma exigida a las reglas $\mathsf{r}[\mathrm{f}, i]$ que representan el reemplazamiento de la relación de reescritura respecto a la función f y al índice i. Por tanto, la asociación de un sistema de reescritura sensible al contexto a LCEP no es inmediata. Debemos realizar previamente algunas transformaciones que nos permitan formalizar la mecanización de procesos a partir del formalismo LCEP.

4.3.1 El sistema de inferencia de LCEP como generador de relaciones

Vamos a estudiar cómo utilizar en LCEP los axiomas de comunicación, los axiomas de reordenación y las reglas de inferencia para definir la relación de reducción correspondiente.

1. Recordemos que la regla de inferencia $\mu_{\mathcal{T}}$:

$$\mu_{\mathcal{T}} \quad : \quad \frac{M \parallel N \ \longrightarrow_{\beta_P} \ M' \parallel N'}{\widehat{Q_1} M \parallel \widehat{Q_2} N \ \longrightarrow_{\beta_P} \ \widehat{Q_1} M' \parallel \widehat{Q_2} N'}$$

   para $Q_1, Q_2 \in \mathcal{L}$ si $P \succ_{\mathcal{L}} Q_1, Q_2$, puede interpretarse como sigue:

   Si dos procesos M y N compuestos en paralelo establecen comunicación paralela a través del canal P, es decir, $M \parallel N \ \longrightarrow_{\beta_P} \ M' \parallel N'$, entonces al aplicar los constructores de output $\widehat{Q_1}$ y $\widehat{Q_2}$ que verifican la condición en función de la relación de orden $\succ_{\mathcal{L}}$, es también posible inferir una comunicación paralela sobre P, que transforma M en M' y N en N' del mismo modo.

   Esta regla de inferencia extiende la relación β_P.

2. Los axiomas de comunicación y reordenación y las propiedades algebraicas expresan conceptos distintos. En realidad, las propiedades algebraicas hacen referencia a características intrínsecas de los operadores empleados en la construcción de los procesos y son bidireccionales, mientras que los axiomas sí pueden ser orientados.

Vamos a describir cómo definir un sistema de inferencia generador de relaciones a partir de los axiomas de comunicación y reordenación de LCEP (a los que vamos a llamar axiomas LCEP) y de las reglas de inferencia y de túnel (a las que denominamos reglas de inferencia LCEP). La relación generada por el sistema de inferencia de LCEP es $\rightarrow \subseteq T(\mathcal{M}) \times T(\mathcal{M})$, que está en biyección con la relación original $\rightarrow \subseteq \mathcal{M} \times \mathcal{M}$.

Interpretación algebraica y sistema generador de la relación de reducción

La interpretación algebraica realizada para los símbolos del lenguaje LCEP permitió, en el Capítulo 3, describir las características dinámicas de los procesos LCEP de una forma simple y precisa. La interpretación de la comunicación a través de túneles en un contexto geométrico, en términos del árbol sintáctico del proceso donde ocurren, se corresponde con la elección realizada de la signatura $\Sigma_{\mathcal{M}}$ que se ha dado.

Definición 4.3.1 ($\mathcal{M}$-sustitución) *Sea $W' \subseteq W$. Decimos que σ es una $\mathcal{M}$-*sustitución *si es una sustitución de la forma $\sigma : W' \rightarrow T(\mathcal{M})$. Informalmente, se trata de una sustitución que asocia a cada variable de W' un proceso LCEP.*

A continuación, describimos qué esquemas de axiomas podemos establecer para LCEP y la relación que definen en $T(\mathcal{M})$. Puesto que la signatura sobre la que trabajamos es potencialmente infinita, en realidad cada regla LCEP se interpreta como un conjunto infinito de esquemas de axiomas. La infinitud de la signatura viene determinada por los dos conjuntos infinitos de operadores: $\mathcal{P}$ (el conjunto de operadores de output) y Λ (el conjunto de operadores de input). No obstante, dichos conjuntos de operadores se definen en términos de $\mathcal{L}$ y V. Por esta razón, todas las reglas definen conjuntos infinitos de esquemas de axiomas indexados con la ayuda de dichos conjuntos. En el caso de los axiomas de reordenación, donde interviene la relación de orden para decidir cuándo puede o no aplicarse una regla, podemos escribir el conjunto de esquemas de axiomas indexándolo con la propia relación de orden.

En lo que sigue, vamos a asumir que $L, M, N, X, Y \in W$, $x, y \in V$ y $m, n \in \mathbb{N}^+$.

1. <u>Esquemas de axiomas del sistemas de inferencia</u>:

   (a) <u>Axiomas de comunicación</u>:

$$\begin{aligned} \beta_{sec} \quad = \quad & \{ \ (X + \lambda_P x.M) \circ (\widehat{P} N \circ L + Y) \ \longrightarrow \\ & M[N/x] \circ L : \ P \in \mathcal{L} - \{0\} \ \} \end{aligned}$$

Se trata de un conjunto de esquemas de axioma indexado por el producto cartesiano $\mathcal{L} \times V$.

$$\beta_{par} = \{ \ \beta_P : \ P \in \mathcal{L} - \{0\} \ \}$$

es una familia de axiomas de comunicación indexados por $\mathcal{L} - \{0\}$ y en donde $\beta_P = \{ \ (X + \lambda_P x.M) \parallel (\widehat{_P}N \circ L + Y) \ \longrightarrow \ M[N/x] \parallel L \ \}$, es decir, cada regla de comunicación define un conjunto de esquemas de axioma indexado por V.

(b) Axiomas de reordenación:

Para describir las reglas de reordenación empleamos la relación de orden estricta $\prec_{\mathcal{L}}$ definida sobre los conjuntos $\mathbb{N}^+ = \mathbb{N} - \{0\}$ y S, que denotamos como $\prec_{\mathbb{N}^+}$ y $\prec_S$.

i. Reordenación con etiquetas simbólicas:

$$\rho_1 = \{ \ (\lambda_p x.\lambda_q y.M) \ \longrightarrow \ (\lambda_q y.\lambda_p x.M) : \ x \neq y, p \succ_S q \ \}$$

Se trata de un conjunto de esquemas de axioma indexado por $(V \times V - Id_V) \times \succ_S$.

$$\rho_2 = \{ \ \widehat{_p}M \circ \widehat{_q}N \ \longrightarrow \ \widehat{_q}N \circ \widehat{_p}M : \ p \succ_S q \ \}$$

Se trata de un conjunto de esquemas de axioma indexado por $\succ_S$.

$$\begin{aligned} \rho_{sec_3} \ = \ & \{ \ (X + \lambda_p x.M) \circ (\widehat{_q}N \circ L + Y) \ \longrightarrow \\ & \lambda_p x.(M \circ \widehat{_q}N \circ L) : \ p \succ_S q \ \vee \ p \prec_S q \ \} \end{aligned}$$

Se trata de un conjunto de esquemas de axioma indexado por $\{0,1\} \times V \times \prec_S$.

ii. Reordenación con etiquetas numéricas:

$$\eta_1 = \{ \ (\lambda_m x.\lambda_n y.M) \ \longrightarrow \ (\lambda_n y.\lambda_{m-1} x.M) : \ x \neq y, \ m \succ_{\mathbb{N}} n \ \}$$

Se trata de un conjunto de esquemas de axioma indexado por $(V \times V - Id_V) \times \succ_{\mathbb{N}^+}$.

$$\eta_2 = \{ \ \widehat{_m}M \circ \widehat{_n}N \ \longrightarrow \ \widehat{_n}N \circ \widehat{_{m-1}}M : \ m \succ_{\mathbb{N}} n \ \}$$

Se trata de un conjunto de esquemas de axioma indexado por $\succ_{\mathbb{N}^+}$.

$$\begin{aligned} \eta_{sec_3} \ = \ & \{ \ (X + \lambda_m x.M) \circ (\widehat{_n}N \circ L + Y) \ \longrightarrow \\ & \lambda_{m-1} x.(M \circ \widehat{_n}N \circ L) : \ m \succ_{\mathbb{N}} n \ \} \end{aligned}$$

Se trata de un conjunto de esquemas de axioma indexado por $V \times \succ_{\mathbb{N}^+}$.

$$\begin{aligned} \eta_{sec_4} \ = \ & \{ \ (X + \lambda_m x.M) \circ (\widehat{_n}N \circ L + Y) \ \longrightarrow \\ & \lambda_m x.(M \circ \widehat{_{n-1}}N \circ L) : \ m \prec_{\mathbb{N}} n \ \} \end{aligned}$$

Se trata de un conjunto de esquemas de axioma indexado por $V \times \prec_{\mathbb{N}^+}$.

2. Reglas de inferencia:

En general, también en este caso el número de reglas de inferencia es infinito.

(a) Reglas asociadas al reemplazamiento de los operadores:
Las reglas de inferencia $\mu_{\circ_1}$, $\mu_{\circ_2}$, $\mu_{\|}$, $\mu_{\frown}$, μ_{λ}, presentadas en el Capítulo 3, están relacionadas con la sustitutividad de los operadores algebraicos. Todas ellas definen conjuntos de esquemas de reglas, unitarios en el caso de $\mu_{\circ_1}$, $\mu_{\circ_2}$, $\mu_{\|}$ y potencialmente infinitos en el caso de $\mu_{\frown}$ y μ_{λ}.

(b) Regla de congruencia estructural:
La regla μ_{Struct} es un caso especial porque involucra dos relaciones. Se considera que define, para cada par $M \to N$, esquemas de reglas de inferencia del tipo:

$$\frac{M \longrightarrow N}{M' \longrightarrow N'}$$

donde $M \equiv M'$ y $N \equiv N'$.

(c) Regla del túnel:
Debemos considerar la regla $\mu_{\mathcal{T}}$ como un conjunto de esquemas de reglas de inferencia indexado por $\mathcal{L}$:

$$\mu_{\mathcal{T}} = \{\ \mu_{\mathcal{T},P} :\ P \in \mathcal{L}\ \}$$

donde

$$\mu_{\mathcal{T},P} = \left\{ \begin{array}{l} \dfrac{M \parallel N \ \longrightarrow\ M' \parallel N'}{\widehat{Q_1} M \parallel \widehat{Q_2} N \ \longrightarrow\ \widehat{Q_1} M' \parallel \widehat{Q_2} N'} \quad : \\ \\ Q_1, Q_2 \in \mathcal{L}, P \succ Q_1, Q_2 \end{array} \right\}$$

que define un conjunto de reglas de inferencia indexado por el conjunto $\succ_P \times \succ_P$, donde $\succ_P$ es el conjunto de pares de $\succ$ que verifican que $\exists Q \in \mathcal{L} :\ P \succ Q$.

Como puede observarse, se ha eliminado de las reglas la dependencia respecto de la etiqueta P del canal sobre el cual se realiza la comunicación. Tenemos en cuenta que cada conjunto de reglas de inferencia $\mu_{\mathcal{T},P}$ solamente puede emplearse junto con el axioma β_P correspondiente, de forma que se preserva el significado de la regla $\mu_{\mathcal{T}}$.

4.3.2 Comunicación secuencial y reordenación

Si extraemos los axiomas de comunicación paralela y la regla de inferencia $\mu_{\mathcal{T}}$, la descripción de la relación generada por el sistema de inferencia es inmediata. Se trata, simplemente, de aplicar las reglas de inferencia sobre los axiomas del sistema, de acuerdo a la concepción habitual de derivación. Sin embargo, la regla $\mu_{\mathcal{T}}$ fuerza a abandonar este esquema puesto que describe una familia de reglas de inferencia $\mu_{\mathcal{T},P}$ y cada una de ellas solamente puede aplicarse en un subsistema de inferencia en el que

el único axioma de partida es β_P; es decir, cada axioma β_P representa una relación diferente y la regla $\mu_\mathcal{T}$ permite mantener el carácter de la relación. En cambio, no hay ningún problema en considerar que los axiomas de comunicación secuencial y reordenación representan la misma relación y, por esta razón, se puede aplicar sobre ellas cualquier regla de inferencia.

4.3.3 Comunicación paralela. $\mathcal{T}$-comunicabilidad

Como acabamos de describir, cada axioma β_P representa una relación distinta y la regla de inferencia $\mu_\mathcal{T}$ solamente puede aplicarse por separado a cada una de estas relaciones. Vamos a utilizar cada conjunto de (esquemas de) axiomas β_P para generar una relación $B_P \subseteq T(\mathcal{M}) \times T(\mathcal{M})$ mediante un subsistema de inferencia $SI_{\mathcal{T},P}$ definido como:

$$SI_{\mathcal{T},P} = \langle \beta_P, \mu_{\mathcal{T},P} \cup \mu_{\|} \cup \mu_{Struct} \rangle \quad \text{para } P \in \mathcal{L}$$

Es decir, cada sistema de inferencia consta de un conjunto de esquemas de axioma tomado de la familia β_{par} y tres conjuntos de esquemas de regla de inferencia, de los que uno de ellos se toma de $\mu_\mathcal{T}$. Ambas selecciones se realizan a través de la etiqueta P, ya que β_{par} y $\mu_\mathcal{T}$ están indexados por $\mathcal{L}$. De esta forma, la relación representada por B_P ya no depende del axioma de partida. Entonces, tomando dicha relación como conjunto de axiomas, utilizamos las restantes reglas de inferencia (prescindiendo de $\mu_\mathcal{T}$) para aplicarlas indistintamente sobre dicho conjunto de axiomas.

La relación generada de esta manera forma parte de $\rightarrow$ y expresa la noción de $\mathcal{T}$-comunicabilidad sobre un canal P vista anteriormente. Por esta razón, se denota como $\rightarrow_{\mathcal{T},P}$, y la llamamos relación de $\mathcal{T},P$-reducción.

La unión de las relaciones de $\mathcal{T},P$-reducción sobre cada canal P de $\mathcal{L}$ define la relación de $\mathcal{T}$-reducción en $\mathcal{M}$ representada por $\rightarrow_\mathcal{T}$. Por supuesto, también está incluida en la relación de reducción del formalismo.

Se puede observar que la noción de $\mathcal{T}$-comunicabilidad se construye a partir de consideraciones puramente sintácticas, en función de la estructura sintáctica del término que representa un proceso. La relación de $\mathcal{T}$-comunicabilidad sobre un canal P es una relación dependiente del proceso A sobre el que se localizan los subprocesos $\mathcal{T}$-comunicables (es una relación sobre $O(A)$). El Teorema 3.7.1 establece la conexión entre esta relación y el sistema de inferencia del formalismo. La noción de $\mathcal{T}$-reducción precisa un poco más este vínculo, estableciendo que es suficiente un subconjunto del sistema de inferencia para dar cuenta de la $\mathcal{T}$-comunicabilidad.

Para enlazar el concepto sintáctico de $\mathcal{T}$-comunicabilidad con el de relación de $\mathcal{T}$-reducción en $\mathcal{M}$, introducimos las siguientes definiciones:

Definición 4.3.2 (Relación de $\mathcal{T},P$-comunicabilidad) *Sea A un proceso. Si dos subprocesos $M = A/u$, $N = A/v$ de A son $\mathcal{T}$-comunicables sobre un canal P lo*

denotamos como $u\ \mathcal{T}_{A,P}\ v$*. De esta forma, la relación* $\mathcal{T}_{A,P}$ *puede verse como una relación en* $O(A)$*:* $\mathcal{T}_{A,P} \subseteq O(A) \times O(A)$*.*

Proposición 4.3.1 ($\mathcal{T}, P$-comunicabilidad y $\mathcal{T}, P$-reducción) *Si dos procesos son* $\mathcal{T}, P$*-comunicables entonces son* $\mathcal{T}, P$*-reducibles.*

DEMOSTRACIÓN.

Si dos procesos son $\mathcal{T}, P$-comunicables, eso significa que cumplen las restricciones sintácticas para que se pueda aplicar una β_P-comunicación, que es uno de los axiomas que forman parte del sistema de inferencia $SI_{\mathcal{T},P}$. Como hemos indicado anteriormente, la relación generada de esta manera se denota como $\rightarrow_{\mathcal{T},P}$ y se llama relación de $\mathcal{T}, P$-reducción.

□

Esta proposición establece la conexión entre la $\mathcal{T}, P$-comunicabilidad y el sistema de inferencia LCEP. La implicación en sentido contrario no se da con la definición actual de túnel, ya que la relación $\rightarrow_{\mathcal{T},P}$ incluye procesos congruentes que añaden símbolos que no están contemplados en la definición de túnel. El Teorema 3.7.1 es una consecuencia inmediata de esta proposición.

Definición 4.3.3 (Relación de $\mathcal{T}$-comunicabilidad) *Si prescindimos de considerar el canal sobre el que se da la* $\mathcal{T}$*-comunicación, podemos asociar a cada proceso* $A \in \mathcal{M}$ *una relación* $\mathcal{T}_A \subseteq O(A) \times O(A)$ *definida como:*

$$M\ \mathcal{T}_A\ N \iff \exists P \in \mathcal{L} :\ M\ \mathcal{T}_{A,P}\ N$$

A continuación, vamos a introducir la noción de grafo de $\mathcal{T}$-comunicabilidad de un proceso para poder obtener una visión global de las posibilidades de $\mathcal{T}$-comunicabilidad ofrecidas por un proceso A a los procesos comunicantes localizados en él.

Grafo de $\mathcal{T}$-comunicabilidad de un proceso

Dado un proceso, podemos asociarle un grafo que representa, esquemáticamente, la relación de $\mathcal{T}$-comunicabilidad entre los subprocesos que lo integran.

Definición 4.3.4 (Proceso comunicante maximal) *Sea* A *un proceso. Un proceso comunicante* $N = A/v$ *es* maximal *si no es sumando de ningún proceso comunicante* $M = A/u$*, con* $u \trianglelefteq v$*.*

Por ejemplo, sea $M = M_1 + M_2 + M_3$ un proceso comunicante. Supongamos que todos los M_i son procesos comunicantes elementales. Entonces M_1, $M_1 + M_2$, $M_2 + M_3$ son

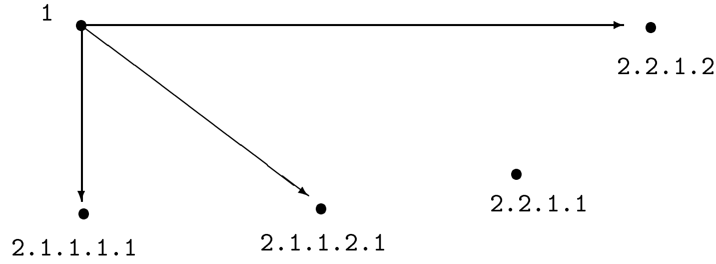


Figura 4.1: Grafo de $\mathcal{T}, F$-comunicabilidad del proceso A

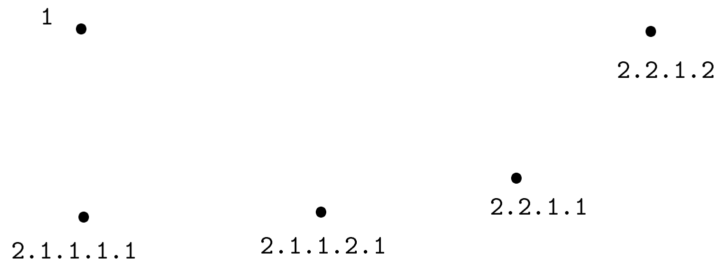


Figura 4.2: Grafo de $\mathcal{T}, G$-comunicabilidad del proceso A

procesos comunicantes pero ninguno de ellos es maximal porque todos son sumandos de M.

Definición 4.3.5 (Grafo de $\mathcal{T}, P$-comunicabilidad) *Sea A un proceso. El grafo de $\mathcal{T}, P$-comunicabilidad de A es el grafo $G_{\mathcal{T},P}(A) = (C_A, A_P)$, donde:*

- *C_A es el conjunto de ocurrencias de los procesos comunicantes maximales.*
- *$A_P \subseteq C_A \times C_A$ es el conjunto de arcos, definidos a partir de la relación de $\mathcal{T}, P$-comunicabilidad como:*

$$A/u \ \mathcal{T}_{A,P} \ A/v \iff \langle u, v \rangle \in A_P$$

El grafo de $\mathcal{T}, P$-comunicabilidad de un proceso A muestra un esquema sencillo de las posibilidades de $\mathcal{T}$-comunicación sobre un canal P de dos procesos comunicantes localizados como subprocesos de A.

Ejemplo 9 *Sea el proceso $A = !\widehat{_F}A' \parallel \widehat{_{P_1}}(\widehat{_Q}(\lambda_F x.M_1) \parallel \widehat{_{P_2}}(\lambda_F x.M_2)) \parallel \widehat{_Q}(!\widehat{_G}A'' \parallel \lambda_F x.M_3)$. La Figura 4.1 muestra el grafo de $\mathcal{T}, F$-comunicabilidad del proceso A. La Figura 4.2 muestra el grafo de $\mathcal{T}, G$-comunicabilidad para el mismo proceso A.*

Observamos que, en este caso, el grafo no tiene arcos. Esto es porque el m-proceso localizado en la ocurrencia 2.2.1.1 no tiene ningún proceso comunicante disponible para establecer contacto con él.

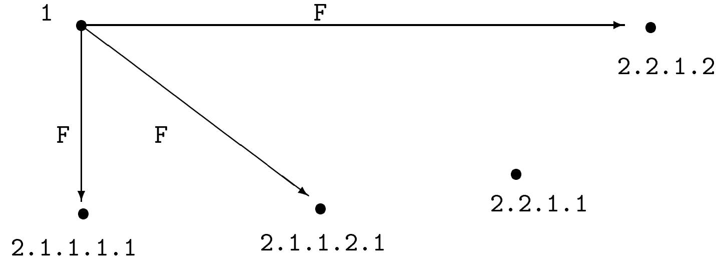


Figura 4.3: Grafo de $\mathcal{T}$-comunicabilidad del proceso A

Como podemos apreciar, el grafo de comunicabilidad solamente muestra un esquema de las relaciones de comunicabilidad entre los procesos comunicantes.

Definición 4.3.6 (Grafo de $\mathcal{T}$-comunicabilidad de un proceso) *Sea A un proceso. Sea C_A el conjunto de ocurrencias de los procesos comunicantes maximales. Sea K_A el conjunto de canales soporte de todos los procesos comunicantes en C_A, es decir, $K_A = \bigcup_{u \in C_A} \zeta(A/u)$, donde K_A es un conjunto finito. Sea $k = |K_A|$. Denotamos por $G_{\mathcal{T},P_1}(A), G_{\mathcal{T},P_2}(A), \ldots, G_{\mathcal{T},P_k}(A)$, con $P_i \in K_A$, el conjunto de todos los grafos de $\mathcal{T}, P$-comunicabilidad para A. El grafo k-coloreado:*

$$G_{\mathcal{T}}(A) = (C_A, A_{P_1}, A_{P_2}, \ldots, A_{P_k})$$

es el grafo de $\mathcal{T}$-comunicabilidad *del proceso A [Ber83].*

Los canales de comunicación juegan el papel de los colores en el grafo etiquetado. Para distinguir el color correspondiente a cada etiqueta, etiquetamos cada arco del grafo con el canal asociado al grafo de $\mathcal{T}, P$-comunicabilidad del que procede. Por ejemplo, en Figura 4.3 mostramos el grafo de $\mathcal{T}$-comunicabilidad para el proceso A anterior. El grafo es la unión de los grafos de las dos figuras previas ya que $K_A = \{F, G\}$. Para distinguir los arcos de F y G añadimos el canal que soporta la comunicación. El grafo de $\mathcal{T}$-comunicabilidad de un proceso representa, por tanto, la relación de $\mathcal{T}$-comunicabilidad establecida en él.

Una línea de trabajo interesante a seguir sería profundizar en el estudio de los grafos de $\mathcal{T}$-comunicabilidad de un proceso como mecanismo para estudiar las posibilidades de comunicación de un sistema.

4.4 Reducción y sistema de inferencia de LCEP

Vamos a definir la relación de reducción $\rightarrow$ para $\mathcal{M}$.

4.4.1 Relación generada por el sistema de inferencia

La relación de reducción se define planteando un sistema de inferencia $SI_{\mathcal{M}}$ que utiliza axiomas generados a su vez por los sistemas de inferencia $SI_{\mathcal{T},P}$.

El sistema de inferencia generador de $\rightarrow$ es:

$$SI_{\mathcal{M}} = \langle\ \mathcal{A}_{\mathcal{M}}, \mathcal{R}_{\mathcal{M}}\ \rangle$$

donde:

- $\mathcal{A}_{\mathcal{M}} = \beta_{sec} \cup \rho_1 \cup \rho_2 \cup \rho_{sec_3} \cup \mu_1 \cup \mu_2 \cup \mu_{sec_3} \cup \mu_{sec_4} \cup \bigcup_{P \in \mathcal{L}} B_P$ define el conjunto de (esquemas de) axiomas del sistema de inferencia. Los axiomas B_P se generan mediante los sistemas de inferencia auxiliares $SI_{\mathcal{T},P}$.

- $\mathcal{R}_{\mathcal{M}} = \mu_{\circ_1} \cup \mu_{\circ_2} \cup \mu_{\|} \cup \mu_{\lambda} \cup \mu_{\frown} \cup \mu_{Struct}$ define el conjunto de (esquemas de) reglas del sistema de inferencia.

4.4.2 Sistema de inferencia asociado a un SRTSC para LCEP

Considerando la definición del sistema $SI_{\mathcal{M}}$, podemos advertir que la regla $\mu_{\mathcal{T}}$ no se encuentra entre las reglas de inferencia del sistema. Esta regla establece una restricción implícita respecto a la relación que es capaz de generar a partir de la representada por los axiomas del sistema de inferencia. Por ello se impone la utilización de un único axioma (o varios, pero representando la misma relación) como punto de partida para generar la relación.

Por otro lado, la regla de congruencia estructural μ_{Struct} no nos permite asociar el sistema de inferencia $SI_{\mathcal{M}}$ con un SRTSC, puesto que no tiene el formato exigido a las reglas de inferencia del sistema de reescritura [Luc95b, Luc95a].

Para conseguir un sistema de inferencia compatible con un SRTSC vamos a trabajar, siguiendo el tratamiento dado en [Mes92], con axiomas definidos sobre clases de congruencia según $\equiv$. En [Mes92] se distingue, en la especificación de un sistema de inferencia para razonar sobre reescritura, entre reglas de reescritura tomadas de un conjunto R y ecuaciones que expresan propiedades algebraicas de las operaciones de la signatura con la que se trabaja, tomadas de un conjunto E. Las ecuaciones E definen una congruencia $\equiv_E$. Dada una regla de reescritura $t_1 \rightarrow t_2$ en R, con $t_1, t_2 \in T_\Sigma(V)$, en [Mes92] se trabaja con la regla de reescritura $[t_1] \rightarrow [t_2]$, con $[t_1], [t_2] \in T_\Sigma(V)/\equiv_E$. Por lo tanto, $[t_1] \rightarrow [t_2]$ representa, en realidad, un conjunto de reglas de reescritura en $T_\Sigma(V)$: $[t_1] \rightarrow [t_2] = \{t'_1 \rightarrow t'_2 : t'_1 \equiv_E t_1 \wedge t'_2 \equiv_E t_2\}$. Esta idea es la que expresa la regla de congruencia estructural. De acuerdo con este planteamiento, definimos el sistema de inferencia.

El sistema de inferencia se denota como $[SI_{\mathcal{M}}]$, y se define como sigue:

$$[SI_{\mathcal{M}}] = \langle\ [\mathcal{A}_{\mathcal{M}}], [\mathcal{R}_{\mathcal{M}}]\ \rangle$$

donde:

$$\begin{aligned}[\mathcal{A}_{\mathcal{M}}] &= [\beta_{sec}] \cup [\rho_1] \cup [\rho_2] \cup [\rho_{sec_3}] \cup [\mu_1] \cup [\mu_2] \cup \\ &\cup [\mu_{sec_3}] \cup [\mu_{sec_4}] \cup \bigcup_{P \in \mathcal{L}} [B_P]\end{aligned}$$

es el conjunto de esquemas de axioma del sistema de inferencia, construido a partir de los siguientes axiomas módulo $\equiv$:

$$[\beta_{sec}] = \{[t_1] \to [t_2] : \; t_1 \to t_2 \in \beta_{sec}\}$$

$$[\rho_1] = \{[t_1] \to [t_2] : \; t_1 \to t_2 \in \rho_1\}$$

$$\ldots$$

$$[\mu_{sec_4}] = \{[t_1] \to [t_2] : \; t_1 \to t_2 \in \mu_{sec_4}\}$$

Al considerar el conjunto $[\mathcal{A}_{\mathcal{M}}]$, cualquier esquema de axioma tomado de dicho conjunto, $[a] = [t] \to [s]$ con $t, s \in T_{\Sigma_{\mathcal{M}}}(W)$, tiene asociado un conjunto de axiomas $A_a = \{t' \to s' : \; t' \equiv t \; \wedge \; s' \equiv s\}$. El conjunto de axiomas $\mathcal{A}_{\mathcal{M}_0}$ es la unión de todos los axiomas asociados a cada esquema de axioma en $\mathcal{A}_{\mathcal{M}}$.

Los axiomas $[B_P]$ se generan, de forma análoga a como se estableció para $SI_{\mathcal{M}}$, mediante un conjunto de sistemas de inferencia:

$$[SI_{\mathcal{T},P}] = \langle \; [\beta_P], [\mu_{\mathcal{T},P}] \cup \{[\mu_{\|}]\} \; \rangle \quad \text{para } P \in \mathcal{L}$$

donde $[\beta_P]$, el conjunto de esquemas de axioma, se describe análogamente.

Por otro lado, $[\mu_{\mathcal{T},P}]$ es un conjunto de reglas de inferencia construido a partir de $\mu_{\mathcal{T},P}$:

$$[\mu_{\mathcal{T},P}] = \left\{ \; \frac{[M \parallel N] \longrightarrow [M' \parallel N']}{[\widehat{Q_1} M \parallel \widehat{Q_2} N] \longrightarrow [\widehat{Q_1} M' \parallel \widehat{Q_2} N']} : \quad Q_1, Q_2 \in \mathcal{L}, P \succ Q_1, Q_2 \; \right\}$$

Análogamente, la regla $[\mu_{\|}]$ se define como:

$$[\mu_{\|}] = \frac{[M] \longrightarrow [M']}{[M \parallel N] \longrightarrow [M' \parallel N]}$$

Por otra parte, cada uno de los conjuntos de esquemas de axioma $[\beta_{sec}]$, $[\rho_1]$, ..., $[\beta_P]$ puede generarse mediante un sistema de inferencia asociado a cada conjunto de axiomas original β_{sec}, ρ_1, ..., β_P:

$$\begin{aligned}&SI_{[\beta_{sec}]} = \langle \; \beta_{sec}, \mu_{Struct} \; \rangle \\ &SI_{[\rho_1]} = \langle \; \rho_1, \mu_{Struct} \; \rangle \\ &\qquad \ldots \\ &\qquad \ldots \\ &SI_{[\beta_P]} = \langle \; \beta_P, \mu_{Struct} \; \rangle\end{aligned}$$

Y, por último:

$$[\mathcal{R}_{\mathcal{M}}] = [\mu_{\circ_1}] \cup [\mu_{\circ_2}] \cup [\mu_{||}] \cup [\mu_{\lambda}] \cup [\mu_{\frown}]$$

define el conjunto de (esquemas de) reglas de inferencia del sistema de inferencia.

Capítulo 5

LCEP como máquina abstracta para ALEPH

En este capítulo se aborda la tarea de tranformar las sentencias de un lenguaje de programación de más alto nivel (ver apéndices) en términos del formalismo LCEP. El objetivo final es disponer de una máquina abstracta capaz de ejecutar un programa de alto nivel empleando los recursos formales que brinda LCEP. En [LO96], se presenta una traducción de las estructuras básicas de un lenguaje funcional a LCEP. De este modo, comenzamos una nueva línea de trabajo en la que vamos a plantear el uso de LCEP desde la perspectiva de un lenguaje funcional.

5.1 Representación de las construcciones del lenguaje

5.1.1 Planteamiento. Modelo computacional y declaraciones

Las entidades fundamentales del formalismo LCEP son los *procesos*. Las relaciones entre procesos paralelos están basadas en la noción de *comunicación* a través de *canales* (extendida a la comunicación a través de *túneles*). Los procesos participantes están constituidos, a su vez, por procesos más simples construidos mediante los operadores algebraicos presentados. El modelo computacional que surge de manera natural es el de *procesos y comunicación*. Sin embargo, el modelo computacional asociado a un lenguaje imperativo como ALEPH (ver Apéndice B) se encuadra en el esquema de *manipulación de datos*. Por tanto, necesitamos realizar una adaptación entre ambos esquemas, que permita la utilización de LCEP como base de una máquina abstracta para ejecutar programas de más alto nivel.

Manipulación de datos frente a procesos y comunicación

En los apéndices se puede ver que las declaraciones de objetos realizadas como parte de un programa determinan el dispositivo de almacenamiento de datos sobre el cual actúan las sentencias del programa. Estamos, por tanto, representando la memoria de la máquina abstracta. Los nombres de las variables que declaramos permiten acceder a los objetos almacenados, cumpliendo así el papel de "dirección" de dicho objeto en memoria. Desde el punto de vista del modelo de procesos y comunicación, se considera que la memoria es un proceso que proporciona, mediante una operación de comunicación, un objeto en función de la dirección indicada.

Otro enfoque alternativo considera que cada dirección de memoria asociada a un proceso, es capaz de entregar el objeto almacenado al establecer la comunicación. La dirección de memoria representa el canal a través del cual se intercambia dicho objeto. Esta es la aproximación que vamos a adoptar para trasladar la memoria abstracta manipulada por el programa al mundo de procesos de la máquina LCEP.

Una vez realizada esta elección, veamos cómo se puede concretar, empleando los recursos expresivos de LCEP, la representación de las declaraciones. De los objetos descritos en la fase de declaraciones de un programa, nos interesan las variables y los subprogramas. Observamos que ambos pueden considerarse residentes en la memoria abstracta de la máquina a condición de que dispongamos de una forma satisfactoria para representarlos. En el caso de las variables, el objeto correspondiente es un término constante. En el caso de un subprograma, el objeto almacenado por la memoria es un término capaz de ser parametrizado y ejecutado por la máquina abstracta. El programa necesita acceder a la memoria para obtener el valor asociado a una variable cuando ésta participe en el cálculo de una expresión o en una sentencia de asignación. Se necesita acceder a la memoria para obtener el "valor" asociado a un subprograma cuando se ejecuta una sentencia de llamada a procedimiento o cuando se evalúa una expresión que involucra funciones. LCEP permite representar ambas situaciones mediante construcciones del lenguaje. Un programa consta de una lista de declaraciones de variables y subprogramas, y un "punto de entrada" o sentencia inicial que indica a la máquina abstracta por dónde debe empezar la ejecución. A la vista de esta organización de un programa, podemos plantear cómo definir los procesos LCEP correspondientes:

a) La memoria abstracta de la máquina se representa asociando a cada objeto con identificador Id declarado en el programa un proceso $!_{\widehat{Id}}\ M_{Id}$. Analicemos su estructura. La celda de memoria abstracta viene representada por el proceso $!_{\widehat{Id}}\ M_{Id}$, donde:

- Puede accederse al objeto mediante comunicación sobre el canal Id (Id es la dirección simbólica del objeto).

- El objeto "contenido" en la celda de memoria es el proceso M_{Id}. Según el tipo de objeto que contenga la celda de memoria (dato o subprograma) el proceso M_{Id} tiene una forma u otra, que luego veremos.

- Se trata de un proceso replicado; por tanto, en virtud de la propiedad $\pi_{!1}$, podemos acceder a la celda de memoria tantas veces como lo necesitemos. Esto permite dar cuenta del carácter estático (almacenamiento) de la información que contiene la memoria.

Definición 5.1.1 (m-proceso) *Un proceso de la forma $A = !\widehat{_P}\ M$ se denomina m-proceso por ser el empleado como representante de una celda de memoria abstracta. Observamos que un m-proceso es una replicación de un proceso comunicante (más aún, de un emisor elemental sobre P). Por extensión, decimos que $\Gamma(A) = \Gamma^-(A) = \{P\}$ y $\Gamma^+(A) = \emptyset$. A P, único canal soporte del proceso A sobre el cual se realizan las comunicaciones en un m-proceso, se le llama* dirección *del m-proceso. La dirección de un m-proceso A se denota como $d(A)$. En este caso, $P = d(A)$. Al proceso M se le denomina* contenido *del m-proceso.*

b) La sentencia que constituye el punto de entrada al programa de alto nivel consiste en un conjunto de asignaciones, llamadas a procedimiento, condicionales y bucles enlazados, agrupados mediante constructores de sentencia secuenciales y paralelos. Cada sentencia se trata como un proceso. Los constructores de sentencias sirven para expresar, mediante constructores de proceso, qué relación tienen los diferentes procesos entre sí. Al proceso asociado al punto de entrada se le llama s-proceso.

Los m-procesos obtenidos al considerar las declaraciones del programa se componen en paralelo. Todos ellos se componen en paralelo con el s-proceso del programa:

1. Una declaración D_i, $0 \leq i \leq d$, de variable o de subprograma, genera un m-proceso M_{m_i}.

2. El s-proceso del programa se denomina M_S.

El proceso generado por el programa completo, por lo tanto, es:

$$M_{\mathcal{H}} = M_{m_1} \parallel M_{m_2} \parallel \ldots \parallel M_{m_d} \parallel M_S$$

El s-proceso se construye de manera que obtiene, mediante $\mathcal{T}$-comunicación, los objetos necesarios para su ejecución (datos y subprogramas). Por este motivo se realiza la composición paralela de los m-procesos del programa con el s-proceso, que tiende un túnel entre los m-procesos y el s-proceso. Al m-proceso construido asociando varios m-procesos y un s-proceso en paralelo se le denomina l-proceso. El prefijo l dado a los l-procesos se justifica porque son los procesos que representan un programa del $l$enguaje de programación.

Las funciones y procedimientos primitivos se representan como m-procesos y todas ellas compuestas en paralelo definen el proceso que representa las instrucciones elementales del lenguaje de programación empleado. Los llamamos *procesos del sistema* porque representan la arquitectura de la máquina abstracta que ejecuta el programa:

$$M_\varphi \;=\; M_{\varphi_1} \parallel M_{\varphi_2} \parallel \ldots \parallel M_{\varphi_n}$$

El proceso que ejecuta la máquina abstracta LCEP es, por tanto:

$$\mathcal{M} \;=\; M_\varphi \parallel M_{\mathcal{H}}$$

Vemos que $\mathcal{M}$ es también un l-proceso.

LCEP permite expresar una jerarquía de acceso a los objetos manipulados. Para ello empleamos la noción de túnel junto con la de filtro, para establecer las relaciones adecuadas de $\mathcal{T}$-comunicabilidad entre los procesos que se ejecutan y los m-procesos derivados de las declaraciones de objetos (tanto globales como locales) realizadas en el programa. Para explicar cómo alcanzar este objetivo, vamos a introducir algunos conceptos.

La sup-adyacencia de una ocurrencia es el conjunto de aquellas ocurrencias inmediatamente superiores a la dada.

Definición 5.1.2 (Sup-adyacencia) *Sea $t \in T_\Sigma(V)$ y $u \in O(t)$. Se define la sup-adyacencia $u_+ \subseteq O(t)$ de la ocurrencia u como sigue:*

$$u_+ = \{v \in O(t) : \; \exists i \in I\!N - \{0\}. \;\; v = u.i\}$$

Se define la sup-adyacencia $U_+ \subseteq O(t)$ del conjunto de ocurrencias U como sigue:

$$U_+ \;=\; \bigcup_{u \in U} u_+ - U$$

Proposición 5.1.1 (Sup-adyacencia y orden de ocurrencias) *Las ocurrencias pertenecientes a la sup-adyacencia de una ocurrencia u son estrictamente superiores a la ocurrencia u; es decir, $\forall v \in u_+ \; u \lhd v$.*

Demostración.

Inmediato.

□

En lo que sigue, empleamos la siguiente notación:

$$u \lhd_+ v \quad \Longleftrightarrow \quad v \in u_+$$

Definición 5.1.3 (Camino elemental contiguo) *Sea $A \in \mathcal{M}$ un proceso. Sea M/u un subproceso de A. Un camino elemental $\gamma = n_1 n_2 \cdots n_k$ en el árbol sintáctico simétrico $\mathcal{A}^\circ(A)$ es contiguo al proceso M si $n_1 \lhd_+ u \ \vee \ n_k \lhd_+ u$.*

Definición 5.1.4 ($\mathcal{T}$-alcance de un m-proceso) *Sea A un proceso, el subproceso $M = A/u$ un m-proceso con dirección P, es decir, $M = !_{\widehat{P}}\ M'$, $\mathcal{T}_P(M)$ el conjunto de túneles P-transparentes contiguos al m-proceso M, $\Omega_P(M)$ el conjunto de ocurrencias que componen dichos túneles, y $\Omega_P(M)_+$ la sup-adyacencia de $\Omega_P(M)$. Se define el $\mathcal{T}$-alcance de M como el conjunto:*

$$\Phi_M = \Omega_P(M)_+ - \{u\}$$

Decimos que un subproceso $N = A/v$ se encuentra bajo el $\mathcal{T}$-alcance del m-proceso M si $v \in \Phi_M$.

Teorema 5.1.1 ($\mathcal{T}$-comunicabilidad) *Sea A un proceso, el subproceso $M = A/u$ un m-proceso con dirección P, y $\nu = min_{\unlhd}\{\nu_{\gamma_\mathcal{T}} \in \mathcal{T}_P(M)\}$. Un subproceso $N = A/v$ es $\mathcal{T}$-comunicable con M sobre el canal P si:*

1. *$\nu \in \Phi_M$, es decir, N está bajo el $\mathcal{T}$-alcance de M.*
2. *N es un proceso comunicante.*
3. *$P \in \Gamma^+(N)$, es decir, P pertenece al conjunto de canales de entrada de N.*
4. *$\forall w : \ w \unlhd \nu \Rightarrow \omega_A(w) \in \{\|, \circ\} \cup \Lambda \cup \mathcal{P}$.*

DEMOSTRACIÓN.

Si el subproceso M es un m-proceso con dirección P, por la Definición 5.1.1, tiene la forma $M = !\ \widehat{P}\ M'$. N es un proceso comunicante. Por la Definición 3.4.3, es de la forma $N \equiv N_1 + N_2 + \ldots + N_n, \quad n \geq 1$, donde algún N_i es un proceso comunicante elemental. Como $P \in \Gamma^+(N)$, es decir, pertenece al conjunto de canales de entrada de N, entonces existe algún N_i que tiene la forma $\lambda_P x.N_i'$. Por la condición 4) P define un túnel P-transparente. Por tanto, se dan las condiciones para que el subproceso N sea $\mathcal{T}$-comunicable con M sobre el canal P.

□

Ejemplo 10 *Sea el siguiente proceso:*

$$A = !\widehat{F}\ A' \parallel \widehat{_{p_1}}(\widehat{_q}(\lambda_F x.M_2)) \parallel \widehat{_q}(!\widehat{F}\ A'' \parallel \lambda_F x.M_3)$$

En la Figura 5.1 mostramos el árbol sintáctico asociado al proceso A y sobre él se pueden detectar los diferentes túneles entre los subprocesos.

ε $\parallel$ $\mathcal{T}_F(A_1)$ 1 2 A_1 $!$ $\widehat{F}$ A'

Este tramo no es F-transparente

2.1 $\widehat{p_1}$ 2.2 $\widehat{q}$ 2.1.1 2.2.1 $\mathcal{T}_F(A_2)$

Este tramo no es F-transparente

2.1.1.1 $\widehat{q}$ $\widehat{p_2}$ $!$ $\lambda_F x$ 2.1.1.2 2.2.1.1 2.2.1.2 2.1.1.1.1 $\lambda_F x$ 2.1.1.2.1 $\lambda_F x$ $\widehat{F}$ M_3 M_1 M_2 A''

Figura 5.1: Arbol sintáctico del proceso A

- *Suponemos que $F \succeq p_i$ para $i = 1, 2$ y que F no es comparable con q.*
- *Se han etiquetado algunos nodos con la ocurrencia que les corresponde.*
- *Se distinguen los dos m-procesos presentes como A_1 y A_2, respectivamente, ambos de dirección F.*
- *Aparecen delimitados los conjuntos de túneles F-transparentes para ambos m-procesos.*

Con estas condiciones, podemos escribir:

$$\Phi_{A_1} = \{2.2, 2.1.1.1, 2.1.1.2.1\} \qquad \Phi_{A_2} = \{2.2.1.2\}$$

Advertimos que es posible la existencia de dos m-procesos con la misma dirección y alcances disjuntos. A_1 y A_2 son ejemplo de tales m-procesos. Observemos que, aunque tienen la misma dirección, no tienen el mismo contenido. Hemos empleado el q-filtro $\widehat{q}(\ !\widehat{F}\ A'' \parallel \lambda_F x.M_3)$ para llevar a cabo el aislamiento de su interior respecto de las comunicaciones externas a través del canal F.

De lo visto, podemos concluir que:

1. La representación de las sentencias básicas del lenguaje de programación debe apoyarse en el concepto de receptor elemental que, mediante una operación de comunicación sobre una determinada dirección, accede al contenido de los m-procesos requeridos (acceso a la memoria abstracta).

2. Los nombres de los objetos (direcciones simbólicas) se tratan como canales simbólicos ($P \in S \subset \mathcal{L}$) LCEP; por tanto, las variables del lenguaje de programación no se corresponden semánticamente a variables del formalismo. Las variables del lenguaje de programación expresan operaciones de comunicación entre los procesos que representan la memoria abstracta (m-procesos) y los subprocesos de M_S.

3. La noción de alcance de los objetos declarados en el programa adquiere realidad física en el árbol sintáctico del proceso bajo la forma del conjunto de ocurrencias $\Phi_{M_{m_i}}$ que define el $\mathcal{T}$-alcance del m-proceso M_{m_i}, obtenido de una declaración D_i del programa.

4. El alcance de los objetos es equivalente al $\mathcal{T}$-alcance de los m-procesos correspondientes, a condición de:

   - definir adecuadamente la geometría del árbol sintáctico del proceso $M_{\mathcal{N}}$, situando en él los filtros necesarios para definir los límites de los túneles adyacentes a cada m-proceso, y
   - una elección adecuada de la relación de orden establecida entre las direcciones de los m-procesos; es decir, las declaraciones en el programa determinan la relación de orden $\preceq_{\mathcal{L}}$.

Aproximación al modelo de ejecución de programas

Consideramos un programa como un proceso único, el proceso $\mathcal{M}$, que, de acuerdo con la definición inductiva de los λ_{EP}-términos, puede estar constituido a su vez por otros procesos relacionados mediante los operadores de construcción de proceso. Existen cierto número de procesos básicos que se corresponden con las constantes del cálculo formal (y combinaciones de ellas mediante constructores) que nos van a permitir expresar instancias de las instrucciones elementales de la máquina abstracta sobre la que ejecutamos nuestros programas. Como veremos, los procesos M_{ϕ_i} que representan dichas instrucciones elementales están definidos en términos de constantes del formalismo. Un subprograma puede considerarse como una abstracción que representa un conjunto de instrucciones elementales ordenadas adecuadamente y cuya forma concreta depende de la parametrización realizada cada vez que dicho subprograma es activado. Podemos plantear la ejecución de un proceso como un mecanismo

de búsqueda de las instrucciones básicas que deben ejecutarse. Los mecanismos formales de LCEP nos permiten soportar directamente las nociones de aplicación de una función sobre sus argumentos y la comunicación entre procesos. Sin embargo, en una ejecución real necesitamos calcular una suma, obtener un carácter, ... El contenido de las instrucciones elementales no es directamente representable en el formalismo. Esto sucede en todos los lenguajes de programación. Es necesario acudir a una máquina "real" (un procesador matemático, por ejemplo) para hacer dichos cálculos. Por ello introducimos los efectos laterales necesarios para llevar a cabo computaciones reales mediante un conjunto de reglas que transforman procesos básicos (indicados como combinaciones de constantes) que expresan una instancia concreta de una instrucción elemental en otros procesos básicos que representan el resultado de la ejecución de dicha instrucción elemental.

Para diferenciar ambos componentes del concepto de ejecución de un programa, distinguimos:

- *Reducción*, es decir, la aplicación directa de los axiomas de reordenación y la β-comunicación secuencial y paralela.

- $\mathcal{H}$-*evaluación*, que corresponde al nivel inferior de la máquina abstracta, donde se produce la interpretación de los símbolos de la base de inducción del lenguaje como instrucciones elementales de la máquina abstracta definida por el lenguaje de programación.

La reducción captura el componente más abstracto del formalismo, aquél que se refiere a las relaciones de los procesos entre sí. La $\mathcal{H}$-evaluación nos permite conectar el formalismo a un universo concreto, definido por un conjunto de acciones básicas sobre la máquina donde residen los procesos.

La reducción de términos del lenguaje se lleva a cabo mediante los axiomas de β-comunicación secuencial y paralela así como los axiomas de reordenación y demás propiedades algebraicas de los constructores de proceso. El mecanismo de reducción contribuye a la ejecución de un proceso, representado por un λ_{EP}-término, reduciendo su complejidad sintáctica, acercándolo a la forma normal en la cual, por definición, ya no son posibles más reducciones. Para ello, la máquina abstracta debe llevar a cabo un proceso de *análisis* del término con el fin de:

1. Identificar los redexes existentes.

2. Seleccionar el redex más adecuado.

Una vez realizado esto, se produce la reducción correspondiente y se reconstruye el proceso. Esta reconstrucción constituye el procedimiento contrario al análisis realizado por el compilador, y consiste en intercambiar, en la estructura sintáctica de M, el

subtérmino reducido N por el reducto N'. De aquí surge M' como reducto de M. Se ha completado un paso de reducción.

La $\mathcal{H}$-evaluación puede contemplarse también como una operación de reescritura de términos del lenguaje, en particular, de términos que expresan aplicaciones de función sobre argumentos constantes, a través de la cual continuamos "simplificando" el proceso inicial. Cuando el análisis de un término no conduce a la detección de ningún redex, hemos llegado a una forma normal para el término. En ese momento, podemos plantear un nuevo proceso de análisis que localice y seleccione subtérminos susceptibles de ser $\mathcal{H}$-evaluados, es decir, que sean equivalentes a un término más simple (estructuralmente) que el original.

La ejecución de un proceso queda pues definida por una combinación de las operaciones de reducción y cálculo. Consiste en la transformación de la λ_{EP}-expresión asociada, mediante reducciones y $\mathcal{H}$-evaluaciones, hasta obtener un término irreducible y sobre el cual tampoco podemos aplicar reglas de $\mathcal{H}$-evaluación. Decimos que un proceso se ha ejecutado cuando ha alcanzado dicha forma normal general respecto al conjunto de reglas de reducción y $\mathcal{H}$-evaluación.

5.1.2 Subprogramas

A continuación, estudiamos la problemática asociada a los subprogramas desde el punto de vista de su tratamiento en el contexto formal de LCEP.

Tratamiento de la activación de subprogramas

Al emplear el esquema de evaluación de llamada por valor CBV (ver apéndices) debemos entregar a las funciones argumentos completamente evaluados. Esto se representa, en nuestro sistema computacional orientado a procesos, considerando que los argumentos de una función son a su vez procesos que deben ser ejecutados antes que el proceso que representa la llamada a una función. La dependencia (orden de ejecución) se establece mediante operaciones de comunicación privadas para cada invocación a un subprograma. Se trata de aplicar un procedimiento de descomposición del problema de ejecutar un subprograma F, estableciendo una serie de canales privados, representados con σ_i, para cada expresión E_i que constituye un parámetro real para F. Dichos canales privados cumplen una doble tarea:

- Permiten enviar el resultado de la evaluación del parámetro real donde se necesita, que es en el proceso donde se aplica la función F a sus parámetros ya evaluados.

- Constituyen un filtro para el proceso que representa la expresión que se evalúa. Dicho filtro debe ser transparente para los m-procesos de alcance global. De

esta manera, el proceso E_i sobre el que se aplica el operador $\widehat{\sigma_i}$ puede acceder a las variables y subprogramas globales que necesita para su evaluación. Sin embargo, el operador debe aislar al proceso de comunicaciones no globales (por ejemplo, sobre otros canales σ_j asociados a la evaluación de otras llamadas a subprograma o sobre canales que representan direcciones de m-procesos locales).

La idea de aplicar un filtro sobre un subproceso para ejecutar o evaluar dicho subproceso es muy importante en el tratamiento computacional de las construcciones del lenguaje de alto nivel. Por eso decimos que el filtro $\widehat{\sigma_i}\ E_i$ es el *entorno de ejecución* del subproceso E_i. El proceso E_i permanece en el entorno de ejecución hasta que ya no puedan aplicarse dentro de él más reglas de reducción ni de $\mathcal{H}$-evaluación. Constituye el “lugar” de ejecución local del subproceso E_i. Volviendo a la descripción del mecanismo de activación, se llevan a ejecución los n argumentos de la función, creando un entorno de ejecución para cada uno de ellos y comunicando su resultado final a través del canal σ_i establecido para cada argumento E_i. Tras recibir los argumentos ya evaluados, se aplica sobre ellos la definición de la función F, que es el contenido de un m-proceso $\mathcal{T}$-comunicado con ellos a través de la dirección F del m-proceso.

La descripción de la ejecución descansa sobre la idea de comunicación a través de canales. El orden de ejecución de los subprogramas viene determinado por la generación adecuada de las operaciones de comunicación necesarias, así como por la elección de una estrategia apropiada para la aplicación de las reglas de reducción para la ejecución del proceso.

Puesto que un argumento para una función puede ser cualquier expresión, la traducción debe reflejar la estructura inductiva de las expresiones, teniendo su caso base en la ocurrencia de un valor constante, o en una ocurrencia de variable, cuyo valor asociado puede obtenerse como resultado directo de una comunicación con el m-proceso asociado a dicha variable.

En los ejemplos siguientes, suponemos que se trabaja con una función F de dos parámetros formales x e y, $F(x,y)$. El significado de esta función es la *resta*[1] de sus dos argumentos.

Ejemplo 11 *Deseamos evaluar la llamada a subprograma de $F(1,2)$. Entonces, el proceso asociado es el siguiente:*

$$\begin{aligned} \mathcal{M}_{F(1,2)} = \ & \lambda_{\sigma_1}\sigma_1'.\lambda_{\sigma_2}\sigma_2'.\lambda_F F' \circ \widehat{1}\ \sigma_1' \circ \widehat{1}\ \sigma_2' \ \| \ \widehat{\sigma_1}\ 1 \ \| \\ & \| \ \widehat{\sigma_2}\ 2 \ \| \ \widehat{F}(\lambda_1 x_1'.\lambda_1 x_2'.x_1' - x_2') \end{aligned}$$

Es interesante advertir que:

[1]Suponemos que la operación *resta* forma parte de las instrucciones elementales del lenguaje de programación.

- *El cuerpo de la función F se ha codificado pensando únicamente en la instanciación posicional de los parámetros. Para ello es suficiente el empleo de la etiqueta numérica* 1 *[AHU74, ASU90].*

- *La aplicación de la función F a sus argumentos se expresa, análogamente a como se hace en el λ-cálculo, mediante la composición secuencial del proceso que describe la abstracción de la función (el contenido del m-proceso generado por la declaración de dicha función) y los distintos procesos que representan la asociación de cada parámetro real (ya evaluado) a cada parámetro formal. Dicha asociación se expresa mediante canales.*

Ejemplo 12 *Deseamos evaluar la llamada a subprograma $F(u, 2)$. Suponemos que a la variable (global) u se le ha asociado el valor* 10*. Entonces, el proceso asociado a esa llamada es el siguiente:*

$$\begin{aligned} \mathcal{M}_{F(u,2)} \;=\;& \lambda_{\sigma_1}\sigma_1'.\lambda_{\sigma_2}\sigma_2'.\lambda_F F' \circ \widehat{1}\ \sigma_1' \circ \widehat{1}\ \sigma_2' \;\|\; \lambda_u u'.(\widehat{\sigma_1}\ u') \;\|\; \widehat{\sigma_2}\ 2 \;\| \\ & \|\; \widehat{F}(\lambda_1 x_1'.\lambda_1 x_2'.x_1' - x_2') \;\|\; \widehat{u}\ 10 \end{aligned}$$

Podemos advertir cómo va a llevarse a cabo el tratamiento recursivo de la ejecución de subprogramas. Como consecuencia de la existencia de la variable global u, aparece un nuevo m-proceso con dirección u y contenido 10*. La evaluación del primer parámetro para F requiere ahora de una operación de comunicación previa que obtiene el contenido de la variable u. Una vez hecho esto, se puede enviar al proceso que evalúa la llamada $F(u, 2)$.*

Es importante hacer notar que:

- *La necesidad de obtener el valor asociado a la variable u mediante una operación de $\mathcal{T}$-comunicación obstaculiza la comunicación a través del canal σ_1 asociado a la evaluación del primer parámetro, pues se corta el túnel establecido con el proceso que evalúa $F(u, 2)$. Esto es lo que deseamos, ya que desde el punto de vista semántico, no tiene ningún sentido enviar algo a través de σ_1 a menos que sea el contenido de la variable u. Este es un ejemplo de cómo la estructura dada a los procesos obtenidos al traducir las sentencias del lenguaje apoya la simulación del proceso que se obtiene del mecanismo de evaluación de una función.*

- *En el caso de que el segundo parámetro sea una variable (por ejemplo, la misma), la evaluación de ambos parámetros reales puede realizarse en paralelo, puesto que la estructura dada al proceso permite la $\mathcal{T}$-comunicación del proceso que evalúa el segundo parámetro.*

```
F(G(1),2)
```

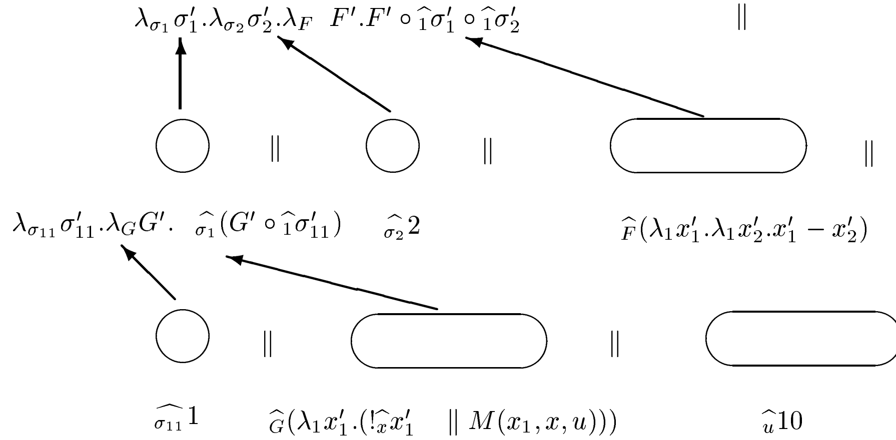


Figura 5.2: Comunicaciones en la llamada a una función

Ejemplo 13 *Por último, planteamos el caso en que uno de los parámetros es una llamada a una función con un parámetro:* $F(G(1),2)$*. Además, suponemos que la función* G *incluye una variable local* x *y que hace uso de la variable global* u*; es decir,*

> **def** $G(x_1)$ =
>   **local** x ;
> $S(x_1, x, u)$

La Figura 5.2 muestra el esquema de comunicaciones que deben realizarse para lograr la evaluación de la llamada a función. En la traducción de la definición de G *se asume que la variable local se inicializa con el valor del parámetro de la función.*

A la vista de la Figura 5.2, observamos que la entrega de la instancia del primer parámetro a la función F *depende de la evaluación de una llamada a la función* $G(1)$*. Se ha definido el s-proceso interno a la definición de* G *como* $M(x_1, x, u)$*, indicando que el cuerpo de* G *va a hacer uso tanto del parámetro real pasado como de la variable local* x *y de la variable global* u*.*

En un momento de la ejecución, una vez instanciada G *con el parámetro real* 1*, tenemos una situación como la mostrada en la Figura 5.3.*

Tal y como puede observarse, el cálculo del valor asociado a $G(1)$ *mantiene una dependencia del valor de la variable* u*. Esto ilustra el problema de la necesidad de que el* σ_1*-entorno (*σ_1*-filtro) definido sea* u*-transparente. También podemos decir que el entorno no debe ser* x*-transparente, dado que* x *es una variable local y como tal sólo debe ser accesible al proceso* $M(x_1, x, u)$*. Otras funciones pueden declarar variables locales con el mismo nombre o, incluso, existir variables globales con el mismo*

`F(G(1),2)`

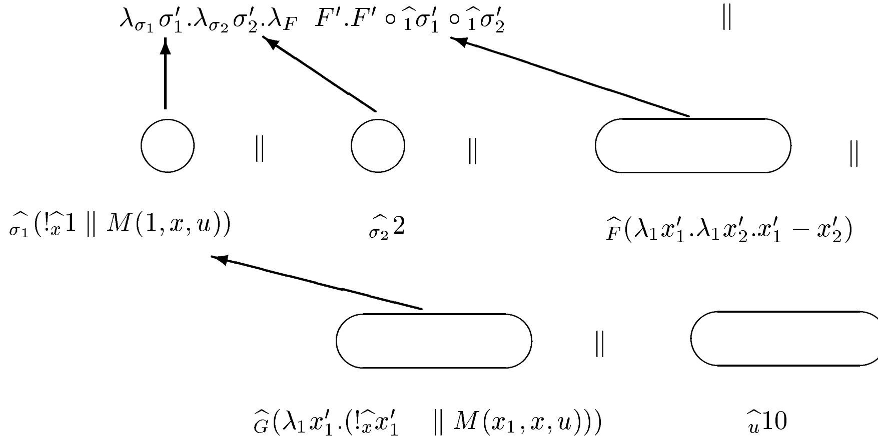


Figura 5.3: Comunicaciones tras la instanciación de G

identificador. En cuanto a la capacidad de las variables locales de ocultar la presencia de variables globales de la misma dirección (mismo nombre), puede incorporarse mediante el algoritmo de ejecución o estrategia de reducción, definiendo adecuadamente la noción de redex y dando mayor prioridad a las comunicaciones más internas.

En general, podemos decir que los σ-entornos deben ser transparentes a las direcciones de todos los procesos globales (variables, subprogramas y procesos que representan las instrucciones elementales del lenguaje de programación) y no transparentes a los restantes canales de comunicación. El control sobre las características de transparencia de los entornos de ejecución lo proporciona la relación de orden parcial $\preceq_{\mathcal{L}}$ dada en función de los distintos elementos del lenguaje de programación.

5.1.3 Estructura de los procesos asociados a subprogramas

Una declaración de subprograma genera un m-proceso cuya dirección coincide con el nombre de éste en la declaración. El contenido del m-proceso debe ser capaz de soportar la funcionalidad esperada de un subprograma, es decir: su interfaz (parámetros formales), su implementación (datos locales y estructuras de control) y el mecanismo de activación. Vamos a estudiar cómo implementar estas características en LCEP. Damos soporte, con algunas restricciones, a la parametrización total y parcial de funciones y a la instanciación posicional y etiquetada, tanto simbólica como numérica.

Una secuencia de parámetros instanciada posicionalmente se traduce asignando la etiqueta numérica 1 como canal a cada instancia de parámetro real. Una secuencia de parámetros instanciada empleando etiquetado simbólico utiliza como canal simbólico

una etiqueta simbólica obtenida por composición del nombre de la función y el nombre del parámetro formal [AHU74, ASU90].

Para el subprograma ejemplo, tenemos:

Tipo de parametrización	**Sintaxis del lenguaje para la activación**	**Proceso que aplica F**
`Instanciación posicional`	$F(a,b)$	$F' \circ_1 \widehat{a} \circ_1 \widehat{b}$
`Instanciación por etiquetado numérico`	$F(b{-}{>}\,2, a{-}{>}\,1)$	$F' \circ_2 \widehat{b} \circ_1 \widehat{a}$
`Instanciación por etiquetado simbólico`	$F(b{-}{>}\,x_2, a{-}{>}\,x_1)$	$F' \circ_{F_{x_2}} \widehat{b} \circ_{F_{x_1}} \widehat{a}$

Y, en general, asumiendo que P_i es el resultado de evaluar la expresión E_i y que $0 \leq m \leq n$ (se contempla la instanciación parcial), tenemos que:

$$\begin{array}{l} \textbf{def}\ \ F(x_1, x_2, \ldots, x_n) = \\ \qquad \textbf{local}\ \ x, y, \ldots, z; \\ \qquad S(x_1, x_2, \ldots, x_n, \ldots) \end{array}$$

Tipo de parametrización	**Sintaxis del lenguaje para la activación**
`Instanciación posicional`	$F(E_1, E_2, \ldots, E_m)$
`Instanciación por etiquetado numérico`	$F(E_1{-}{>}\,j_1, E_2{-}{>}\,j_2, \ldots, E_m{-}{>}\,j_m)$
`Instanciación por etiquetado simbólico`	$F(E_1{-}{>}\,x_{j_1}, E_2{-}{>}\,x_{j_2}, \ldots, E_m{-}{>}\,x_{j_m})$

Tipo de parametrización	**Proceso que aplica F**
`Instanciación posicional`	$F' \circ_1 \widehat{P_1} \circ_1 \widehat{P_2} \cdots \circ_1 \widehat{P_m}$
`Instanciación por etiquetado numérico`	$F' \circ_{n_1} \widehat{P_1} \circ_{n_2} \widehat{P_2} \cdots \circ_{n_m} \widehat{P_m}$
`Instanciación por etiquetado simbólico`	$F' \circ_{F_{x_{j_1}}} \widehat{P_1} \circ_{F_{x_{j_2}}} \widehat{P_2} \cdots \circ_{F_{x_{j_m}}} \widehat{P_m}$

Cada etiqueta numérica n_i de la instanciación por etiquetado numérico, $1 \leq i \leq m$ se calcula como sigue:

$$n_i = o(j_i) \qquad \text{donde } o(j_i) = j_i - card\{k : j_k < j_i \wedge 1 \leq k < i\}$$

El contenido del m-proceso asociado a un subprograma refleja la necesidad de dar soporte a estos tres tipos de instanciación.

Representamos el contenido del m-proceso generado por la declaración de F como:

$$\Lambda(F, x_1, x_2, \ldots, x_n, M_S(x_1, x_2, \ldots, x_n, x, y, \ldots, z, u, v, \ldots, w)).$$

La notación empleada expresa que el proceso tiene carácter de abstracción (Λ) y que su descripción completa se da en términos del nombre del subprograma F, de sus parámetros formales $x_1, x_2, \ldots, x_n$ y del l-proceso , que denotamos como:

$$M_S(x_1, x_2, \ldots, x_n, x, y, \ldots, z, u, v, \ldots, w).$$

El proceso puede manipular los parámetros formales como variables locales de nombre $x_1, x_2, \ldots, x_n$, las variables locales $x, y, \ldots, z$ definidas en la declaración, así como las variables globales $u, v, \ldots, w$ especificadas como tales en las declaraciones globales. Llamamos a este proceso la *abstracción* del subprograma F.

El proceso se define inductivamente como:

$$\begin{array}{l}\Lambda(F, x_i, x_{i+1}, \ldots, x_n, M_S(x_1, x_2, \ldots, x_n, x, y, \ldots, z, u, v, \ldots, w)) = \\ \left\{\begin{array}{ll} \left(\begin{array}{l} \lambda_1 x_i'.\Lambda(F, x_{i+1}, \ldots, x_n, \\ M_S(x_1, x_2, \ldots, x_n, x, y, \ldots, z, u, v, \ldots, w))+ \\ \lambda_{F_{x_i}} x_i'.\Lambda(F, x_{i+1}, \ldots, x_n, \\ M_S(x_1, x_2, \ldots, x_n, x, y, \ldots, z, u, v, \ldots, w)) \end{array}\right) & \text{si } i \leq n \\ M_S(x_1, x_2, \ldots, x_n, x, y, \ldots, z, u, v, \ldots, w)) & \text{otro caso} \end{array}\right.\end{array}$$

La idea que pretende capturar esta definición es la de que el proceso es capaz de enfrentarse a cualquiera de los tipos de instanciación vistos.

La estructura del l-proceso $M_S(x_1, x_2, \ldots, x_n, x, y, \ldots, z, u, v, \ldots, w)$ es la siguiente:

$$\begin{array}{l} M_S(x_1, x_2, \ldots, x_n, x, y, \ldots, z, u, v, \ldots, w) = \\ \left(\begin{array}{l} !\widehat{_{x_1}}\ x_1' \ \| \ !\widehat{_{x_2}}\ x_2' \ \| \ \cdots \ \| \ !\widehat{_{x_n}}\ x_n' \ \| \ !\widehat{_{x}}\ 0 \ \| \ !\widehat{_{y}}\ 0 \ \| \ \cdots \ \| \ !\widehat{_{z}}\ 0 \ \| \\ \| \ \ \tau[S(x_1, x_2, \ldots, x_n, x, y, \ldots, z, u, v, \ldots, w)] \end{array}\right) \end{array}$$

Como puede observarse, a cada parámetro formal se le asigna un m-proceso que resulta de inicializar el valor del parámetro real que se pasa a la abstracción a través del canal correspondiente. Las variables locales son asignadas al proceso nulo. El problema de diferenciar entre funciones y procedimientos introduce pequeñas modificaciones en la definición, que veremos posteriormente. La modificación del contenido de un m-proceso que representa una variable se va a tratar como una instrucción elemental del lenguaje. Por último, el s-proceso asociado al cuerpo del subprograma se obtiene mediante una función τ:

$$\tau : \mathcal{H} \rightarrow \mathcal{M}$$

que traduce las sentencias del lenguaje de más alto nivel - LMAN - (elementos del conjunto $\mathcal{H}$ de LMAN) a procesos LCEP. El l-proceso constituye el entorno de ejecución para el s-proceso asociado al cuerpo del subprograma.

Es interesante observar que la estructura del proceso que representa el subprograma es muy parecida a la estructura del l-proceso que representa el programa que

se ejecuta en el sistema. Un subprograma, por tanto, tiene la misma estructura, a excepción del interfaz, que un programa.

Para evaluar la adecuación de esta manera de construir la abstracción del subprograma, en cuanto al soporte de la parametrización, trabajamos con la abstracción del subprograma $F(x_1, x_2)$ declarado anteriormente:

$$\begin{aligned}\Lambda(F, x_1, x_2, M_S(x_1, x_2)) = &\\ &\lambda_1 x_1'.(\lambda_1 x_2'.M_S(x_1, x_2) \ + \ \lambda_{F_{x_2}} x_2'.M_S(x_1, x_2)) \ + \\ &\lambda_{F_{x_1}} x_1'.(\lambda_1 x_2'.M_S(x_1, x_2) \ + \ \lambda_{F_{x_2}} x_2'.M_S(x_1, x_2))\end{aligned}$$

Hemos supuesto que F es un procedimiento y que no emplea variables globales. Vamos ahora a plantear cómo se lleva a cabo la parametrización (total) del procedimiento, según los distintos esquemas, si sustituimos F' por la abstracción de F propuesta. Asumimos que a y b son constantes.

Ejemplo 14

Instanciación posicional: $F(b, a)$.
El proceso aplicativo es $F' \circ \widehat{1}\ b \circ \widehat{1}\ a$.

$$\left(\begin{array}{l} \lambda_1 x_1'.(\lambda_1 x_2'.M_S(x_1, x_2) + \lambda_{F_{x_2}} x_2'.M_S(x_1, x_2)) + \\ \lambda_{F_{x_1}} x_1'.(\lambda_1 x_2'.M_S(x_1, x_2) + \lambda_{F_{x_2}} x_2'.M_S(x_1, x_2)) \end{array} \right) \circ \widehat{1}\ b \circ \widehat{1}\ a$$

$$\begin{aligned}&\rightarrow_{\beta_{sec}} (\lambda_1 x_2'.M_S(b, x_2) + \lambda_{F_{x_2}} x_2'.M_S(b, x_2)) \circ \widehat{1}\ a \\ &\rightarrow_{\beta_{sec}} \ M_S(b, a)\end{aligned}$$

Recordemos que $M_S(x_1, x_2)$ *tiene una forma semejante a la mostrada anteriormente para el* l*-proceso general de un subprograma. Queda clara, por tanto, la ligadura de las variables* x_1' *y* x_2' *con los parámetros formales* x_1 *y* x_2.

Como puede observarse, el resultado obtenido es el correcto, el l*-proceso de la abstracción instanciado con los parámetros reales.*

Ejemplo 15

Instanciación etiquetada numérica: $F(a{-}{>}\ 2, b{-}{>}\ 1)$.
El proceso aplicativo es $F' \circ \widehat{2}\ a \circ \widehat{1}\ b$.

$$F' \circ \widehat{2}\ a \circ \widehat{1}\ b \ \rightarrow_{\eta_2} \ F' \circ \widehat{1}\ b \circ \widehat{1}\ a \ \equiv$$

$$\left(\begin{array}{l} \lambda_1 x_1'.(\lambda_1 x_2'.M_S(x_1, x_2) + \lambda_{F_{x_2}} x_2'.M_S(x_1, x_2)) + \\ \lambda_{F_{x_1}} x_1'.(\lambda_1 x_2'.M_S(x_1, x_2) + \lambda_{F_{x_2}} x_2'.M_S(x_1, x_2)) \end{array} \right) \circ \widehat{1}\ b \circ \widehat{1}\ a$$

$$\begin{aligned}&\rightarrow_{\beta_{sec}} (\lambda_1 x_2'.M_S(b, x_2) + \lambda_{F_{x_2}} x_2'.M_S(b, x_2)) \circ \widehat{1}\ a \\ &\rightarrow_{\beta_{sec}} \ M_S(b, a)\end{aligned}$$

Ejemplo 16
Instanciación etiquetada simbólica. $F(a{-}{>}\,x_2, b{-}{>}\,x_1)$.
El proceso aplicativo es $F' \circ \widehat{F_{x_2}}\ a \circ \widehat{F_{x_1}}\ b$.

Asociamos a las etiquetas construidas la siguiente relación de orden: $F_{x_1} \preceq F_{x_2}$.

$$F' \circ \widehat{F_{x_2}}\ a \circ \widehat{F_{x_1}}\ b\ \rightarrow_{\rho_2}\ F' \circ \widehat{F_{x_1}}\ b \circ \widehat{F_{x_2}}\ a\ \equiv$$

$$\left(\begin{array}{l} \lambda_1 x_1'.(\lambda_1 x_2'.M_S(x_1,x_2) + \lambda_{F_{x_2}} x_2'.M_S(x_1,x_2)) + \\ \lambda_{F_{x_1}} x_1'.(\lambda_1 x_2'.M_S(x_1,x_2) + \lambda_{F_{x_2}} x_2'.M_S(x_1,x_2)) \end{array}\right) \circ \widehat{F_{x_1}}\ b \circ \widehat{F_{x_2}}\ a$$

$$\rightarrow_{\beta_{sec}} (\lambda_1 x_2'.M_S(b,x_2) + \lambda_{F_{x_2}} x_2'.M_S(b,x_2)) \circ \widehat{F_{x_2}}\ a$$
$$\rightarrow_{\beta_{sec}}\ M_S(b,a)$$

5.1.4 Funciones primitivas del lenguaje

Para soportar un lenguaje de programación mediante los recursos expresivos de LCEP debemos ir definiendo construcciones lingüísticas de nivel superior, es decir, que pudiendo ser expresadas en términos de la gramática básica, expresen realidades del LMAN. De esta forma, introducimos constantes sobre las cuales mostrar los elementos típicos presentes en todos los lenguajes de programación, así como las construcciones de control de flujo y tipos de datos básicos.

Con relación a la gramática del lenguaje, exigimos que el conjunto C de constantes del formalismo contenga todos aquellos elementos que necesitamos para dar soporte a las diferentes características del LMAN. Debemos poseer:

- Valores numéricos. C incluirá un conjunto NUM de números.
- Cadenas alfanuméricas. C poseerá un conjunto CAD de cadenas alfanuméricas definidas sobre un cierto alfabeto. Pueden expresarse entre comillas dobles.
- Expresiones. Hacemos que C contenga un conjunto de operadores aritméticos $OP = \{+, -, /, \times, \ldots\}$, con una sintaxis adecuada para permitir su aplicación infija.
- Valores lógicos. C incluirá el conjunto $BOOL = \{True, False\}$ de valores lógicos.
- Expresiones booleanas. Para ello, C incluye un conjunto $OPB = \{\neg, \wedge, \vee, \ldots\}$ de operadores lógicos.
- Operadores relacionales. $OPR = \{=, <, >, \ldots\}$.

Por tanto, debe cumplirse que:

$$NUM\ \cup\ CAD\ \cup\ OP\ \cup\ BOOL\ \cup\ OPB\ \cup\ OPR\ \subseteq C$$

Las funciones primitivas φ_i, así como los diversos operadores (que se tratan como funciones), pueden ser descritos análogamente a las funciones definidas por el usuario. Pueden verse, desde el punto de vista del acceso a sus servicios, como canales siempre disponibles, que "transmiten" las abstracciones de dichas funciones (subprogramas en general) necesarias para implementarlas.

La cuestión verdaderamente importante en las funciones primitivas es la de su evaluación, que representa la conexión del programa con una máquina real. La evaluación puede introducirse en la máquina abstracta a través de reglas de evaluación análogas a las reglas de reducción. Por ejemplo, en el lenguaje de expresiones matemáticas utilizadas en aritmética, podemos definir la evaluación del seno de un número a través de una regla como la siguiente[2]:

$$seno(a) \rightarrow_{eval_{seno}} b \iff a \in \mathcal{R}, b = sin(a)$$

donde $\mathcal{R}$ representa el conjunto de los números reales.

El efecto de la regla es, como en las reglas de reducción del formalismo, un intercambio de términos del lenguaje. Podemos escribir el número real b (que es $sin(a)$) en lugar del término $seno(a)$ siempre que a sea una constante (es decir, que no sea un término complejo). En otro caso, el proceso de cálculo de $seno(E)$, siendo E una expresión, debe, primero, calcular el valor de E (evaluación CBV). Podemos expresar esta idea en nuestro cálculo empleando las construcciones sintácticas de LCEP. Para ello:

1. Suponemos definidos los <u>procesos del sistema</u> o funciones primitivas, de forma parecida a la que acabamos de ver:

   $$\textbf{def } seno(x) = !(\widehat{seno}(\lambda_1 x.SENO \circ x))$$

   La notación nos sirve para expresar que a las funciones primitivas se les asigna directamente un cuerpo perfectamente definido en términos de las constantes LCEP que representan dicha función primitiva. En este caso, hacemos que $SENO \in C$. Como puede observarse, el contenido del proceso que representa la función primitiva tiene una parte de interfaz que permite tratar las llamadas a la función *seno* como una función cualquiera desde el punto de vista de la evaluación de sus parámetros reales, parametrización, ... Lo verdaderamente nuevo es que el l-proceso M_S correspondiente puede darse ya en términos de construcciones muy simples del lenguaje.

2. Definimos tantas <u>reglas de evaluación</u> como necesitemos. Recordemos que las reglas deben considerarse como relaciones binarias en el conjunto $\mathcal{M}$ de

[2] sin representa a la función matemática y $seno$ a la expresión aritmética que define a la función definida seno.

λ_{EP}-términos, es decir, simplemente reescriben un λ_{EP}-término a otro, bajo la satisfacción de ciertas condiciones. Por ejemplo, para la función *seno* tenemos:

$$SENO \circ a \rightarrow_{eval_{seno}} b \ \Leftarrow \ a \in NUM, b = sin(a)$$

Es decir, necesitamos que se satisfaga la condición $a \in NUM$. La aplicación efectiva de la regla transforma $SENO \circ a$ en el número b, que corresponde al seno de a. Trabajando de esta manera, podemos interpretar las funciones del sistema como hacemos con cualquiera otra llamada de función, e introducirlas en el cálculo computacional de forma coherente. Este tipo de reglas constituyen las denominadas reglas de $\mathcal{H}$-evaluación.

5.1.5 Diferencias entre funciones y procedimientos

Hasta ahora hemos considerado los subprogramas de manera única sin distinguir apenas entre funciones y procedimientos. La diferencia fundamental entre ellos radica en que las funciones proporcionan un valor de retorno como resultado final de la computación, mientras que no ocurre eso con los procedimientos. Pero, incluso esta diferencia puede relajarse, excepto en lo que respecta a la participación de ambos en la construcción de expresiones, si consideramos que un procedimiento es un caso particular de función cuyo valor de retorno es un elemento especial, que denotamos con el símbolo $\Diamond$. Esta diferencia entre funciones y procedimientos se refleja en el l-proceso que representa las declaraciones locales y el cuerpo del subprograma. Añadimos un m-proceso asociado al valor de retorno del subprograma cuando éste es de tipo función. Dicha variable debe contener, al final de la ejecución de la función, el valor que se ha computado. La asociación de ese valor a la variable se puede hacer, por ejemplo, como en Pascal, asignando explícitamente el valor al nombre de la función en cualquier punto del programa. El compilador que genera código LCEP incluye, al final de las sentencias que definen el cuerpo de la función, una sentencia especial (**return**) que se traduce (con la función de traducción τ) como una lectura de la variable:

$$\begin{array}{l} M_S(x_1, x_2, \ldots, x_n, x, y, \ldots, u, v, \ldots) = \\ \left(\begin{array}{l} !\widehat{_{x_1}} \ x_1' \ \parallel !\widehat{_{x_2}} \ x_2' \ \parallel !\widehat{_{x_n}} \ x_n' \ \parallel !\widehat{_x} \ 0 \ \parallel !\widehat{_y} \ 0 \ \parallel \ \cdots \ \parallel !\widehat{_z} \ 0 \ \parallel !\widehat{_f} \ 0 \ \parallel \\ \parallel \ \tau[S(x_1, x_2, \ldots, x_n, x, y, \ldots, z, u, v, \ldots, w, f); \mathbf{return} \ f] \end{array} \right) \end{array}$$

$$\tau[\mathbf{return} \ f] = \lambda_f x.x$$

Hacemos notar que la sentencia *return* no es utilizada por el programador. Es una instrucción especial añadida por el compilador que consigue el efecto, al ser transformada en un proceso, de leer el valor de retorno que se ha asociado a la función

mediante alguna asignación. Al añadirse como última instrucción del cuerpo del subprograma, queda como valor de retorno (resultado de la ejecución) del mismo. Al final, el proceso resultante de la ejecución de la función es el propio valor resultado de la función.

En el caso de los procedimientos, el resultado de la ejecución es el proceso nulo 0 de LCEP, que hace las veces de valor de retorno especial ($\diamondsuit$). Las propiedades algebraicas del proceso nulo como elemento neutro de las operaciones $\circ$ y $\|$ permiten que la composición secuencial y paralela de s-procesos resultantes de transformar procedimientos complete correctamente su ejecución.

También los procesos del sistema que expresan subprogramas de tipo procedimiento son $\mathcal{H}$-evaluados siempre al proceso nulo.

Ejemplo 17 *Con la instrucción elemental $PRINT$ asociamos una regla de $\mathcal{H}$-evaluación:*

$$PRINT \circ a \ \rightarrow_{eval_{print}} \ 0$$

cuyo efecto lateral es la presentación de a por pantalla, y que se evalúa siempre al proceso nulo.

Un tratamiento análogo se da a los demás procedimientos primitivos. De esta manera, un procedimiento definido mediante composición de llamadas a procedimiento (que, en última instancia, son llamadas a procedimientos primitivos) tiene como resultado final de la ejecución un proceso nulo, puesto que la composición secuencial ($\circ$) y paralela ($\|$) de procesos (finalmente evaluados) nulos es un proceso nulo.

5.1.6 Parametrización parcial

Podemos aprovechar las características del formalismo para permitir utilizar funciones definidas a partir de otras ya existentes, por instanciación parcial de sus argumentos. Al traducir la definición de una función, establecemos una ligadura entre el nombre de los parámetros formales y la función donde aparecen. Ello dificulta emplear, a través de sentencias **def**, la instanciación parcial de argumentos para definir una función en términos de otra. Por ejemplo,

$$\textbf{def } F(x, y) = S(x, y);$$
$$\textbf{def } G(z) = F(2{-}{>}\, y);$$

La función G tiene un único parámetro formal, que debe ser asociado al parámetro x de F. Pero el mecanismo actual de traducción no permite hacer esto con comodidad. Además, una definición como "**def** $G = F(2{-}{>}\, y)$;" también produce una función

con un parámetro formal que debe ser referenciado, en llamadas con especificación de parámetros simbólicos, mediante la etiqueta x heredada de la declaración de F.

Parece más sencillo prohibir el uso de llamadas parciales en el contexto de la definición de una función y permitirlo solamente en el contexto de la asignación de variables, donde el tratamiento es más sencillo. Analicemos cómo puede hacerse. Vamos a mostrar los problemas existentes con un ejemplo.

Ejemplo 18 *Supongamos que tenemos el siguiente programa:*

$$\begin{array}{l} \textbf{def } F(x_1, x_2) = S(x_1, x_2); \\ \quad \textbf{begin} \\ \qquad x := F(2 -> x_2); \\ \quad \textbf{end}. \end{array}$$

La traducción del mismo es:

$$\begin{array}{l} !\left(\widehat{_F}\left(\begin{array}{l} \lambda_1 x_1'.(\lambda_1 x_2'.M_S(x_1', x_2') + \lambda_{F_{x_2}} x_2'.M_S(x_1', x_2'))+ \\ \lambda_{F_{x_1}} x_1'.(\lambda_1 x_2'.M_S(x_1', x_2') + \lambda_{F_{x_2}} x_2'.M_S(x_1', x_2')) \end{array}\right)\right) \parallel \\ \lambda_\sigma \sigma'.(ASIG \circ x \circ \sigma' \parallel 0) \parallel \widehat{_\sigma}\ \tau[F(2 -> x_2)] = \end{array}$$

$$\begin{array}{l} !\left(\widehat{_F}\left(\begin{array}{l} \lambda_1 x_1'.(\lambda_1 x_2'.M_S(x_1', x_2') + \lambda_{F_{x_2}} x_2'.M_S(x_1', x_2'))+ \\ \lambda_{F_{x_1}} x_1'.(\lambda_1 x_2'.M_S(x_1', x_2') + \lambda_{F_{x_2}} x_2'.M_S(x_1', x_2')) \end{array}\right)\right) \parallel \\ \lambda_\sigma \sigma'.(ASIG \circ x \circ \sigma' \parallel 0) \parallel \widehat{_\sigma}(\lambda_F F'.(F' \circ \widehat{_{F_{x_2}}}\ 2)) \end{array}$$

y esto, al ejecutarlo, lleva a:

$$\begin{array}{l} !\left(\widehat{_F}\left(\begin{array}{l} \lambda_1 x_1'.(\lambda_1 x_2'.M_S(x_1', x_2') + \lambda_{F_{x_2}} x_2'.M_S(x_1', x_2'))+ \\ \lambda_{F_{x_1}} x_1'.(\lambda_1 x_2'.M_S(x_1', x_2') + \lambda_{F_{x_2}} x_2'.M_S(x_1', x_2')) \end{array}\right)\right) \parallel \\ \lambda_\sigma \sigma'.(ASIG \circ x \circ \sigma' \parallel 0) \parallel \widehat{_\sigma}(\lambda_F F'.(F' \circ \widehat{_{F_{x_2}}}\ 2)) \ \rightarrow_{\beta_F} \end{array}$$

$$\begin{array}{l} !\left(\widehat{_F}\left(\begin{array}{l} \lambda_1 x_1'.(\lambda_1 x_2'.M_S(x_1', x_2') + \lambda_{F_{x_2}} x_2'.M_S(x_1', x_2'))+ \\ \lambda_{F_{x_1}} x_1'.(\lambda_1 x_2'.M_S(x_1', x_2') + \lambda_{F_{x_2}} x_2'.M_S(x_1', x_2')) \end{array}\right)\right) \parallel \\ \lambda_\sigma \sigma'.(ASIG \circ x \circ \sigma' \parallel 0) \parallel \\ \widehat{_\sigma}\left(\left(\begin{array}{l} \lambda_1 x_1'.(\lambda_1 x_2'.M_S(x_1', x_2') + \lambda_{F_{x_2}} x_2'.M_S(x_1', x_2'))+ \\ \lambda_{F_{x_1}} x_1'.(\lambda_1 x_2'.M_S(x_1', x_2') + \lambda_{F_{x_2}} x_2'.M_S(x_1', x_2')) \end{array}\right) \circ \widehat{_{F_{x_2}}}\ 2\right) \\ \rightarrow_{\rho_{sec3}} \end{array}$$

Podemos aplicar esta regla considerando, de manera natural, que los parámetros for-

males inducen el orden $F_{x_1} \prec F_{x_2}$.

$$
\begin{array}{l}
!\left(\widehat{_F}\left(\begin{array}{l}\lambda_1 x_1'.(\lambda_1 x_2'.M_S(x_1',x_2') + \lambda_{F_{x_2}} x_2'.M_S(x_1',x_2'))+ \\ \lambda_{F_{x_1}} x_1'.(\lambda_1 x_2'.M_S(x_1',x_2') + \lambda_{F_{x_2}} x_2'.M_S(x_1',x_2'))\end{array}\right)\right) \parallel \\
\lambda_\sigma \sigma'.(ASIG \circ x \circ \sigma' \parallel 0) \parallel \\
\widehat{_\sigma}(\lambda_{F_{x_1}} x_1'.((\lambda_1 x_2'.M_S(x_1',x_2') + \lambda_{F_{x_2}} x_2'.M_S(x_1',x_2')) \circ \widehat{_{F_{x_2}}}\ 2)) \ \rightarrow_{\beta_{sec}}
\end{array}
$$

$$
\begin{array}{l}
!\left(\widehat{_F}\left(\begin{array}{l}\lambda_1 x_1'.(\lambda_1 x_2'.M_S(x_1',x_2') + \lambda_{F_{x_2}} x_2'.M_S(x_1',x_2'))+ \\ \lambda_{F_{x_1}} x_1'.(\lambda_1 x_2'.M_S(x_1',x_2') + \lambda_{F_{x_2}} x_2'.M_S(x_1',x_2'))\end{array}\right)\right) \parallel \\
\lambda_\sigma \sigma'.(ASIG \circ x \circ \sigma' \parallel 0) \parallel \widehat{_\sigma}(\lambda_{F_{x_1}} x_1'.M_S(x_1',2)) \ \rightarrow_{\beta_\sigma}
\end{array}
$$

$$
\begin{array}{l}
!\left(\widehat{_F}\left(\begin{array}{l}\lambda_1 x_1'.(\lambda_1 x_2'.M_S(x_1',x_2') + \lambda_{F_{x_2}} x_2'.M_S(x_1',x_2'))+ \\ \lambda_{F_{x_1}} x_1'.(\lambda_1 x_2'.M_S(x_1',x_2') + \lambda_{F_{x_2}} x_2'.M_S(x_1',x_2'))\end{array}\right)\right) \parallel \\
ASIG \circ x \circ \lambda_{F_{x_1}} x_1'.M_S(x_1',2) \parallel 0 \ \rightarrow_{eval_{asig}}
\end{array}
$$

$$
\begin{array}{l}
!\left(\widehat{_F}\left(\begin{array}{l}\lambda_1 x_1'.(\lambda_1 x_2'.M_S(x_1',x_2') + \lambda_{F_{x_2}} x_2'.M_S(x_1',x_2'))+ \\ \lambda_{F_{x_1}} x_1'.(\lambda_1 x_2'.M_S(x_1',x_2') + \lambda_{F_{x_2}} x_2'.M_S(x_1',x_2'))\end{array}\right)\right) \parallel \\
!(\widehat{_x}\ \lambda_{F_{x_1}} x_1'.M_S(x_1',2)) \parallel 0 \ \rightarrow_{\pi_{\parallel_1}}
\end{array}
$$

$$
\begin{array}{l}
!\left(\widehat{_F}\left(\begin{array}{l}\lambda_1 x_1'.(\lambda_1 x_2'.M_S(x_1',x_2') + \lambda_{F_{x_2}} x_2'.M_S(x_1',x_2'))+ \\ \lambda_{F_{x_1}} x_1'.(\lambda_1 x_2'.M_S(x_1',x_2') + \lambda_{F_{x_2}} x_2'.M_S(x_1',x_2'))\end{array}\right)\right) \parallel \\
!(\widehat{_x}\ \lambda_{F_{x_1}} x_1'.M_S(x_1',2))
\end{array}
$$

Como puede observarse, el contenido del m-proceso asociado a la variable x contiene una abstracción (que ya no es multicanal) pero permanece ligado al nombre de la función de la que procede (F). Sin embargo, en el proceso de instanciación parcial de la función original hemos perdido la posibilidad de instanciar el primer parámetro formal a través de un canal numérico.

Debemos permitir también utilizar x como un subprograma, esto es, debemos permitir pasar parámetros al proceso representado por la variable x. Pero la traducción de una sentencia como $x(1)$ no produce el resultado correcto, ya que se traduce como $\lambda_x x'.(x' \circ \widehat{_1}\ 1)$ y esto no puede nunca llegar a instanciar el primer argumento de F, como sería de esperar. Una solución simple a este problema pasa por la prohibición, en la sintaxis del lenguaje de programación, de la instanciación parcial mediante etiquetado simbólico[3].

[3]Un compilador, sin embargo, puede permitir también la instanciación parcial de parámetros reales en un verdadero subprograma (no anónimo) generando código LCEP con el etiquetado numérico correspondiente.

Es fácil comprobar, sobre el mismo ejemplo planteado, que la sentencia $x := F(2-> 2)$; produce el m-proceso $!(\underset{x}{\frown}\ \lambda_1 x'.\tau[S(x',2)])$ que sí puede activarse de la manera habitual mediante instanciación posicional (e incluso por etiquetado numérico) mediante la llamada $x(1)$, por ejemplo.

A los subprogramas así definidos, recogidos en una variable, los llamamos subprogramas *anónimos*. Poseen las mismas características que los subprogramas de los que proceden, pero sólo pueden ser parametrizados posicional o numéricamente. Con lo que acabamos de describir, hemos indicado cómo utilizar en el LMAN y soportar mediante LCEP la parametrización parcial de subprogramas.

Inhibición de la evaluación de parámetros reales

Otro aspecto interesante a considerar, y que está relacionado con los subprogramas anónimos, es la posibilidad de inhibir la evaluación de parámetros al realizar la llamada a un subprograma. Esto puede verse como una forma de combinar los esquemas CBV y CBN (llamada por "nombre") a la hora de tratar la evaluación de los parámetros de un subprograma. Lenguajes como LISP [All90, Ste84] permiten inhibir la evaluación de parámetros a una función prefijando éste con un apóstrofe ("quote"). Empleamos el símbolo @ para indicar que un parámetro no debe ser evaluado.

Sin embargo, debido a la posibilidad de que la evaluación de un parámetro emplee variables locales que no sean visibles en el momento (diferido) de la evaluación total del parámetro, se da soporte a una versión restringida de esta característica. En particular, se permite emplear cualquier subprograma G (incluso un procedimiento[4]) total o parcialmente instanciado como parámetro real de un subprograma F. Sin embargo, aunque la ejecución del subprograma G se deja en suspenso para que se realice bajo el control del subprograma F donde se pasa como parámetro real, no ocurre así con la evaluación de los parámetros reales de @, que deben ser evaluados en el entorno de ejecución donde se está procesando la llamada. Esto se debe a que un posible uso de variables locales en la parametrización de G puede no ser gestionado desde F, ya que F se ejecuta en un entorno de ejecución propio.

Sintácticamente, podemos escribir en nuestro programa (suponiendo que x es una variable local asignada previamente al valor 2) $F(1, @G(x))$ lo cual se interpreta como: "Ejecutar F pasando como primer parámetro el número 1 y como segundo parámetro el subprograma $G(2)$". Observemos que el parámetro para G sí es evaluado. La variable (local) x no debe ser visible dentro de F, por lo que no podemos computar su valor asociado con posterioridad. Debe instanciarse al realizar la llamada a F. Por

[4]Observemos que, puesto que la ejecución del subprograma pasado como parámetro se realiza posteriormente por el subprograma que lo recibe como parámetro, es responsabilidad del segundo subprograma emplear adecuadamente el subprograma recibido. En el Apéndice B mostramos cómo hacer esto al definir ALEPH, que incorpora esta característica.

eso, el subprograma pasado a F es $G(2)$. Reparemos en que $G(2)$ es un subprograma sin parámetros. Su ejecución, dentro de F, puede tener lugar varias veces, sin más que F utilice su segundo parámetro x_2 para efectuar varias llamadas a procedimiento, escribiendo simplemente:

$$x_2; \ldots; x_2; \ldots; x_2; \ldots$$

Observemos que, en el entorno de ejecución de F, x_2 puede ser accedido como un subprograma anónimo.

Implementación de la recursividad

La posibilidad de definir subprogramas recursivos es una característica importante que ofrecen muchos lenguajes de programación. También puede recibir soporte formal en el contexto de LCEP. A lo largo de este apartado se destaca la importancia de la idea de entorno de ejecución para la ejecución de subprogramas. El problema de la recursividad está intimamente relacionado con él, ya que las etiquetas para los entornos de ejecución de los distintos subprogramas activados en el programa se definen por el compilador en tiempo de compilación. Esto significa que, en tiempo de ejecución, en un subprograma recursivo tenemos entornos de ejecución homónimos anidados. Podría haber una cierta confusión en los destinos de cada evaluación al completarse la ejecución de una fase de la recursividad en su correspondiente entorno de ejecución, si no fuera porque la regla $\mu_{\mathcal{T}}$ exige, para el establecimiento de un túnel transparente, que los constructores de output interpuestos vengan determinados por etiquetas estrictamente mayores que la del canal sobre el cual se realiza la comunicación paralela. Entonces, el contenido de un entorno de ejecución homónimo anidado no es $\mathcal{T}$-comunicable mediante un túnel que contenga el canal que etiqueta el entorno, pues ninguna etiqueta es estrictamente superior a sí misma. Para clarificar esto, presentamos una de las etapas de la ejecución del subprograma recursivo F que calcula el factorial de un número entero.

Ejemplo 19 *Factorial de n:*

```
def F(n) =
block
    if n > 0 then F := n * F(n - 1) else F := 1;
eblock;
    begin
        print(F(3)) ;
    end.
```

En un momento dado de la ejecución del proceso $\mathcal{M}$ asociado a este programa aparece el siguiente proceso:

$$
\begin{array}{l}
M_\varphi \parallel M_F \parallel \lambda_{\sigma_2}\sigma'_2.\lambda_{print}print'.(print' \circ \widehat{1}\ \sigma'_2) \parallel \\
\widehat{\sigma_2}\left(\begin{array}{l}
!\widehat{n}\ 3 \parallel !\widehat{f}\ 0 \parallel \lambda_{\sigma_{112}}\sigma'_{112}.(ASIG \circ f \circ \sigma'_{112} \parallel \lambda_f x.x) \parallel \\
\lambda_{\sigma_{1122}}\sigma'_{1122}.\lambda_{mult}mult'.\widehat{\sigma_{112}}(mult' \circ \widehat{1}\ 3 \circ \widehat{1}\ \sigma'_{1121}) \parallel \\
\widehat{\sigma_{1122}}\left(\begin{array}{l}
!\widehat{n}\ 2 \parallel !\widehat{f}\ 0 \parallel \lambda_{\sigma_{112}}\sigma'_{112}.(ASIG \circ f \circ \sigma'_{112} \parallel \lambda_f x.x) \parallel \\
\lambda_{\sigma_{1122}}\sigma'_{1122}.\lambda_{mult}mult'.\widehat{\sigma_{112}}(mult' \circ \widehat{1}\ 2 \circ \widehat{1}\ \sigma'_{1121}) \parallel \\
\widehat{\sigma_{1122}}\left(\begin{array}{l}
!\widehat{n}\ 1 \parallel !\widehat{f}\ 0 \parallel \lambda_{\sigma_{112}}\sigma'_{112}.(ASIG \circ f \circ \sigma'_{112} \\
\parallel \lambda_f x.x) \parallel \\
\lambda_{\sigma_{1122}}\sigma'_{1122}.\lambda_{mult}mult'.\widehat{\sigma_{112}} \\
(mult' \circ \widehat{1}\ 1 \circ \widehat{1}\ \sigma'_{1121}) \parallel \\
\lambda_F F'.(\widehat{\sigma_{1122}}(F' \circ \widehat{1}\ 0))
\end{array}\right)
\end{array}\right)
\end{array}\right)
\end{array}
$$

Debemos tener en cuenta que:

- *El m-proceso obtenido de la declaración de F se representa como M_F. Es un m-proceso con dirección F.*
- *Se ha simplificado la traducción, eliminando elementos que no son relevantes en la discusión.*
- *Observamos la presencia de los procesos del sistema (mayor, mult, menos, if, ...) agrupados en el proceso M_φ.*

Esquemáticamente, podemos representar el proceso analizado como sigue:

$$
\begin{array}{l}
M_\varphi \parallel M_F \parallel print(F(3)) \parallel \\
\widehat{\sigma_2}\left(\begin{array}{l}
!\widehat{n}\ 3 \parallel !\widehat{f}\ 0 \parallel f := 3 * F(2) \parallel \\
(1)\quad \lambda_{\sigma_{1122}}\sigma'_{1122}.\lambda_{mult}mult'.\widehat{\sigma_{112}}(mult' \circ \widehat{1}\ 3 \circ \widehat{1}\ \sigma'_{1121}) \parallel \\
\widehat{\sigma_{1122}}\left(\begin{array}{l}
!\widehat{n}\ 2 \parallel !\widehat{f}\ 0 \parallel f := 2 * F(1) \parallel \\
(2)\quad \lambda_{\sigma_{1122}}\sigma'_{1122}.\lambda_{mult}mult'.\widehat{\sigma_{112}} \\
(mult' \circ \widehat{1}\ 2 \circ \widehat{1}\ \sigma'_{1121}) \parallel \\
\widehat{\sigma_{1122}}\left(\begin{array}{l}
!\widehat{n}\ 1 \parallel !\widehat{f}\ 0 \parallel f := 1 * F(0) \parallel \\
(3)\quad \lambda_{\sigma_{1122}}\sigma'_{1122}.\lambda_{mult}mult'.\widehat{\sigma_{112}} \\
(mult' \circ \widehat{1}\ 1 \circ \widehat{1}\ \sigma'_{1121}) \parallel \\
\lambda_F F'.(\widehat{\sigma_{1122}}(F' \circ \widehat{1}\ 0))
\end{array}\right)
\end{array}\right)
\end{array}\right)
\end{array}
$$

El proceso mostrado corresponde al inicio de la última fase del cálculo de $F(3)$, la base de la recursión (el cálculo de $F(0)$). Todavía no se ha aplicado F al argumento 0. Sin embargo, podemos notar:

- *La existencia de un entorno de ejecución σ_2 para el cálculo de $F(3)$.*
- *El primer entorno de ejecución σ_{1122} definido para la evaluación de $F(2)$ (el más externo).*

- *El segundo entorno de ejecución* σ_{1122} *definido para la evaluación de* $F(1)$.

- *Un tercer entorno de ejecución* σ_{1122} *definido para la evaluación de* $F(0)$, *que todavía está prefijado por* $\lambda_F F'$.

- *Los receptores del resultado de la ejecución local en los entornos* σ_{1122} *se han distinguido con* (1), (2) *y* (3), *respectivamente. Responden a la necesidad de obtener los valores de* $F(2)$, $F(1)$ *y* $F(0)$, *antes de poder calcular* $3 * F(2)$, $2 * F(1)$, ...

- *Las variables locales* n *(parámetro formal) y* f *(valor de retorno), como* m*-procesos encerrados en cada entorno de ejecución mencionado.*

Debemos asegurar que el contenido del segundo entorno de ejecución σ_{1122} *no es recibido en el primero. Observamos que, en principio, existe un túnel* γ_{τ_1} *entre el segundo entorno* σ_{1122} *y el receptor de la evaluación del primer entorno* σ_{1122}, *que se ha marcado con* (1). *El s-túnel asociado* τ_1 *verifica que* $h(\tau_1) = \{\sigma_{1122}\}$ *y, por tanto, no es* σ_{1122}*-transparente. No hay posibilidad de comunicación con* (1).

Sin embargo, el túnel γ_{τ_2} *entre el mismo entorno de ejecución* σ_{1122} *(el segundo) y* (2) *verifica* $h(\tau_2) = \emptyset$, *lo que permite la* $\beta_{\sigma_{1122}}$*-comunicación con* (2). *Esto es lo que esperamos.*

El mismo planteamiento que acabamos de expresar podemos repetirlo para el entorno de ejecución σ_{112} *que se asocia a la ejecución local de mult. Aunque no se hayan distinguido los subprocesos relacionados con él, podemos ver que se encuentran también replicados, pero no exactamente anidados. Tenemos igualmente casos de* $\mathcal{T}$*-comunicación potencial sobre el canal* σ_{112} *que involucran túneles cuyo conjunto de símbolos incluyen etiquetas* σ_{1122}. *Puesto que nos interesa inhibir dichas comunicaciones, que violan la localidad de la ejecución de cada etapa de recursión, debemos asegurar que dichos túneles no sean* σ_{112}*-transparentes. Lo hacemos exigiendo que las etiquetas empleadas para construir entornos de ejecución no sean comparables. En ese caso, no se cumplen las condiciones necesarias y no hay* $\beta_{\sigma_{112}}$*-comunicación, más que cuando el túnel no contiene etiquetas empleadas para construir entornos. Entonces se da únicamente la comunicación que deseamos.*

Pueden hacerse consideraciones semejantes respecto de las variables locales cuyos m*-procesos tienen direcciones sobrecargadas. En este caso, basta con exigir a la relación de orden establecida en* $\mathcal{L}$ *que las etiquetas asignadas como direcciones de los* m*-procesos locales no sean comparables con las etiquetas empleadas para definir los* σ*-entornos de ejecución.*

5.1.7 Definición de la relación de orden $\preceq_{\mathcal{L}}$

La simulación de los programas de más alto nivel con LCEP se apoya en una definición apropiada de la relación de orden $\preceq_{\mathcal{L}}$ sobre el conjunto de etiquetas que expresan canales de comunicación. Ya se han descrito todos los conceptos donde dicha relación de orden juega algún papel:

- La representación de la memoria abstracta mediante m-procesos y la definición correcta del alcance de los objetos del lenguaje de programación que describen dichos m-procesos.
- El mecanismo de activación de subprogramas, incluyendo la simulación del esquema de evaluación CBV, emplea σ-entornos de ejecución para los argumentos y para el propio cuerpo del subprograma, y dichos entornos deben ser transparentes respecto a los objetos globales pero capaces de acotar el alcance de los m-procesos locales que puedan existir en su interior.
- La implementación de la parametrización de los subprogramas también está intimamente relacionada con el orden establecido entre las etiquetas que representan los parámetros formales para el mismo.
- La posibilidad de definir funciones recursivas exige que los entornos de ejecución donde se ejecuta cada etapa de la recursión (entornos que están anidados) no pueden interferirse entre sí. Ello exige definir adecuadamente el orden existente entre las etiquetas σ que sirven para establecer los entornos de ejecución.

De acuerdo con estas consideraciones, se procede como sigue:

1. Se realiza una partición del conjunto S en cuatro subconjuntos:
   - S_G, del cual toma el compilador las etiquetas necesarias para traducir los objetos globales del lenguaje de programación: funciones, variables, canales síncronos, etc.
   - S_S, del cual toma el compilador las etiquetas que necesita para traducir las sentencias $S \in \mathcal{H}$ a procesos $M \in \mathcal{M}$. Son las de tipo σ empleadas para crear los entornos de ejecución.
   - S_E, mediante el cual se representan los parámetros formales simbólicos de las funciones y procedimientos del lenguaje de programación.
   - S_L, de donde se escogen las etiquetas simbólicas necesarias para representar los objetos locales: variables y canales síncronos, además de los parámetros formales de las funciones y procedimientos, que reciben tratamiento de variables locales durante la ejecución del cuerpo del subprograma. También se incluyen aquí los canales auxiliares empleados para implementar la composición paralela de sentencias.

2. Definimos, así, la relación de orden parcial $\preceq_{\mathcal{L}}$:

   - $\preceq_{\mathcal{L}_1}$: $m \leq n \Rightarrow m \preceq_{\mathcal{L}} n,\ m, n \in \mathbb{N}$. Es decir, las etiquetas numéricas heredan el orden de los números naturales.

   - $\preceq_{\mathcal{L}_2}$: $n \prec_{\mathcal{L}} P.\ \forall n \in \mathbb{N},\ \forall P \in S$: Las etiquetas simbólicas son mayores que las numéricas.

   - $\preceq_{\mathcal{L}_3}$: $P \prec_{\mathcal{L}} Q.\ \forall P \in S_S,\ \forall Q \in S_G$: Cualquier etiqueta asociada a la evaluación de sentencias es menor que una etiqueta simbólica global. Esto permite que los entornos de ejecución sean transparentes a las comunicaciones que involucran procesos del sistema y objetos de alcance global.

   - $\preceq_{\mathcal{L}_4}$: $F_{x_i} \prec_{\mathcal{L}} F_{x_{i+1}},\ i = 1, 2 \ldots n-1$ y $F_{x_i} \in S_E$ para $i = 1, 2 \ldots n$ si la función F está definida en el programa fuente como:

     $$\textbf{def}\ F(x_1, x_2, \ldots, x_i, \ldots, x_j, \ldots, x_n) = S(x_1, x_2, \ldots, x_n, \ldots)$$

     Esto permite el tratamiento completo de los distintos esquemas de parametrización.

   - $\preceq_{\mathcal{L}_5}$: Sólo pertenecen a $\preceq_{\mathcal{L}}$ los pares $\langle P, Q \rangle$ definidos por las reglas anteriores y las propiedades de $\preceq_{\mathcal{L}}$ como orden parcial.

   Reparemos en que la última condición evita la existencia de túneles no deseados. En particular, permite restringir el alcance de los m-procesos locales al entorno de ejecución donde se encuentran, así como mantener los distintos entornos definidos para evaluar subprogramas protegidos de interferencias externas no deseadas.

En la Figura 5.4 se presenta un esquema de la relación de orden definida.

5.1.8 Tratamiento de las sentencias de un LMAN como procesos

Vamos a desarrollar la definición de la función $\tau : \mathcal{H} \rightarrow \mathcal{M}$, que nos permite transformar las construcciones del lenguaje de más alto nivel (LMAN) en el proceso LCEP $M_{\mathcal{H}}$ correspondiente. Como se describe en los apéndices, las construcciones del LMAN que vamos a manejar son las siguientes: el uso de variables locales y globales, la asignación, los subprogramas (procedimientos y funciones), el condicional, el bucle, el envío y la recepción de mensajes, la secuenciación de instrucciones y el paralelismo. El lenguaje ALEPH recoge todas estas construcciones y su descripción sintáctica completa aparece en el Apéndice B.

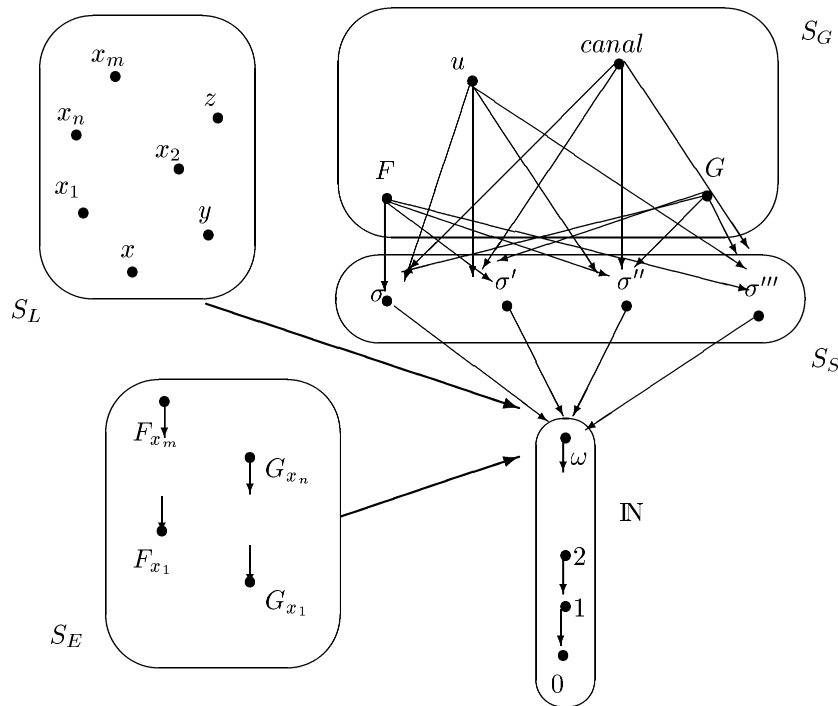


Figura 5.4: Esquema de la relación de orden

Tratamiento dinámico de la memoria abstracta

En el Apartado 5.1.1 vimos un modelo para el tratamiento de los objetos residentes en memoria como m-procesos capaces de entregar un valor (contenido del m-proceso) mediante una operación de comunicación sobre un canal dedicado (la dirección del m-proceso). Este esquema parece claro cuando se consideran operaciones de lectura, pero, ¿qué ocurre con las operaciones de escritura en memoria? Es evidente que para reflejar adecuadamente la actualización de una variable del lenguaje de programación necesitamos sustituir el contenido del m-proceso que la representa por otro con el nuevo contenido. Esto no puede soportarse directamente en el formalismo de manera practicable, de forma que su tratamiento se efectúa mediante un *efecto lateral*, introduciendo una regla de evaluación especial que realiza este cometido. La operación de *asignación*, que es el recurso expresivo que el LMAN proporciona para realizar las modificaciones en la memoria abstracta, se considera como un proceso del sistema, *asig*, que:

- Evalúa la expresión cuyo valor final se desea asociar con una variable para que quede ligado al símbolo bajo su forma normal.

- Crea un proceso que recoge la forma evaluada del proceso que se desea asociar

a la variable, y la emplea para construir un proceso simple $ASIG \circ Var \circ \sigma'$, donde la variable σ' recibe el valor calculado para la expresión a asociar a la variable Var.

- Se introduce la constante $ASIG$ en el conjunto C y se asocia una regla de evaluación a $ASIG$ que permite crear un m-proceso con la misma dirección Var y el nuevo contenido. Además, elimina los m-procesos de la misma dirección Var que puedan encontrarse en el alcance del nuevo m-proceso creado. La regla de $\mathcal{H}$-evaluación sólo se activa cuando el nuevo valor a asociar a la variable del lenguaje ya está evaluado. Esto se consigue sin más que exigiendo que el segundo argumento en su regla de $\mathcal{H}$-evaluación no sea una variable ligada. La regla asociada es:

$$ASIG \circ x \circ M \rightarrow_{eval_{asig}} !_{\widehat{x}}\ M$$

siempre que M no sea una variable ligada. Como efecto lateral, elimina todas las ocurrencias de procesos $!_{\widehat{x}}\ M$ en el alcance del nuevo proceso.

Otra vertiente de la problemática de dar soporte al dinamismo de la utilización de la memoria procede del uso de variables locales. En efecto, las variables locales están asociadas a la presencia de elementos de computación transitorios, como son los subprogramas. Los subprogramas introducen, al ser activados, un conjunto de objetos en la memoria abstracta que deben ser eliminados cuando el subprograma completa su ejecución. Este efecto se implementa también mediante efectos laterales. La solución dada pasa por introducir un proceso del sistema, $kill$, que se invoca al terminar la ejecución de un subprograma en su entorno de ejecución y que elimina todos los m-procesos que puedan encontrarse en éste. Dichos m-procesos son los asociados a las variables locales y parámetros del subprograma. La acción descrita viene asociada a una regla de $\mathcal{H}$-evaluación:

$$\begin{array}{c} KILL \circ (!_{\widehat{x_1}}\ a_1\ \parallel !_{\widehat{x_2}}\ a_2\ \parallel\ \cdots\ \parallel !_{\widehat{x_n}}\ a_n\ \parallel \\ \parallel !_{\widehat{x}}\ a\ \parallel !_{\widehat{y}}\ b\ \parallel\ \cdots\ \parallel !_{\widehat{z}}\ c\ \parallel !_{\widehat{f}}\ d\ \parallel\ M)\ \rightarrow_{eval_{kill}}\ M \end{array}$$

cuyo efecto es precisamente el indicado, eliminar todos los m-procesos presentes en el entorno de ejecución del subprograma, dejando únicamente el resultado de su ejecución. Dicho resultado es el proceso nulo 0, si se trata de un procedimiento, o el valor de retorno, si se trata de una función. Esto modifica el l-proceso asociado a un

subprograma, que toma la forma definitiva siguiente:

$$M_S(x_1, x_2, \ldots, x_n, x, y, \ldots, z, u, v, \ldots, w, f(*)) =$$

$$\lambda_\sigma \sigma'.\lambda_{kill} kill'.(kill' \circ\widehat{_1}\, \sigma') \parallel$$
$$\widehat{_\sigma} \begin{pmatrix} \lambda_{asig} asig'.asig' \circ\widehat{_1}\, x_1 \circ\widehat{_1}\, x'_1 \parallel \cdots \parallel \\ \lambda_{asig} asig'.asig' \circ\widehat{_1}\, x_n \circ\widehat{_1}\, x'_n \parallel \\ \lambda_{asig} asig'.asig' \circ\widehat{_1}\, x \circ\widehat{_1}\, 0 \parallel \cdots \parallel \lambda_{asig} asig'.asig' \circ\widehat{_1}\, z \circ\widehat{_1}\, 0 \parallel \\ \lambda_{asig} asig'.asig' \circ\widehat{_1}\, f \circ\widehat{_1}\, 0 \quad (*) \\ \tau[S(x_1, x_2, \ldots, x_n, x, y, \ldots, z, u, v, \ldots); \mathbf{return}\ f(*)] \end{pmatrix}$$

$(*)$ Sólo en el caso de funciones.

Puede apreciarse la creación de un σ-entorno de ejecución para el subprograma. Cuando se completa, el contenido del entorno se pasa como argumento de la función *kill* que se obtiene del proceso del sistema correspondiente. A continuación, se debe aplicar la regla de evaluación anterior.

Traducción de las sentencias del LMAN

Vamos a describir la función de traducción τ aplicándola a cada una de las sentencias del lenguaje. La ligadura que se establece entre los procesos asociados a cada sentencia, como consecuencia de la existencia de constructores de sentencia, se da implícitamente en el caso de la composición secuencial y explícitamente en la composición paralela. La definición de la función τ es recursiva. A todos los efectos, cuando, por causa de la definición recursiva de la función de traducción, encontremos que debemos aplicarla sobre una sentencia vacía, asumimos que, si representamos dicha sentencia como ε, entonces $\tau[\varepsilon] + 0$, es decir, el proceso que asociamos a la sentencia vacía es el proceso nulo.

a) Asignación:

La operación de asignación es el recurso expresivo que el LMAN proporciona para realizar modificaciones en la memoria abstracta. Sin embargo, distinguimos la asignación sobre variables locales y sobre variables globales porque implican modificar ámbitos muy diferentes del proceso que se ejecuta, debido a los distintos alcances asociados a los m-procesos que representan los objetos locales y los globales. Esto es importante a la hora de implementar la máquina abstracta que ejecuta los procesos.

a.1 Asignación de una variable local:

Expresión del LMAN	$x := E; S$
Expresión LCEP	$\tau[x := E; S] =$ (1) $\lambda_\sigma \sigma' \quad .\lambda_{asig} asig'. \quad (asig' \circ x \circ \sigma' \parallel \quad \tau[S]) \quad \parallel \quad \tau[E \to \sigma](*)$ (3) (4) (5) (6) (2)

$(*)\ \sigma' \notin FV(\tau[S])$[5]

Observamos en (1) cómo se trata el efecto de la composición secuencial de sentencias en la definición del proceso resultante, y en (2) la creación de un entorno de ejecución para la expresión a evaluar E cuyo resultado final se recoge en (3) y se somete a la acción de $ASIG$ en (4). Vemos también cómo la sentencia que debe ejecutarse a continuación, S, se traduce en un proceso que continúa en paralelo (5) y (6) respecto al m-proceso que se construye por la evaluación de $ASIG$. Podemos advertir que la regla de evaluación de $ASIG$ no puede aplicarse hasta que el valor calculado para la expresión sea recibido (la regla no puede aplicarse cuando el segundo argumento es una variable ligada) pues es entonces cuando se recibe la abstracción de $asig$, que introduce la constante $ASIG$. Otro detalle a considerar es que al prefijar la expresión con el operador de input $\lambda_\sigma \sigma'$, aislamos del exterior el proceso sobre el cual se aplica. Ello significa que no puede tener lugar la ejecución del proceso $\tau[S]$ hasta que la evaluación de E haya finalizado. Esto es lo que el programador espera al escribir una sentencia como la propuesta. Esta manera de trabajar con las variables permite también (al contrario que el esquema CBN) la definición de una variable en función de sí misma. Esto es posible porque el proceso que redefine la variable no es "visible" hasta que no se ha evaluado el proceso que se asocia a la variable; por tanto, la evaluación de la expresión E puede emplear la propia variable x que se está redefiniendo. La traducción debe contribuir a que la semántica de la ejecución de $M_{\mathcal{H}}$, el proceso obtenido a partir del programa, sea la que se espera.

[5] Esta condición, que está presente en todas las fases de definición de τ, expresa la necesidad de que las variables ligadas (que denotamos siempre situando apóstrofes al final) empleadas para traducir una sentencia no deben interferir en la traducción de las siguientes sentencias. Ello se consigue fácilmente empleando, al traducir cada sentencia, símbolos todavía no usados. El mismo problema surge al definir los entornos de ejecución relacionados con la evaluación de parámetros reales y su solución es semejante.

a.2 Asignación de una variable global:

Expresión del LMAN	$u := E; S$
Expresión LCEP	$\tau[u := E; S] =$ $\lambda_\sigma \sigma'. \quad (\widehat{global} \quad (u \circ \sigma') \circ \tau[S]) \parallel \quad \tau[E \rightarrow \sigma](*)$ (1) (2) (3)

$(*)\ \sigma' \notin FV(\tau[S])$

La asignación de una variable global se realiza a través de un proceso del sistema, $global$, que se define de este modo:

$$!(\lambda_{global} x.GLOBAL \circ x)$$

y que tiene asociada la regla de evaluación siguiente:

$$GLOBAL \circ (u \circ M) \rightarrow_{eval_{global}} !\widehat{u}\ M$$

Como efecto lateral, elimina cualquier m-proceso con la misma dirección u, en el entorno de ejecución donde se realiza la evaluación. Por la definición del proceso, dicho entorno de evaluación es el básico del sistema. Podemos advertir en (1) que esta vez la conexión con el proceso del sistema se realiza mediante un envío (output) en lugar de esperando la abstracción de un proceso del sistema. El tratamiento diferenciado para variables locales y globales tiene importancia en la implementación de la máquina abstracta. Si consideramos la estructura que tiene el proceso $\mathcal{M}$ que ejecuta la máquina abstracta LCEP, advertimos que todos los m-procesos M_m derivados de la traducción de declaraciones del programa están "juntos" (compuestos mediante el constructor de paralelismo). Si empleamos $asig$, el nuevo m-proceso se crea en el entorno de ejecución donde ocurre la asignación. Además, la ejecución paralela de varias asignaciones sobre la misma variable global resulta más difícil de tratar en la máquina. Existe una razón aún más importante: la asignación de una variable global con $asig$ hace que la variable se pierda, al completarse la ejecución del subprograma pues $KILL$ elimina los m-procesos existentes en el entorno de ejecución del subprograma. De esta manera, $GLOBAL$ funciona como un gestor de memoria global. Por este motivo, en (2) se ha "empaquetado" la dirección (nombre de la variable) y el contenido (expresión ya evaluada) en un único proceso enlazado con el constructor de secuencialidad. Para representar la secuencialidad de los procesos derivados de la sentencia traducida podemos emplear, como en (3), el operador de construcción secuencial de LCEP, dado que la regla β_P permite llevar a cabo la

comunicación con el proceso global.

b) Llamada a procedimiento:

Expresión del LMAN	$F(R_1, R_2, \ldots, R_n); S$
Expresión LCEP	$\underbrace{\tau[F(R_1, R_2, \ldots, R_n)}_{(1)} \; \underbrace{; S]}_{(2)} =$ $\underbrace{\lambda_\sigma \sigma'.(}_{(3)} \; \underbrace{0\circ}_{(5)} \; \underbrace{\tau[S])}_{(6)} \parallel \underbrace{\tau[F(R_1, R_2, \ldots, R_n) \to \sigma]}_{(4)} (*)$

$(*)\ \sigma' \notin FV(\tau[S])$

El tratamiento de esta sentencia es fundamental. Se puede advertir en (1) que se permiten distintos esquemas de parametrización, ya descritos anteriormente. Sin embargo, se puede ver que se exige la instanciación total de todos ellos. De acuerdo con lo expuesto, una llamada a procedimiento debe ser ejecutable[6], para lo cual todos los parámetros formales del subprograma deben resultar instanciados. De nuevo se define implícitamente, en (2), el significado computacional de la composición secuencial de sentencias. En (3) y (4) observamos la creación de un σ-entorno de ejecución para el procedimiento. La evaluación de los parámetros reales y la propia ejecución del procedimiento una vez instanciado con ellos tiene lugar ahí. El algoritmo de ejecución se encarga de que esto sea así[7]. Una vez terminada la ejecución se produce la cancelación del entorno de ejecución mediante comunicación paralela. En (5) podemos reparar en que el posible valor recibido a través del canal σ se desecha. En ese momento, tiene lugar la ejecución del proceso asociado a las restantes sentencias, aquí representado en (6) por $\tau[S]$.

c) Condicional:

Expresión del LMAN	**if** E **then** $S_1; S$ **if** E **then** S_1 **else** $S_2; S$
Expresión LCEP	$\tau[\textbf{if } E \textbf{ then } S_1; S] =$ $\lambda_\sigma \sigma'.\lambda_{if} if'.(if' \circ\widehat{_1}\ \sigma' \circ\widehat{_1}\ \tau[S_1; S] \circ\widehat{_1}\ \tau[S]) \parallel \tau[E \to \sigma]\ (*)$ $\tau[\textbf{if } E \textbf{ then } S_1 \textbf{ else } S_2; S] =$ $\underbrace{\lambda_\sigma \sigma'}_{(1)} \; \underbrace{.\lambda_{if} if'}_{(3)} \; \underbrace{.(if' \circ\widehat{_1}\ \sigma' \circ\widehat{_1}\ \tau[S_1; S] \circ\widehat{_1}\ \tau[S_2; S])}_{(4)} \parallel \underbrace{\tau[E \to \sigma]}_{(2)}$

[6] Dada la semántica asociada con el operador de composición secuencial de sentencias, no tiene sentido plantear otras opciones.

[7] Observar que, en principio, hay una posibilidad de β_P-comunicación inmediata con el canal σ. El algoritmo de ejecución debe ser capaz de retardar dicha comunicación hasta que la ejecución del procedimiento haya concluido. En ese momento, la comunicación sobre σ permite que continue ejecutándose secuencialmente el proceso siguiente $\tau[S]$.

$(*)\ \sigma' \notin FV(\tau[S])$

Se puede observar en (1) y (2) la creación de un entorno de ejecución para la evaluación de la expresión E, que constituye la condición sobre la cual tomar la decisión de ejecutar S_1 o S_2. En (3) y (4) advertimos que se le pasan directamente a la función primitiva **if** los procesos resultantes de la traducción de las sentencias $S_1; S$ y $S_2; S$ (o sólo S en el caso condicional simple). Esto se explica teniendo en cuenta la regla de $\mathcal{H}$-evaluación para IF:

$$IF \circ True \circ M_1 \circ M_2 \rightarrow_{eval_{if}} M_1$$
$$IF \circ False \circ M_1 \circ M_2 \rightarrow_{eval_{if}} M_2$$

d) <u>Bucle</u>:

Expresión del LMAN	**while** E **do** $S_1; S_2$
Expresión LCEP	$\tau[$**while** E **do** $S_1; S_2] =$ $\underset{(1)}{\lambda_{\sigma_0}\sigma'_0.\lambda_{while} while'}\quad \underset{(3)}{(while' \circ\widehat{_1}\ \sigma'_0 \circ\widehat{_1}\ \sigma_0\widehat{_1}\circ}$ $\underset{(4)}{\tau[E \rightarrow \sigma_0]}\quad \underset{(5)}{\circ\widehat{_1}\ \sigma_1 \circ\widehat{_1}}\quad \underset{(6)}{\tau[S_1; 0 \rightarrow \sigma_1]}\quad \underset{(7)}{\circ\widehat{_1}\ \tau[S])}\quad \underset{(2)}{\parallel \tau[E \rightarrow \sigma_0]}$

$\sigma'_0 \notin FV(\tau[E{-}{>}\ \sigma_0]) \cup FV(\tau[S_1; 0{-}{>}\ \sigma_1]) \cup FV(\tau[S])$

Se interpreta $while$ como una función de seis argumentos:

$while(B, \sigma_0, M_E, \sigma_1, M_{S_1}, M_S)$

- El primero (B) adquiere valor $True$ o $False$, una vez evaluada la expresión condicional E (la guarda del bucle) en el entorno de ejecución ((1) y (2)) creado en la primera llamada a $while$. Sirve para determinar qué proceso debe continuar ejecutándose.

- El segundo (σ_0) y el tercero ($M_E = \tau[E \rightarrow \sigma_0]$) permiten reproducir, en una posible iteración posterior, el proceso de evaluación de la guarda. Efectivamente, lo que hacemos es asociar un canal σ_0 a la evaluación de la futura guarda, y preparar su evaluación definiendo un proceso (M_E) que envía el valor evaluado a través de dicho canal. El proceso M_E es, en realidad, un entorno de ejecución para la expresión E que evalúa la guarda del bucle.

- El cuarto (σ_1) y quinto ($M_{S_1} = \tau[S_1; 0 \rightarrow \sigma_1]$) permiten ejecutar el cuerpo del bucle. Para sincronizarlo con la evaluación de la guarda, se emplea el canal σ_1. Esto queda claro al considerar las reglas de $\mathcal{H}$-evaluación de $WHILE$. El proceso M_{S_1} realiza las siguientes tareas:
  - Ejecutar la sentencia (en general compuesta) S_1 (cuerpo del bucle).

  - Enviar un proceso nulo sobre el canal σ_1.

  El canal σ_1 debe tomarse del conjunto S_L, dado que se utiliza para tareas de sincronización.

- El último argumento ($M_S = \tau[S]$) permite continuar la ejecución del proceso una vez concluido el bucle.

Las reglas de $\mathcal{H}$-evaluación clarifican el funcionamiento de esta traducción.

* <u>Continuación del bucle</u>:

$$WHILE \circ TRUE \circ \sigma_0 \circ M_E \circ \sigma_1 \circ M_{S_1} \circ M_S \rightarrow_{eval_{while}}$$
$$(M_{S_1} \parallel \lambda_{\sigma_0}\sigma'_0.(WHILE \circ \sigma'_0 \circ \sigma_0 \circ M_E \circ \sigma_1 \circ M_{S_1} \circ M_S) \parallel \lambda_{\sigma_1}\sigma'_1.M_E)$$

Puede observarse que la regla ejecuta el proceso M_{S_1} que, primero, ejecuta el cuerpo del bucle y, a continuación, envía un proceso nulo sobre el canal σ_1. Al hacerlo, se ejecuta M_E, que es el σ_0-entorno de ejecución para E, la guarda del bucle. Cuando se ha evaluado la guarda del bucle, se envía sobre el canal σ_0 y de nuevo se puede aplicar la regla de $\mathcal{H}$-evaluación para $WHILE$. Esto se repite hasta que la evaluación de la guarda es $False$.

* <u>Finalización del bucle</u>:

$$WHILE \circ FALSE \circ \sigma_0 \circ M_E \circ \sigma_1 \circ M_{S_1} \circ M_S \rightarrow_{eval_{while}} M_S$$

Al terminar el bucle, se continua con el proceso M_S que sigue a éste.

e) <u>Comunicaciones síncronas</u>:

Por comunicación síncrona entendemos aquélla en la que los procesos que intentan establecer comunicación esperan hasta que otro proceso accede al punto de reunión, representado por el canal, con la intención de realizar una operación complementaria.

e.1 <u>Envío de mensajes</u>:

Esta modalidad de envío de mensajes suspende al proceso que la realiza hasta que se produce la $\mathcal{T}$-comunicación con un proceso distinto que esté intentando recibir en el mismo canal. Desglosamos el estudio de las sentencias de *output* en dos casos:

1. <u>Sincronización pura y envío de una constante</u>:

Expresión del LMAN	$-\!> canal; S$ `(internamente` $0-\!> canal; S$`)` $a-\!> canal; S$ `(siendo` a `una constante` $a \in C_{LP}$`)`
Expresión LCEP	$\tau[A-\!> canal; S] = \widehat{canal}\ A \circ \tau[S] \qquad A \in C \cup \{0\}$ (1) (2)

Como puede observarse, la traducción de la sentencia de sincronización pura,

"$->$ *canal*", se apoya en el envío, (1), de un proceso nulo, (2), a través del canal indicado. La sentencia de envío de una constante se considera aquí aparte de la sentencia más general de envío de una expresión, porque puede tratarse de modo simple: no es necesario crear un entorno de ejecución para evaluar la expresión, ya que una constante se evalúa a sí misma.

2. Envío de una expresión:

Expresión del LMAN	$F(R_1, R_2, \ldots, R_m) -> canal; S \quad (0 \le m \le n)$
Expresión LCEP	$\tau[F(R_1, R_2, \ldots, R_m) -> canal; S] =$ $\tau_\lambda[R_1, R_2, \ldots, R_m]\lambda_F F'.(\widehat{_{canal}}(F' \hat{\tau}[F, R_1, R_2, \ldots, R_m]) \circ \tau[S]) \parallel$ $\parallel \tau[\tau_\sigma[R_1, R_2, \ldots, R_m]]$

Se ha empleado $F(R_1, R_2, \ldots, R_m)$ en lugar de E para representar la expresión porque una expresión puede verse como una composición de funciones. Presentamos la "capa externa" de la composición, que permite definir recursivamente la evaluación de una expresión en términos de la evaluación de las subexpresiones que aparecen en ella. Las variables se tratan exactamente igual que las funciones sin parámetros ($n = 0$).

Con R_i, $1 \le i \le m$, se denota la especificación del tipo de parametrización (posicional, etiquetado numérico, etiquetado simbólico) escogida para cada parámetro.

Las funciones τ_λ, $\hat{\tau}$ y τ_σ son funciones auxiliares que permiten definir τ. Están asociadas a las diferentes partes en que se descompone el problema de la evaluación de un subprograma. Vamos a dar su definición y su descripción informal. Para ello, debemos recordar los distintos tipos de parametrización:

$$R_i = \begin{cases} E_i & \text{instanciación posicional o estándar} \\ E_i \to j_i & \text{instanciación numérica } j_i \in \{1, 2, \ldots, n\} \\ E_i \to x_i & \text{instanciación simbólica } x_i \in X \\ & X : \text{conjunto de parámetros formales de } F \end{cases}$$

- τ_σ: Creación de los entornos de ejecución para los parámetros reales.

  Esta función define cómo deben crearse los entornos de ejecución en función de que la expresión asociada a cada parámetro real sea o no una constante:

$$\tau_\sigma[R_i, R_{i+1}, \ldots, R_m] \quad = \begin{cases} \varepsilon & \text{si } m < i \\ \tau_\sigma[R_{i+1}, \ldots, R_m] & \text{si } E_i \in C \\ E_i \to \sigma_i; \tau_\sigma[R_{i+1}, \ldots, R_m] & \text{otro caso} \end{cases}$$

  El significado de la definición es el siguiente:

Si la expresión E_i asociada al parámetro i-ésimo es una constante, no se crea entorno de ejecución. En otro caso, se crea un σ_i-entorno de ejecución para su evaluación, $\sigma_i \in S_S$.

- τ_λ: Obtención del valor evaluado de los parámetros reales.

  Esta función establece la forma en la que llegan los parámetros evaluados en sus respectivos entornos de ejecución, para poder ejecutar el subprograma F instanciado con ellos.

$$\tau_\lambda[R_i, R_{i+1}, \ldots, R_m] \;=\; \begin{cases} \varepsilon & \text{si } m < i \\ \tau_\lambda[R_{i+1}, \ldots, R_m] & \text{si } E_i \in C \\ \lambda_{\sigma_i} \sigma_i'.\tau_\lambda[R_{i+1}, \ldots, R_m] & \text{otro caso} \end{cases}$$

  Comparando la definición de esta función con la anterior se puede ver que lo único que se hace es establecer una operación de *input* asociada a cada etiqueta empleada en τ_σ para definir un entorno de ejecución para el correspondiente parámetro real.

  Las funciones τ_σ y τ_λ describen, por tanto, cómo se implementa la función *eval* en LCEP para obtener la instancia de parametrización sobre la cual aplicar F.

- $\widehat{\tau}$: Parametrización de la aplicación del subprograma.

  Esta función define cómo debe parametrizarse la instancia de parametrización $P = \{P_1, P_2, \ldots, P_m\}$ obtenida para el subprograma. Implementa la función *param* a partir de la información contenida en los parámetros reales R_i.

$$\widehat{\tau}[F, e_1, e_2, \ldots, e_{i-1}, R_i, R_{i+1}, \ldots, R_m] =$$

$$\begin{cases} \varepsilon & \text{si } m < i \\ \circ_{\widehat{P_i}} \; \sigma_i' \; \widehat{\tau}[F, e_1, e_2, \ldots, e_{i-1}, e_i, R_{i+1}, \ldots, R_m] & \text{otro caso} \end{cases}$$

  donde:

$$e_i \;=\; \begin{cases} 1 & \text{si } R_i = E_i \\ j_i & \text{si } R_i = E_i \to j_i \\ x_i & \text{si } R_i = E_i \to x_i \end{cases}$$

  representa cómo son las etiquetas e_i empleadas en cada R_i para especificar la parametrización.

$$P_i \;=\; \begin{cases} 1 & \text{si } e_i = 1 \\ o(j_i) & \text{si } e_i = j_i \\ F_{x_i} & \text{si } e_i = x_i \end{cases}$$

donde $o(j_i) = j_i - card\{k : e_k < j_i \wedge 1 \leq k < i\}$

asocia a cada P_i de la instancia de parametrización P una etiqueta de $\mathcal{L}$, que establece la conexión con los parámetros reales X, definiendo así *param*.

La transformación que se acaba de describir es una de las más importantes que lleva a cabo la función τ, puesto que está relacionada con la definición de los entornos de ejecución para la evaluación de subprogramas. En la presentación de las sentencias anteriores se puede advertir la aparición repetida del proceso $\tau[E \to \sigma]$. Este proceso es el σ-entorno de ejecución de la expresión E. Se puede observar que $E \to \sigma$ es un caso particular de la sentencia $E\text{−>}\ canal; S$ en el cual la sentencia S, que debe ejecutarse a continuación, es la sentencia nula. Por eso, $\tau[S] = 0$ y se habla de la existencia de un túnel que permite el acceso al interior del entorno de ejecución asociado a la etiqueta σ. El canal de comunicación es un canal $\sigma \in S_S$. Este detalle es importante, dado que las propiedades de transparencia del entorno dependen de la relación de orden entre la etiqueta que lo define y las demás etiquetas en $\mathcal{L}$. Una sentencia de este tipo escrita por el programador implica que el canal por él especificado para la comunicación se asocia a una etiqueta simbólica que verifica $canal \in S_G$ o $canal \in S_L$. Pero dichas etiquetas son maximales para cualquier subconjunto de $\mathcal{L}$ que las contenga, es decir, no hay otras etiquetas superiores a ellas. Por tanto, definen un entorno de ejecución no transparente a ninguna comunicación. Por eso no se habla de entornos de ejecución cuando la etiqueta que los define es de esa clase. De hecho, la "posición" del conjunto S_S dentro de la relación de orden en $\mathcal{L}$ ha sido seleccionada para que los entornos de ejecución construidos tengan las propiedades de transparencia necesarias. Por otro lado, la construcción $E\text{−>}\ \sigma$ no es una verdadera sentencia del lenguaje, dado que un programador no puede, por definición, enviar un valor sobre un canal σ. Los canales σ son canales auxiliares creados por el compilador y sólo él puede manipularlos.

e.2 Recepción de mensajes:
Se distinguen dos casos en la recepción de mensajes.

1. Sincronización pura:

Expresión del LMAN	$\text{<−}canal; S$
Expresión LCEP	$\tau[\text{<−}canal; S] = \lambda_{canal} x'.\tau[S]$ (∗)

(∗) $x' \notin FV(\tau[S])$

Como puede observarse, lo único que interesa es que haya una comunicación sobre el canal indicado, pero sin atender al contenido de dicha comunicación. Por ello,

simplemente se desecha el mensaje ($x' \notin FV(\tau[S])$).

2. Recepción de un valor en una variable:

Expresión del LMAN	$x <\!\!-canal; S$ $u <\!\!-canal; S$	
Expresión LCEP	$\tau[x <\!\!-canal; S] =$ $\lambda_{canal} x'.\lambda_{asig} asig'.(asig' \circ x \circ x' \parallel \tau[S])$	x `local`
	$\tau[u <\!\!-canal; S] =$ $\lambda_{canal} u'.(\widehat{_{global}}(u \circ u') \circ \tau[S])$ (*)	u `global`

(*) $x' \notin FV(\tau[S]), u' \notin FV(\tau[S])$

El significado de la traducción es el siguiente: el resultado de la comunicación establecida a través del canal indicado se lleva, mediante la variable ligada x' (u'), a una construcción adecuada en términos de *asig* (*global*) para convertir el valor recibido junto con el identificador de la variable local (global) en un m-proceso.

f) Constructor de paralelismo:

Las definiciones anteriores incluyen el constructor de sentencias secuenciales como parte de las construcciones transformadas mediante τ. De esta forma, se está dando implícitamente la forma que debe tener el proceso asociado a una secuencia de sentencias. Sin embargo, falta por describir en detalle el constructor de *paralelismo*. Su tratamiento es muy sencillo.

Expresión del LMAN	**block** $S_1 \parallel S_2 \parallel \cdots \parallel S_n$ **eblock** $; S$
Expresión LCEP	$\tau[$**block** $S_1 \parallel S_2 \parallel \cdots \parallel S_n$ **eblock** $; S] =$ $\begin{pmatrix} \tau[S_1; 0 \rightarrow c_S] \parallel \tau[S_2; 0 \rightarrow c_S] \parallel \cdots \parallel \tau[S_n; 0 \rightarrow c_S] \\ \lambda_{c_S} x'.\lambda_{c_S} x'. \overset{n}{\cdots} .\lambda_{c_S} x'.\tau[S] \end{pmatrix}$

(*) $x' \notin FV(\tau[S])$. $c_S \in S_L$

El mecanismo de traducción es simple. A cada sentencia se le asocia un proceso, que realiza las acciones determinadas en ella y, al terminar, envía una señal de sincronización (en forma de un proceso nulo 0) sobre un canal c_S, dedicado a la tarea de reunir n comunicaciones procedentes de cada uno de los procesos antes de ejecutar el proceso $\tau[S]$.

La razón de añadir los canales c_S al conjunto de etiquetas simbólicas locales obedece exclusivamente a la necesidad de que las etiquetas c_S no sean comparables con el resto de etiquetas simbólicas, puesto que si lo fueran, se podrían aplicar las reglas de reordenación simbólica (ρ) donde se sitúen los procesos de sincronización, haciendo que el orden de ejecución de la sincronización no sea el adecuado. Las etiquetas en

S_L tienen las propiedades necesarias para garantizar esto, de forma que se toman los canales de entre ellas.

5.2 Ejemplos

Retomamos los problemas resueltos en los Capítulos 2 y 3, ahora utilizando la sintaxis de ALEPH.

5.2.1 Ejemplo 1: El problema de los Filósofos

Veamos la solución en ALEPH, el lenguaje de más alto nivel asociado al λ-cálculo etiquetado paralelo:

```
Program ComidaFilosofos
    Channel
        pos_i, chop_i  ; { Para i = 0,1,...,n-1 }
    Procedure
        PHIL_i  ; PHIL_i1  ; PHIL_i2  ;
        CHOP_i  ; MAID_i  ; { Para i = 0,1,...,n-1 }
    Procedure PHIL_i =
        Block
            <::  pos_i  ; PHIL_i1  ;
        Eblock ;
    Procedure PHIL_i1 =
        Block
            Random
                ?  <::  chop_i  ;
                    Block <:: chop_i⊕1  ; PHIL_i2 ; Eblock ;
                ?  <::  chop_i⊕1  ;
                    Block <:: chop_i  ; PHIL_i2 ; Eblock ;
            Erandom
        Eblock ;
    Procedure PHIL_i2 =
        Block
            Random
                ?  ::>  chop_i  ;
                    Block ::>  chop_i⊕1  ; ::>  pos_i  ;
                        PHIL_i  ; Eblock ;
                ?  ::>  chop_i⊕1  ;
                    Block ::>  chop_i  ; ::>  pos_i  ;
```

```
                    PHILi ; Eblock ;
            Erandom
        Eblock ;
    Procedure CHOPi =
        Block
            ::> chopi ;
                    Block <:: chopi ; CHOPi ; Eblock ;
        Eblock ;
    Procedure MAID0 =
        Block
            Random
                ? ::> pos0 ; MAID1 ;
                ? ::> pos1 ; MAID1 ;
                    ⋮
                ? ::> pos(n-1) ; MAID1 ;
            Erandom
        Eblock ;
    Procedure MAIDi = { Para 1 ≤ i ≤ n-2 }
        Block
            Random
                ? <:: pos0 ; MAID(i-1) ;
                ? <:: pos1 ; MAID(i-1) ;
                    ⋮
                ? <:: pos(n-1) ; MAID(i-1) ;
                ? ::> pos0 ; MAID(i+1) ;
                ? ::> pos1 ; MAID(i+1) ;
                    ⋮
                ? ::> pos(n-1) ; MAID(i+1) ;
            Erandom
        Eblock ;
    Procedure MAID(n-1) =
        Block
            Random
                ? <:: pos0 ; MAID(n-2) ;
                ? <:: pos1 ; MAID(n-2) ;
                    ⋮
                ? <:: pos(n-1) ; MAID(n-2) ;
            Erandom
```

```
        Eblock ;
    BEGIN
        PHIL0 || PHIL1 || ··· PHILn ||
        CHOP0 || CHOP1 || ··· CHOPn ||
        MAID0
    END.
```

5.2.2 Ejemplo 2: Teléfonos móviles

Veamos la solución general al problema usando ALEPH, el lenguaje de más alto nivel asociado al λ-cálculo etiquetado paralelo:

```
Program COCHES
    Channel t1, t2, s1, s2, g1, g2, a1, a2 ;
    Procedure CAR1 =
        Block
            Random
                ? <:: t1 ; CAR1 ;
                ? <:: s1 ;
                    Block
                        <:: t2 ; <:: s2 ; CAR2 ;
                    Eblock
            Erandom
        Eblock
    Procedure CAR2 =
        Block
            Random
                ? <:: t2 ; CAR2 ;
                ? <:: s2 ;
                    Block
                        <:: t1 ; <:: s1 ; CAR1 ;
                    Eblock
            Erandom
        Eblock
    Procedure BASE1 =
        Block
            Random
                ? ::> t1 ; BASE1 ;
                ? <:: g1 ;
                    Block
```

```
                    <::  t2 ; <::  s2 ; ::> s1 ;
                    ::> t2 ; ::> s2 ; IDLEBASE1 ;
                Eblock
        Erandom
    Eblock
Procedure BASE2 =
    Block
        Random
            ? ::> t2 ; BASE2 ;
            ? <::  g2 ;
                Block
                    <::  t1 ; <::  s1 ; ::> s2 ;
                    ::> t1 ; ::> s1 ; IDLEBASE2 ;
                Eblock
        Erandom
    Eblock
Procedure IDLEBASE1 =
    Block
        <::  a1 ; BASE1 ;
    Eblock
Procedure IDLEBASE21 =
    Block
        <::  a2 ; BASE2 ;
    Eblock
Procedure CENTRE1 =
    Block
        ::> g1 ;
            Block
                ::> t2 ; ::> s2 ;
                ::> a2 ; CENTRE2 ;
            Eblock
    Eblock
Procedure CENTRE2 =
    Block
        ::> g2 ;
            Block
                ::> t1 ; ::> s1 ;
                ::> a1 ; CENTRE1 ;
            Eblock
```

```
    Eblock
BEGIN
    CAR₁  || BASE₁  || IDLEBASE₁  ||  CENTRE₁
END.
```

Capítulo 6

Conclusiones y trabajo futuro

En esta tesis se presenta un nuevo cálculo (LCEP), extensión directa del λ-cálculo, para modelizar la concurrencia y la comunicación entre procesos. LCEP (λ-cálculo Etiquetado Paralelo) surge a partir de una propuesta inicial de Hassan Aït-Kaci, el *Label-selective λ-calculus*, que describe un lenguaje, extensión del λ-cálculo de Church, en el que los argumentos de las funciones se seleccionan mediante etiquetas. El conjunto de etiquetas es la unión disjunta de dos conjuntos, un conjunto numérico y otro de símbolos. En la literatura existen numerosas propuestas para modelizar la concurrencia, pero el lenguaje LCEP constituye la única que puede entenderse como una extensión conservativa del λ-cálculo, en el sentido de que, cuando el conjunto de etiquetas es el conjunto unario $\{1\}$ y se eliminan los operadores de paralelismo, LCEP coincide exactamente con el λ-cálculo, tal y como se concibió.

Se describen las reglas del nuevo cálculo y se presenta su semántica operacional siguiendo la aproximación estándar, que pasa por definir la semántica de los sistemas concurrentes por medio del conjunto de experimentos que los sistemas ofrecen a un observador. Para ello usamos el modelo de los sistemas de transición etiquetados de Plotkin [Plo81].

Por otro lado, un objetivo importante del presente trabajo ha sido la obtención de un lenguaje que permite la construcción de programas de una forma útil y apropiada. LCEP resulta demasiado complejo en la mayoría de los casos como para pensar en él como un recurso expresivo para la programación. Por esta razón, se ha definido un lenguaje de programación de más alto nivel (ALEPH), que permite aprovechar la potencia expresiva del formalismo y para ello utiliza el cálculo como código máquina al que se traduce el programa. Se ha construido el traductor de código ALEPH a código LCEP, mecanizando así la ejecución del lenguaje. De este modo, disponemos de un producto que facilita la construcción de programas en un lenguaje de alto nivel estándar y su ejecución utiliza la potencia del formalismo.

Finalmente, se ha utilizado la *relación de reescritura sensible al contexto* [Luc95b, Luc95a, Luc96] para dar una caracterización alternativa al comportamiento operacional de los programas sin pérdida de completitud.

Las líneas de trabajo futuro abiertas por esta tesis son varias. La más importante, en principio, es la extensión del lenguaje para introducir características de la programación orientada a objetos. Actualmente se encuentra ya en desarrollo dentro de nuestro grupo y forma parte de los objetivos de un proyecto de investigación coordinado, en el que participan también la Universidad Complutense de Madrid y la Universidad de Málaga.

Otra extensión significativa al trabajo realizado, consiste en el desarrollo de un estudio formal comparativo, utilizando diversos mecanismos conocidos, entre las diferentes propuestas presentadas en el Capítulo 1 y el cálculo LCEP. Dicho estudio contribuiría a clarificar las similitudes y diferencias existentes entre ellos.

También puede resultar muy positivo para el futuro del entorno presentado continuar la formalización y el desarrollo de los grafos de $\mathcal{T}$-comunicabilidad de un proceso, como un mecanismo para estudiar las posibilidades de comunicación efectivas de un sistema desde un punto de vista muy intuitivo.

En cuanto al lenguaje de alto nivel ALEPH, sería muy útil extender su expresividad para permitir el tratamiento de tipos de datos.

Por último, es interesante profundizar en el estudio detallado de las equivalencias que se pueden construir a partir de las estructuras del lenguaje, haciendo uso de las diferentes nociones de bisimulación y observabilidad introducidas en los últimos años.

Bibliografía

[Abr89] S. Abramsky. The Lazy λ-Calculus. In D. Turner, editor, *Research Topics in Functional Programming*, pages 65–116. Addison-Wesley, 1989.

[ACCL90] M. Abadi, L. Cardelli, P.L. Curien, and J.J. Lévy. Explicit substitutions. *In Proc. of ACM Symposium on Principles of Programming Languages*, pages 31–46, 1990.

[Agh90] G. Agha. *ACTORS: A Model of Concurrent Computation in Distributed Systems*. The MIT Press, Cambridge, MA., 1990.

[AH86] G. Agha and C. Hewitt. Concurrent Programming using ACTORS. 1986.

[AHU74] A.V. Aho, J.E. Hopcroft, and J.D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison Wesley, 1974.

[AKG93a] H. Aït-Kaci and J. Garrigue. Label-Selective λ-Calculus. Technical report, PRL 31, Digital Equipment Corporation, France, May 1993.

[AKG93b] H. Aït-Kaci and J. Garrigue. Label-Selective λ-Calculus: Syntax and Confluence. In *Proc. of the 13th Int. Conf. on Foundations of Software Technology and Theoretical Computer Science*, volume 761 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, 1993.

[AKG93c] H. Aït-Kaci and J. Garrigue. The Typed Polymorphic label-Selective λ-Calculus. Technical report, PRL, Digital Equipment Corporation, France, September 1993.

[AL88] F. Alonso Amo and A. Morales Lozano. *Técnicas de programación*. Paraninfo, 1988.

[All90] J. Allen. *Anatomy of LISP*. McGraw-Hill, 1990.

[Ama92] R. Amadio. A uniform presentation of CHOCS and π-calculus. Technical report, 1726, INRIA-Lorraine, Nancy, 1992.

[Ama93] R. Amadio. On the reduction of CHOCS bisimulation to π-calculus bisimulation. In E. Best, editor, *Proc. of CONCUR 93*, volume 715 of *Lecture Notes in Computer Science*, pages 112–126. Springer-Verlag, Berlin, 1993.

[ASU90] A.V. Aho, R. Sethi, and J.D. Ullman. *Compiladores. Principios, técnicas y herramientas.* Addison Wesley, 1990.

[AZ84] E. Astesiano and E. Zucca. Parametric channels via label expressions in CCS. *Theoretical Computer Science*, 33:45–64, 1984.

[Bar87] J.G.P. Barnes. *Programación en ADA.* Díaz de Santos, 1987.

[Bar91] H.P. Barendregt. *The λ-calculus, its Syntax and Semantics*, volume 103 of *Studies in Logic and the Foundations of Mathematics.* Elsevier Science Publishers, 1984 edition, Amsterdam, 1991.

[BB93] G. Berry and G. Boudol. The Chemical Abstract Machine. *In Proc. of 20'th ACM Annual Symp. on Principles of Programming Languages*, pages 81–93, ACM Press, 1993.

[Ber78] G. Berry. Séquentialité de l'évaluation formelle des λ-expressions. *In Proc. of 3-e Colloque International sur la Programmation*, Dunod, Paris 1978.

[Ber83] C. Berge. *Graphes.* Ed. Gauthier-Villars, 1983.

[BK85] J.A. Bergstra and J.W. Klop. *Algebra for Communicating Processes with Abstractions*, volume 37, pages 77–121. Journal of Theoretical Computer Science, 1985.

[BM86] J.P. Banâtre and D. Le Metayer. A New Computational Model and Its Discipline of Programming. Technical report, INRIA 566, Sophia Antipolis, France, 1986.

[BM90] J.P. Banâtre and D. Le Metayer. *The Gamma model and its discipline of programming*, volume 15 (1), pages 55–77. Science of Computer Programming, 1990.

[BN92] M. Boreale and R. De Nicola. Testing equivalences for mobile processes. In R. Cleaveland, editor, *Proc. of CONCUR 92*, volume 630 of *Lecture Notes in Computer Science*, pages 2–16. Springer-Verlag, Berlin, 1992.

[BN94] M. Boreale and R. De Nicola. A fully abstract semantics for causality in the π-calculus. Technical report, ECS-LFCS-94-297, Laboratory for Foundations of Computer Science, Departament of Computer Science, University of Edimburgh, UK, 1994.

[Bou89] G. Boudol. Towards a λ-calculus for Concurrent and Communicating Systems. In *Proc. of TAPSOFT'89*, volume 351 of *Lecture Notes in Computer Science*, pages 149–161. Springer-Verlag, Berlin, 1989.

[Bou90] G. Boudol. A λ-Calculus for Parallel Functions. Technical report, INRIA 1231, Sophia Antipolis, France, 1990.

[Bou94] G. Boudol. Some chemical abstract machines. In *Proc. of the REX School/Workshop A Decade of Concurrency*, Lecture Notes in Computer Science, pages 92–123. Springer-Verlag, Berlin, 1994.

[Chu32] A. Church. *A Set of Postulates for the Foundation of Logic*, volume 2 of *Annals of Math*, pages 346–366. 1932.

[CLO96] L. Climent, M.L. Llorens, and J. Oliver. Building an interpreter for label-selective λ-calculus. In B. Clares, editor, *Proc. of II Jornadas de Informática*, pages 325–334. Asociación Española de Informática y Automática, Almuñecar (Spain), 1996.

[CNR92] M. Cosnard, M. Nivat, and Y. Robert. *Algorithmique parallèle*. Ed. Masson, 1992.

[CO95] L. Climent and J. Oliver. El λ-cálculo Seleccionado por Etiquetas. Desarrollo de una Máquina Abstracta en Prolog. Technical Report DSIC-II/15/95, UPV, 1995.

[Cur58] H.B. Curry. *Combinatory Logic*, volume I of *Studies in Logic*. North-Holland, Amsterdam, 1958.

[Cur72] H.B. Curry. *Combinatory Logic*, volume II of *Studies in Logic*. North-Holland, Amsterdam, 1972.

[Dam93] M. Dam. Model checking mobile processes. In E. Best, editor, *Proc. of CONCUR 93*, volume 715 of *Lecture Notes in Computer Science*, pages 22–36. Springer-Verlag, Berlin, 1993.

[dB72] N.G. de Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation. *Indag. Math.*, pages 34:381–392, 1972.

[dBP91a] F.S. de Boer and C. Palamidessi. Embedding as a tool for language comparison. Technical report, CS-R9102, Centre for Mathematics and Computer Science (CWI), Amsterdam, 1991.

[dBP91b] F.S. de Boer and C. Palamidessi. Embedding as a tool for language comparison: On the CSP hierarchy. In J.C.M. Baeten and J.F. Groote, editors, *Proc. of CONCUR 91*, volume 527 of *Lecture Notes in Computer Science*, pages 127–141. Springer-Verlag, Berlin, 1991.

[Der93] N. Dershowitz. A Taste of Rewrite Systems. In P. E. Lauer, editor, *Proc. of Functional Programming, Concurrency, Simulation and Automated Reasoning*, volume 693 of *Lecture Notes in Computer Science*, pages 199–228. Springer-Verlag, Berlin, 1993.

[Dij68] E.W. Dijkstra. Cooperating Sequential Processes. In F. Genus, editor, *Programming Languages*, pages 43–112. Academic Press, London, 1968.

[DJ90] N. Dershowitz and J.P. Jouannaud. Rewrite Systems. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B: Formal Models and Semantics, pages 243–320. Elsevier, Amsterdam and the MIT Press, Cambridge, MA., 1990.

[EM85] H. Ehrig and B. Mahr. *Fundamentals of Algebraic Specification*, volume 6 of *EATCS Monographs on Theoretical Computer Science*. Springer-Verlag, Berlin, 1985.

[EN86] U. Engberg and M. Nielsen. A Calculus of Communicating Systems with Label-passing. Technical report, DAIMI PB-208, University of Aarhus, 1986.

[Eng93] J. Engelfriet. A multiset semantics for the π-calculus with replication. In E. Best, editor, *Proc. of CONCUR 93*, volume 715 of *Lecture Notes in Computer Science*, pages 37–51. Springer-Verlag, Berlin, 1993.

[GS89] J. Gallier and W. Snyder. Designing unification procedures using transformations: a survey. In Y.N. Moschovakis, editor, *Logic from Computer Science*, pages 153–215. Springer-Verlag, Berlin, 1989.

[HB88] K. Hwang and F.A. Briggs. *Arquitectura de computadoras y procesamiento paralelo*. McGraw-Hill, 1988.

[Hen91] M. Hennessy. A model for the π-calculus. Technical report, 8/91, School of Cognitive and Computing Sciences, University of Sussex, UK, 1991.

[HM85] M. Hennessy and R. Milner. *Algebraic Laws for Non-determinism and Concurrency*, volume 32, pages 137–161. Journal of ACM, 1985.

[HO95] C. Herrero and J. Oliver. Construcción de una Máquina Química Abstracta. In J.M. Troya and C. Rodríguez, editors, *Proc. of I Jornadas de Informática*, pages 121–134, Tenerife (Spain), 1995.

[Hoa85] C.A.R. Hoare. *Communicating Sequential Processes.* Prentice Hall International, Great Britain, 1985.

[Kle36] S.C. Kleene. λ-definability and Recursiveness. *Duke Math*, pages 340–353, 1936.

[KR35] S.C. Kleene and J.B. Rosser. The Inconsistency of certain Formal Logics. *Annals of Math*, (2)36:630–636, 1935.

[Lan65] P.J. Landin. A correspondence between ALGOL 60 and Church's lambda notation. *Communications of the ACM*, 8:89–101, 158–165, 1965.

[Liu94] X. Liu. Characterizing bisimulation congruence in the π-calculus. In J. Parrow and B. Jonsson, editors, *Proc. of CONCUR 94*, volume 836 of *Lecture Notes in Computer Science*, pages 331–350. Springer-Verlag, Berlin, 1994.

[LO94a] S. Lucas and J. Oliver. Context-sensitive rewriting. Technical Report DSIC-II/23/94, UPV, 1994.

[LO94b] S. Lucas and J. Oliver. Parallel label-selective λ-calculus (LCEP). In M. Alpuente, R. Barbuti, and I. Ramos, editors, *Proc. of 1994 Joint Conference on Declarative Programming GULP-PRODE'94*, pages 125–139, 1994.

[LO95] M.L. Llorens and J. Oliver. El λ-cálculo Seleccionado por Etiquetas. Desarrollo de una Máquina Abstracta en Pascal. Technical Report DSIC-II/16/95, UPV, 1995.

[LO96] S. Lucas and J. Oliver. A new proposal of concurrent process calculus. In K.G. Jeffery and J. Kral, editors, *XXIII International Winter School on Theoretical and Practical Aspects of Computer Science, SOFSEM'96*, Lecture Notes in Computer Science. Springer-Verlag, Berlin, 1996. To appear.

[Luc95a] S. Lucas. Computational properties in context-sensitive rewriting. In M. Alpuente and M. Sessa, editors, *Proc. of 1995 Joint Conference on Declarative Programming GULP-PRODE'95*, pages 435–446, 1995.

[Luc95b] S. Lucas. Fundamentals of context-sensitive rewriting. In J. Staudek M. Bartosek and J. Wiedermann, editors, *Proc. of the XXII Seminar on*

Current Trends in Theory and Practice of Informatics, SOFSEM'95, volume 1012, pages 405–412. Springer-Verlag, Berlin, 1995.

[Luc96] S. Lucas. Termination of context-sensitive rewriting by rewriting. In Friedhelm Meyer auf der Heide, editor, *Proc. of the 23rd International Colloquium on Automata, Languages, and Programming, ICALP'96*. Springer-Verlag, Berlin, 1996. To appear.

[Mes92] J. Meseguer. Conditional Rewriting Logic as a Unified Model of Concurrency. *In Journal of Theoretical Computer Science*, 96:73–155, 1992.

[Mil89] R. Milner. *Communication and Concurrency.* Prentice Hall International, Great Britain, 1989. ISBN:0-13-114984-9.

[Mil92a] R. Milner. Action structures. Technical report, ECS-LFCS-92-249, Laboratory for Foundations of Computer Science, Department of Computer Science, University of Edimburg, UK, 1992.

[Mil92b] R. Milner. Action structures and polynomial π-calculus. In E. Best, editor, *Proc. of a CALIBAN-Workshop, Sheffield*, pages 77–105. Universität Hildesheim, 1992.

[Mil92c] R. Milner. Functions as Processes. Research Report 1154, Sofia Antipolis, 1990. *Extended version in Journal of Mathematical Structures in Computer Science*, 2(2):119–141, 1992.

[Mil93a] R. Milner. Action calculi and the π-calculus. In *Proc. of the NATO Summer School on Logic and Computation*. Springer-Verlag, Berlin, 1993.

[Mil93b] R. Milner. An action structure for the synchronous π-calculus. In Z. Ésik, editor, *Proc. of FCT'93*, volume 710 of *Lecture Notes in Computer Science*, pages 87–105. Springer-Verlag, Berlin, 1993.

[Mil93c] R. Milner. Elements of Interaction. Turing Award Lecture. *Communications of the ACM*, 36(1), January 1993.

[Mil93d] R. Milner. The polyadic π-calculus: A tutorial. In F.L. Brauer, W. Bauer, and H. Schwichtenberg, editors, *Logic and Algebra of Specifications*. Springer-Verlag, Berlin, 1993.

[MP94] U. Montanari and M. Pistore. Concurrent semantics for the π-calculus. Technical report, Università di Pisa, Italy, November 1994.

[MPW92] R. Milner, J. Parrow, and D. Walker. A Calculus of Mobile Processes, Parts I and II. *Information and Computation*, 100:1–77, 1992.

[Nie96] J. Niehren. Functional computation as concurrent computation. In *Proc. of 23rd ACM Symposium on Principles of Programming Languages.* ACM Press, St. Petersburg Beach, Florida, 1996.

[OL94] J. Oliver and S. Lucas. Survey of Concurrent Calculi. In M.A. Fernández, J.M. García, J.A. Guerrero, and G. Moreno, editors, *Nuevas Tendencias en la Informática: Arquitecturas Paralelas y Programación Declarativa*, pages 269–289, 1994.

[Oli95] J. Oliver. Extensiones del λ-cálculo para modelizar la concurrencia. *Boletín AEPIA*, pages 18–21, 1995.

[OP90] F. Orava and J. Parrow. Algebraic description of mobile networks: an example. In L. Logrippo, R. Probert, and H. Ural, editors, *Proc. of PSTV'X*, pages 275–291. North-Holland, 1990.

[OP92] F. Orava and J. Parrow. An algebraic verification of a mobile network. *Journal of Formal Aspects of Computing*, 4:497–543, 1992.

[Pet62] C.A. Petri. Fundamentals of a Theory of Asynchronous Information Flow. *In Proc. of IFIP Congress'62*, pages 386–390, North Holland 1962.

[Pie93] B. Pierce. Programming in the π-calculus. A Case Study in Programming Language Design. Technical Report Preliminary Draft, Dept. of Computer Science, Edinburgh University, 1993.

[Plo75] G.D. Plotkin. Call-by-name, call-by-value and the λ-calculus. *Theoretical Computer Science*, 1:125–159, 1975.

[Plo81] G.D. Plotkin. A Structural Approach to Operational Semantics. Technical Report DAIMI FN-19, Computer Science Dept., University of Aarhus, 1981.

[Pra86] V.R. Pratt. Modeling concurrency with partial orders. *International Journal of Parallel Programming*, 15(1):33–71, 1986.

[Pre90] R.S. Pressman. *Ingeniería del software. Un enfoque práctico.* McGraw-Hill, 1990.

[PS92] B. Pierce and D. Sangiorgi. Typing and Subtyping for Mobile Processes. *Extended Abstract*, December 1992.

[San92] D. Sangiorgi. The Lazy Lambda Calculus in a Concurrent Scenario. *In Proc. of the seventh annual IEEE Symposium on Logic in Computer Science*, 1992.

[San93a] D. Sangiorgi. A theory of bisimulation for the π-calculus. In E. Best, editor, *Proc. of CONCUR 93*, volume 715 of *Lecture Notes in Computer Science*, pages 127–142. Springer-Verlag, Berlin, 1993.

[San93b] D. Sangiorgi. An Investigation into Functions as Processes. *In Proc. Math. Foundations of Program Semantics*, New Orleans, April 1993.

[Sco90] G. Scollo. Abstract data types. *Course Notes. Universiteit Twente. Faculteit Informatica*, 1990.

[Smu71] R. Smullyan. *Theory of Formal Systems.* Princeton University Press, N.J., 1971. Revised edition.

[Ste84] G.L. Steele. *Common LISP, The Language.* Digital Press, 1984.

[Sto88] A. Stoughton. Substitution Revisited. volume 59 of *Theoretical Computer Science*, pages 317–325, 1988.

[Tho89] B. Thomsen. A Calculus of Higher Order Communicating Systems. *In Proc. of 16th ACM Annual Symp. on Principles of Programming Languages*, pages 143–154, ACM Press, 1989.

[Tho93] B. Thomsen. Plain CHOCS. A second generation calculus for higher order processes. *Acta Informatica*, (1) 30:1–59, 1993.

[Tho95] B. Thomsen. A Theory of Higher Order Communicating Systems. *Journal of Information and Computation*, (1) 116:38–57, 1995.

[Tur37] A.M. Turing. Computability and λ-definability. *Symbolic Logic*, 2:153–163, 1937.

[Wal94] D. Walker. On bisimulation in the π-calculus. In J. Parrow and B. Jonsson, editors, *Proc. of CONCUR 94*, volume 836 of *Lecture Notes in Computer Science*, pages 315–330. Springer-Verlag, Berlin, 1994.

Apéndice A

Un lenguaje con paralelismo explícito

La escritura directa de programas en LCEP resulta demasiado compleja, en la mayoría de los casos, como para pensar en él como recurso expresivo adecuado para la programación. Por este motivo, se ha definido un lenguaje de más alto nivel (ALEPH), descrito en el Apéndice B, que posee los recursos expresivos deseables en programación y que permite aprovechar la potencia del formalismo.

Como modelo de referencia, describimos en este apéndice la clase de lenguajes imperativos orientados a la programación estructurada, por ser ampliamente extendidos y utilizados. Se analizan sus características básicas en cuanto lenguajes de programación secuencial. Finalmente, se contemplan las extensiones realizadas sobre el esquema secuencial para soportar el procesamiento paralelo.

A.1 Esquema de un lenguaje estructurado

En un lenguaje de programación imperativo, un programa establece explícitamente cómo se produce el resultado deseado, pero no define de forma explícita las propiedades que se esperan de ese resultado[1]. El mecanismo para producir el resultado deseado adopta la forma de una secuencia de instrucciones a la máquina abstracta para que manipule los datos de manera adecuada. El modelo computacional de estos lenguajes se enmarca dentro del modelo de *control y manipulación de datos*. Los ordenadores existentes siguen, en su mayoría, la arquitectura von Neumann, que recoge las características de este modelo. Esta es la razón de la elección de un lenguaje imperativo

[1] Por contra, en un lenguaje de programación declarativo, cobra mayor importancia la descripción, por parte del programa, de las propiedades del resultado que se desea obtener. Se deja a la máquina abstracta que ejecuta el programa la tarea de resolver cómo obtener dicho resultado.

como lenguaje de más alto nivel asociado a LCEP.

a) *Lenguajes imperativos.* En los lenguajes imperativos son esenciales los siguientes conceptos:

- Instrucción: la unidad básica de manipulación de la máquina abstracta. Entre ellas, la instrucción de *asignación* juega un papel fundamental como enlace entre la parte de control y la de almacenamiento de la máquina abstracta.
- Orden de ejecución: da sentido al programa, y consta de un conjunto de instrucciones.
- Flujo de control: el medio de establecer el orden de ejecución de las instrucciones.

b) *Programación estructurada.* En un lenguaje orientado a la programación estructurada [AL88, Pre90], los datos que manejan los programas adquieren una determinada estructura, y los programas que los manipulan se diseñan en función de la estructura que adoptan los datos. Se definen tres formas básicas para construir datos compuestos a partir de datos elementales:

- Como una secuencia de datos.
- Como una selección de entre un conjunto de datos.
- Como la agrupación repetida de elementos de datos.

Otro aspecto importante surge de la posibilidad de descomponer el tratamiento del problema global en subproblemas, que pueden ser resueltos de forma separada, y cuyas soluciones pueden agruparse para dar respuesta al problema inicial: el diseño procedural [Pre90].

c) *Orden de ejecución.* Es importante señalar que, para describir lo que habitualmente se entiende por orden de ejecución de un programa, es preciso distinguir entre las instrucciones de un programa imperativo y las operaciones realizadas por la máquina abstracta que ejecuta las sentencias del programa. Se debe considerar cada ejecución de una misma instrucción del programa como una entidad distinta, es decir, una *operación* [CNR92]. El orden de ejecución de un programa es el orden de ejecución de las operaciones derivadas de las sentencias del programa. Si se representa una instrucción elemental del lenguaje de programación como $r, s, \ldots$, las operaciones asociadas a cada una como $\omega_r, \omega_s, \ldots$ y el orden de ejecución de las operaciones como $\prec$, entonces, la sentencia compuesta $S = r; s$ permite escribir $\omega_r \prec \omega_s$, significando que la operación asociada a r se ejecuta antes que la asociada a s. La relación de orden $\prec$ es, en general, un orden parcial. En programas secuenciales ejecutados en

una máquina secuencial[2], $\prec$ es un orden total. Cuando el lenguaje de programación permite expresar paralelismo, es posible construir programas en los que no siempre se puede forzar el orden de ejecución de las operaciones asociadas a las instrucciones.

Vamos a describir brevemente los elementos sintácticos de que se componen los lenguajes de programación imperativos orientados a la programación estructurada. Junto con una descripción de carácter general, se presenta la notación empleada para denotar cada elemento.

A.2 Declaraciones

Muchos lenguajes de programación actuales, incorporan la posibilidad de establecer en los programas determinadas propiedades de las entidades manejadas, en el marco de una fase de declaración de dichas entidades. En la fase de declaraciones de un programa:

- Se asocia un *nombre* con determinadas entidades que el lenguaje de programación pone a disposición del programador.
- Se establecen, explícita o implícitamente, las propiedades estáticas (tipo, tamaño) y dinámicas (alcance, modificabilidad) que se atribuyen a la entidad identificada con el nombre indicado.

a) Entidades descritas mediante declaraciones:

- Posibilidad de definición de *tipos* a partir de determinados tipos básicos y constructores de tipo ofrecidos por el lenguaje. Se puede ver la definición de tipos como la implementación de las formas básicas de definición de datos en el modelo estructurado.
- Posibilidad de definición de *constantes*, entendidas como objetos de un determinado tipo que no experimentan variaciones durante la ejecución del programa.
- Posibilidad de definición de *variables*, entendidas como objetos de un determinado tipo, susceptibles de sufrir variaciones en sus contenidos durante la ejecución del programa[3].
- Definición de *subprogramas*. Posteriormente se describen en detalle.

[2] Si se ejecuta un programa secuencial en una máquina paralela es posible utilizar un compilador paralelizador que sea capaz de aprovechar los recursos de la máquina donde se ejecuta el programa para hacerlo de manera más eficiente.

[3] Tanto en el caso de las variables como en el de los tipos, la forma de acceso a los objetos, el dominio de valores que pueden tomar, ... vienen determinados por el tipo que ha sido asignado a cada objeto.

b) Declaraciones y modelo de computación:

Mediante el conjunto de variables se representa la memoria de datos de la máquina sobre la que trabajamos. En ALEPH, vamos a suponer que todos los objetos declarados pueden tratarse de manera uniforme. Por esta razón, no vamos a considerar el problema de los tipos. Suponemos un conjunto de constantes en el lenguaje de programación, que denotamos por C_{LP}.

A.3 Subprogramas: funciones y procedimientos

El término subprograma [Pre90] se refiere a un componente de un programa que contiene datos y estructuras de control.

a) Características generales de los subprogramas:

- Una sección de especificación, que incluye su nombre y la descripción de su interfaz[4].
- Una sección de implementación, que incluye los datos y las estructuras de control.
- Un mecanismo de activación, que permite que el subprograma sea requerido desde cualquier punto del programa donde sea visible.

b) Los subprogramas como elementos de programa:

La siguiente notación:

NOTACION:

$$\begin{array}{l}\textbf{def}\ (x_1, x_2, \ldots, x_n) = \\ \textbf{local}\ x, y, \ldots, z; \\ S_F(x_1, x_2, \ldots, x_n, x, y, \ldots, z, u, v, \ldots, w)\end{array}$$

representa la definición de un subprograma F con parámetros formales (interfaz) $x_1, x_2, \ldots, x_n$, $(n \geq 0)$ que maneja objetos locales $x, y, \ldots, z$. Las sentencias que implementan el subprograma (cuerpo del subprograma) se representan como $S_F(x_1, x_2, \ldots, x_n, x, y, \ldots, z, u, v, \ldots, w)$. Dichas sentencias pueden acceder tanto a los parámetros del subprograma como a las variables locales. La presencia de las variables $u, v, \ldots, w$ significa la posibilidad adicional de manipulación de variables globales en el cuerpo del subprograma. Se puede tratar el bloque de parámetros formales de un subprograma como un conjunto $X = \{x_1, x_2, \ldots, x_n\}$. X se puede considerar como un conjunto totalmente ordenado por el orden $<<$ de aparición de los parámetros en la declaración del subprograma: $x_i << x_j \Longleftrightarrow i < j$.

[4]Como interfaz de un subprograma se entiende una lista de parámetros (formales). Cada parámetro es un par $\langle nombre, tipo \rangle$. Por tanto, especificar la lista de parámetros formales de un subprograma no es más que dar una lista de declaraciones de variables.

A.4 Activación de un subprograma

La *activación* de un subprograma consiste en ejecutar las sentencias incluidas en el cuerpo del mismo para una instancia de los parámetros formales, que se obtienen a partir de los parámetros reales especificados en la sentencia de llamada a subprograma. El problema de la parametrización de un subprograma consiste en definir cómo relacionar el conjunto de parámetros reales y el conjunto de parámetros formales. Esto se hace en dos pasos:

1. Obtención de la instancia de parametrización del subprograma a partir de los parámetros reales.

2. Asociación de la instancia de parametrización del subprograma con los parámetros formales.

NOTACION:

$$F(R_1, R_2, \ldots, R_m)$$

representa la llamada al subprograma F con parámetros reales $R_1, R_2, \ldots, R_m$. Si n es el número de parámetros formales establecidos en la declaración de F, debe cumplirse en general que $m \leq n$. R_i representa una expresión. Se puede considerar que los parámetros reales definen un conjunto finito $R = \{R_1, R_2, \ldots, R_m\}$ (que puede estar vacío). También se puede considerar R ordenado por el orden $<<$ de aparición de los parámetros reales en la llamada a subprograma. El problema de la parametrización se describe como sigue:

1. El mecanismo de evaluación transforma un conjunto de parámetros reales R en una instancia de parametrización $P = \{P_1, P_2, \ldots, P_m\}$, mediante la función biyectiva $eval : R \rightarrow P$. Asumimos, además, que $eval(R_i) = P_i$.

2. Establecer un modelo de parametrización de un subprograma consiste en dar una función inyectiva $param : P \rightarrow X$, que determina la asociación entre la instancia de parametrización P y el conjunto de parámetros formales X.

A.4.1 Modelos de parametrización

Se distinguen dos aspectos fundamentales:

a) Según la extensión de la parametrización:

- Parametrización total. Se especifican parámetros reales para cada parámetro formal; es decir, la función *param* es biyectiva. El número m de parámetros reales coincide con el número n de parámetros formales. El objeto resultante de la parametrización es ejecutable.

- Parametrización parcial. Se especifica sólo parte de los parámetros que necesita el subprograma para ser ejecutado. Se tiene entonces que $m < n$. El objeto resultante de la parametrización no es ejecutable, define en realidad un nuevo subprograma cuyo significado queda englobado en el subprograma original.

b) Según la forma de asociar los parámetros reales a los parámetros formales:

- Instanciación posicional. Es la forma habitual de parametrizar un subprograma. El orden $<<$ de los parámetros reales en la llamada a subprograma coincide con el orden de los parámetros formales en la declaración. Es decir, $param(P_i) = x_i$ para $1 \leq i \leq m$.

- Instanciación etiquetada numérica[5]. A cada parámetro real se le asocia una etiqueta numérica que lo relaciona con un parámetro formal, que especifica el orden que ocupa el parámetro real en la lista de parámetros formales de la declaración del subprograma. Por lo tanto, $param(P_i) = x_j$ para $1 \leq i \leq m$ y algún j, $1 \leq j \leq n$.

- Instanciación etiquetada simbólica. Análogamente al caso anterior, a cada parámetro real se le hace corresponder el nombre del parámetro formal en la declaración del subprograma, $param(P_i) = x_j$ para $1 \leq i \leq m$ y algún j, $1 \leq j \leq n$.

El empleo de una u otra de estas modalidades de asociación de parámetros reales a parámetros formales, viene indicado por la propia sintaxis de la llamada a subprograma. Para ello, se introduce la siguiente notación:

[5]En ADA se emplea la expresión *instanciación nombrada* [Bar87] para referirse a la instanciación etiquetada simbólica, única permitida en ese lenguaje además de la posicional; no obstante, se introduce esta terminología, que resulta más específica y cercana al tratamiento formal realizado.

NOTACION:

Tipo de instanciación	Sintaxis llamada a subprograma
`Posicional`	$F(E_1, E_2, \ldots, E_m)$
`Etiquetada numérica`	$F(E_1 {-}{>}\, j_1, E_2 {-}{>}\, j_2, \ldots, E_m {-}{>}\, j_m)$
`Etiquetada simbólica`	$F(E_1 {-}{>}\, x_{j_1}, E_2 {-}{>}\, x_{j_2}, \ldots, E_m {-}{>}\, x_{j_m})$

donde E_i representa una expresión, con $1 \leq j_i \leq n$ para $1 \leq i \leq m$.

Como puede advertirse, para cada parámetro real R_i, $1 \leq i \leq m$, se define, en la llamada a subprograma, tanto la expresión E_i que va a servir para calcular la instancia de parametrización P_i correspondiente mediante *eval*, como la función *param* que asocia el parámetro real a un parámetro formal. Esto se resume en la siguiente tabla:

Sintaxis de la llamada a subprograma $F(R_1, R_2, \ldots, R_m)$	**Función** *eval*
$F(E_1, E_2, \ldots, E_m)$	$eval(E_i) = eval(E_i) = P_i$
$F(E_1 {-}{>}\, j_1, E_2 {-}{>}\, j_2, \ldots, E_m {-}{>}\, j_m)$	$eval(E_i {-}{>}\, j_i) = eval(E_i) = P_i$
$F(E_1 {-}{>}\, x_{j_1}, E_2 {-}{>}\, x_{j_2}, \ldots, E_m {-}{>}\, x_{j_m})$	$eval(E_i {-}{>}\, x_{j_i}) = eval(E_i) = P_i$

Sintaxis de la llamada a subprograma $F(R_1, R_2, \ldots, R_m)$	**Función** *param*
$F(E_1, E_2, \ldots, E_m)$	$param(P_i) = x_i$
$F(E_1 {-}{>}\, j_1, E_2 {-}{>}\, j_2, \ldots, E_m {-}{>}\, j_m)$	$param(P_i) = x_{j_i}$
$F(E_1 {-}{>}\, x_{j_1}, E_2 {-}{>}\, x_{j_2}, \ldots, E_m {-}{>}\, x_{j_m})$	$param(P_i) = x_{j_i}$

Se observa que cada parámetro real $R_i = E_i {-}{>}\, e_i$ puede descomponerse en dos partes, una expresión E_i, que constituye el parámetro que se va a pasar al subprograma, y una parte de etiquetado e_i, que permite definir el modelo de parametrización empleado. En el caso de la instanciación posicional, se puede asumir que $e_i = i$, y, por tanto, se elimina el etiquetado quedando sobreentendido.

A.4.2 Evaluación. Esquemas CBV y CBN

Al evaluar una expresión (por ejemplo, aritmética) como $f(g(a), h(b))$, se puede proceder de acuerdo a dos estrategias fundamentales [All90]:

- "Call by value" (CBV). Se evalua primero $x' = g(a)$ y luego $y' = h(b)$, es decir, se obtienen valores para $g(a)$ y $h(b)$ y los llamamos x' e y', respectivamente. A continuación, se calcula $f(x', y')$.

- "Call by name" (CBN). Se pasa a desarrollar $f(x, y)$ sin evaluar $g(a)$ y $h(b)$. Se sustituyen en la definición de $f(x, y)$ las ocurrencias de x por $g(a)$ y las de y por $h(b)$. A continuación, se simplifica la expresión resultante.

Ambas estrategias pueden emplearse para implementar la transformación de las expresiones iniciales contenidas en los parámetros reales a expresiones que sirven como instancia de parametrización al subprograma. En el segundo caso, CBN, la función *eval* se define de manera muy simple: $P_i = eval(E_i) = E_i$, es decir, no se realiza transformación alguna. En el primer caso, CBV, se puede dar una definición recursiva sencilla, como ahora vamos a ver.

La definición de una función *eval* que trabaje bajo el esquema CBV implica aclarar el tratamiento computacional de las ocurrencias de variables del lenguaje de programación en las expresiones definidas como parámetros. Se puede considerar una "entrada" de la memoria como un par $\langle nombre, valor \rangle$ que asocia a un identificador *nombre* un objeto representable en la máquina denotado como *valor*. Para ello, en la línea de Plotkin [All90, Plo81], consideramos esta memoria abstracta representada mediante entornos. Un *entorno* ε es, en general, una estructura de datos dinámica capaz de albergar pares $\langle nombre, valor \rangle$, provista de un conjunto de operaciones de manipulación que permiten respetar las propiedades de alcance y sobrecarga asignadas a los identificadores mediante las declaraciones. Las operaciones que necesitamos son:

- $obtener(x, \varepsilon)$, que devuelve el valor asociado al nombre x en el entorno ε.
- $actualizar(\langle x, a_x \rangle, \varepsilon)$, que hace que el identificador x tenga asociado, a partir de ese momento, el valor a_x en el entorno ε.

La función *eval*, según la estrategia CBV, se define como:

- $eval(a) = a$ si $a \in C_{LP}$.
- $eval(x) = obtener(x, \varepsilon)$.
- $eval(F(E_1, E_2, \ldots, E_k)) = f(eval(E_1), eval(E_2), \ldots, eval(E_k))$, siendo f la función matemática k-aria que calcula la función (subprograma) F.

A.4.3 Modelo de ejecución de un subprograma

A partir de ahora, nos centramos en la estrategia de evaluación CBV. La ejecución de una llamada a subprograma $F(R_1, R_2, \ldots, R_n)$ puede describirse, entonces, como sigue:

1. Evaluar los parámetros R_i, para obtener la instancia de parametrización P. Esto puede realizarse de manera secuencial o paralela. Normalmente, se asume que R_i

se evalúa con anterioridad a R_j si $i < j$. Pero puede no ser así en un lenguaje que desee aprovechar el paralelismo al máximo. Por tanto, si simbolizamos como $\Omega(R_i)$ al conjunto de operaciones que la máquina realiza para evaluar el parámetro R_i, podemos tener que: $\forall\omega \in \Omega(R_i),\ \forall\omega' \in \Omega(R_j),\ i < j \Rightarrow \omega \prec \omega'$, en el caso de evaluación de parámetros secuencial, o bien que no se cumpla ninguna condición especial respecto al orden de ejecución de las operaciones asociadas a la evaluación de los parámetros.

2. Ejecutar el procedimiento F, instanciando los parámetros reales según indique la función *param* aplicada sobre la instancia de parametrización P obtenida. Es evidente que, en cualquier caso, se debe cumplir que si $\Omega(F(P))$ denota el conjunto de operaciones generado por la ejecución de F, instanciados sus parámetros formales con la instancia P de la manera indicada por *param*, entonces: $\forall\omega \in \Omega(R_i),\ \forall\omega' \in \Omega(F(P)),\ \forall i : 1 \leq i \leq m.\ \ \omega \prec \omega'$.

   La ejecución de un subprograma primitivo φ, instanciado completamente con P, puede considerarse realizada mediante una operación única, de manera que el conjunto de operaciones asociado es unitario, $\Omega(\varphi(P)) = \{\omega_{\varphi(P)}\}$, aunque dependiente de la instancia de parametrización de la función. $\omega_{\varphi(P)}$ representa una instrucción elemental de la máquina abstracta donde ejecutamos nuestro programa.

A.5 Sentencias básicas

Aparte de las declaraciones, que pueden entenderse como directivas para la configuración de la memoria abstracta (entorno ε) que manipula el programa, las sentencias básicas que un lenguaje imperativo orientado a la programación estructurada pone a nuestra disposición son: la asignación, llamada a procedimiento, condicional y bucle.

* Asignación. La sentencia de asignación permite manipular la memoria de la máquina. Dado que la representamos como un entorno ε, podemos definir su efecto sobre la memoria en términos de las operaciones sobre los entornos.

NOTACION:

$$x := E;$$

representa la asignación a la variable x del valor obtenido al evaluar la expresión E. La evaluación de E se lleva a cabo mediante la función *eval*. El efecto de la asignación sobre la memoria es equivalente a: $actualizar(\langle x, eval(E)\rangle, \varepsilon)$.

* Llamada a procedimiento. Una llamada a procedimiento es un caso particular de llamada a subprograma. La instanciación del subprograma debe ser total para que

éste sea ejecutable.

* Condicional. La sentencia condicional permite la posibilidad de ejecutar selectivamente sentencias del programa en función de una expresión lógica E[6].

NOTACION:

if E **then** S
if E **then** S_1 **else** S_2

que se interpreta en la forma estándar.

* Bucle. Otro mecanismo de control de flujo en un programa imperativo es el bucle.

NOTACION:

while E **do** S

que representa la ejecución de la sentencia compuesta S mientras la evaluación de la expresión lógica E sea evaluada a $TRUE$.

A.6 Constructores de sentencias

Los constructores de sentencias tienen la misión de hacer explícito el orden de ejecución deseado para las sentencias del programa. Distinguimos los siguientes:

* Constructor de *secuencialidad*, que permite enlazar ordenadamente la ejecución de dos sentencias.

NOTACION:

$s; S$

De esta manera, se tiene que: $\forall \omega \in \Omega(s),\ \forall \omega' \in \Omega(S) \quad \omega \prec \omega'$.

* Constructor de *bloque*, que permite agrupar un conjunto de sentencias posibilitando que sean consideradas como una sola sentencia compuesta.

NOTACION:

block $S_1; S_2; \ldots; S_n$ **eblock**

El símbolo ";" puede interpretarse como el encargado de establecer explícitamente el orden de ejecución entre las instrucciones que agrupa. En este sentido, puede verse como un operador binario entre sentencias que establece que la ejecución debe realizarse de modo secuencial.

[6]Asumimos que el conjunto de constantes del lenguaje de programación, C_{LP}, incorpora las constantes $TRUE$ y $FALSE$.

A.7 Extensiones orientadas a la programación paralela

Se han desarrollado distintos mecanismos para expresar el paralelismo en los lenguajes de programación a diferentes niveles de abstracción. Veamos algunos de los más comunes.

A.7.1 Comunicaciones

A menudo los procesos que cooperan entre sí en un entorno multiprocesador deben comunicarse y sincronizarse. La ejecución de un proceso puede influir en otro a través de la comunicación. La comunicación entre procesos emplea uno de los dos esquemas siguientes [HB88]:

1. Uso de variables compartidas. Cuando se utiliza este esquema se emplean habitualmente dos tipos de sincronización: la *exclusión mutua* y la *sincronización de condición*. La primera asegura que un recurso físico o virtual se mantenga indivisible. La segunda surge al considerar un conjunto de procesos cooperantes cuando un objeto de datos compartido se encuentra en un estado inapropiado para la ejecución de una operación dada, en cuyo caso debe permanecer en suspenso.

2. Paso de mensajes. Proporcionan un método más flexible y directo para la comunicación entre procesos. Primitivas típicas son las funciones de *envío* (*send*) y *recepción* (*receive*), que permiten pasar una cadena de caracteres entre procesos.

En el lenguaje de programación que estamos definiendo, se han implementado las operaciones de comunicación síncrona análogas a *send* y *receive*. Se presentan como operadores de envío/recepción de mensajes.

NOTACION: (Operaciones de envío de mensajes)
−> canal;
E −> canal;

Representa, en el primer caso, la sincronización respecto del canal indicado del proceso que ejecuta la sentencia. Hasta que otro proceso no efectúa una operación de recepción sobre el mismo canal, no continúa la ejecución del proceso. En el segundo caso, el valor resultante de evaluar la expresión E se envía a través del canal.

NOTACION: (Operaciones de recepción de mensajes)
<− canal;
Var <− canal;

Representa, en el primer caso, la sincronización respecto del canal indicado del proceso que ejecuta la sentencia. Hasta que otro proceso no efectúa una operación de envío sobre el mismo canal no continúa la ejecución del proceso. En el segundo caso, el mensaje recibido a través del canal se deposita en la variable Var.

A.7.2 Constructores de paralelismo

Un programa paralelo para un multiprocesador consta de dos o más procesos que interactúan. Un proceso es un programa secuencial que se ejecuta concurrentemente con otros procesos. En la programación paralela, se pueden distinguir esencialmente dos tipos de paralelismo: explícito e implícito [CNR92, HB88]. Hemos optado por la primera alternativa: emplear un constructor de paralelismo, cuya misión es definir de forma explícita el orden de ejecución de las sentencias. Dicho constructor puede verse, análogamente a ";" en el caso de la construcción de sentencias secuenciales, como un operador binario entre sentencias.

NOTACION:

$$s \parallel S$$

Representa la ejecución de la sentencia s en paralelo con las sentencias recogidas en S. Se tiene que: $\forall \omega \in \Omega(s),\ \forall \omega' \in \Omega(S)\ \ \omega \not\prec \omega' \ \wedge\ \omega' \not\prec \omega$.

En general:

NOTACION:

$$\textbf{block}\ S_1 \parallel S_2 \parallel \ldots \parallel S_n\ \textbf{eblock}$$

Representa el agrupamiento de las sentencias S_i, $1 \leq i \leq n$, en una única sentencia (compuesta). Todas las sentencias se ejecutan en paralelo.

A.8 Estructura de un programa

Por último, veamos cómo combinar los diferentes componentes de un programa[7]:

NOTACION:

program P;
$DD_1; DD_2; \ldots; DD_n;$
$DS_1; DS_2; \ldots; DS_m;$
begin S **end**.

[7]Para separar las diferentes declaraciones se ha empleado el símbolo ";" que también denota la construcción secuencial de sentencias, pero en este caso se trata sólo de una ayuda sintáctica.

Esta construcción representa un programa, donde:

- DD_i, $0 \leq i \leq n$, representan declaraciones de datos.
- DS_j, $0 \leq j \leq m$, representan declaraciones de subprogramas.
- S es el punto de entrada del programa.

Apéndice B

El lenguaje de programación ALEPH

Después de estudiar cómo representar un programa escrito en un lenguaje de alto nivel imperativo como un proceso LCEP, por medio de la función de traducción o codificación τ, vamos a particularizarlo a un lenguaje de programación concreto, ALEPH (Aplicación del Lambda Cálculo Etiquetado Paralelo a un HLL). Posteriormente, se presenta una gramática completa para el lenguaje en la notación BNF. Con ella definimos el conjunto de sentencias Ψ que constituye un programa ALEPH válido. Terminamos con la definición de la función de traducción para ALEPH, $\tau_\Psi : \Psi \rightarrow \mathcal{M}$, que incorpora el tratamiento de las extensiones sintácticas añadidas al lenguaje.

B.1 Descripción de ALEPH

B.1.1 Elementos del lenguaje de programación

Perseguimos la simulación de un programa de alto nivel sobre la máquina LCEP y esto significa la adecuación de las semánticas definidas por el programa y por el proceso LCEP que lo representa. Para ello es conveniente identificar el papel que va a desempeñar cada uno de los objetos declarados en el programa. Sin entrar en un tratamiento de los elementos del lenguaje que permita hablar de *tipos*, asociamos determinados atributos a las entidades declaradas para establecer claramente el papel que se les asigna en los programas.

1. Elementos utilizados en el lenguaje de programación. Son:
   - Constantes
   - Variables

- Subprogramas: funciones y procedimientos
- Canales de comunicación

2. Características asociadas a los objetos del lenguaje. Los objetos del lenguaje pueden adquirir matices diversos en cuanto a su utilización, asignándoles ciertos atributos. Los atributos de los objetos son:

   - Dato
   - pCode
   - fCode

   Tanto las funciones como los procedimientos son compatibles con las variables y canales de atributos fCode y pCode, respectivamente. En adelante, a menudo nos referiremos conjuntamente a estos atributos como xCode.

3. Alcance de los elementos declarados en un programa. También podemos hablar del alcance de los elementos manejados en el programa. En este sentido, distinguimos elementos:

   - *Globales*: disponibles desde cualquier punto del programa.
   - *Locales*: disponibles sólo en el cuerpo de definición de un procedimiento o función.

   No pueden darse todas las combinaciones imaginables. En la tabla presentamos las permitidas.

Expresión	**Nombre ALEPH**	**Atributos**	**Alcance**
`Constante`	**Const**	`Dato`	`Global`
`Variable`	**Var**	`Dato, pCode, fCode`	`Global, Local`
`Procedimiento`	**Procedure**	`pCode`	`Global`
`Función`	**Function**	`fCode`	
`Canal`	**Channel**	`Dato, pCode, fCode`	`Global, Local`
`Parámetros formales`		`Dato, pCode, fCode`	`Local`

4. Los atributos xCode

   Los atributos xCode son una herramienta para permitir el paso de llamadas completa y parcialmente instanciadas de funciones y procedimientos como parámetros de otras funciones y procedimientos. Una llamada completamente instanciada de una función puede considerarse también un *dato*, pues se evalúa a una constante del lenguaje de programación. Con este tipo de elemento se permite facilitar el soporte de la inhibición de la evaluación de parámetros y, en general, la gestión de subprogramas anónimos.

A continuación se presentan las normas que definen el comportamiento de los elementos que poseen estos atributos:

- Los parámetros formales de un subprograma que se declaran con el atributo pCode o fCode pueden asociarse con parámetros reales de las características mencionadas y emplearse dentro del cuerpo del subprograma como procedimientos y funciones, respectivamente.

- También se pueden enviar variables xCode a través de canales de comunicación con el mismo atributo que el elemento que se desea transferir.

- La única restricción impuesta sobre ellas es la necesidad de que sean asignadas únicamente a procedimientos o funciones parcialmente instanciados, que no se evalúan al efectuar la asignación. Tampoco se evalúan al pasarlos como argumentos a un subprograma, aunque sí sus parámetros reales.

- Las variables pCode se comportan en todo momento como procedimientos y las fCode como funciones. Pueden parametrizarse exactamente como cualquier programa, excepto mediante etiquetado simbólico. Sin embargo, no se controla la correcta parametrización de las llamadas, que deben estar siempre completamente instanciadas. Esto puede originar problemas en el caso de objetos fCode.

- Al instanciar un parámetro formal de tipo xCode se prefija el elemento instanciado con el operador @. También ocurre lo mismo al efectuar una asignación y al enviar un objeto compatible con fCode o pCode a través de un canal.

Ejemplo 20 *Como ejemplo inicial de la sintaxis de ALEPH y del uso correcto de los objetos xCode, veamos el siguiente programa:*

```
      program Ejemplo;
            /* DECLARACIONES DE OBJETOS GLOBALES */
      var x;      /* Los objetos tienen, por defecto, atributo dato.  */
      channel cf:fCode; cp:pCode;
/* Los atributos acompañan las declaraciones de objetos.  */
/* En las declaraciones de los subprogramas, se distingue entre */
/* funciones y procedimientos.  */
      function G(x,y) =
         block
            G:=x-y;
         eblock;
      function F(x,y,g:fCode) =
         block
            F:=g(x)*y;;
/* Se emplea una ocurrencia fCode como una función.  */
         eblock;
      procedure Presentar =
         var f:fCode; p:pCode;
         block
/* El procedimiento puede parametrizarse a través de cf y cp.  */
            f <::  cf; print(f(1));
/* Se recibe una función a través de un canal fCode.  */
            p <::  cp; p;
/* Se recibe un procedimiento a través de un canal pCode.  */
         eblock
               /* PROGRAMA PRINCIPAL */
         begin
            print(F(1,2,@G(3->y)));
/* Instanciación parcial de un parámetro formal.  */
/* G(3->y) es la función x-3.  El resultado de esta sentencia */
/* es la presentación por pantalla de -4.  */
            block
               @G(1) ::> cf || Presentar || @Presentar ::> cp
/* G(1) representa la función 1-y.  */
/* El resultado de estas sentencias es la presentación por */
/* pantalla de 0.  Posteriormente, el sistema se bloquea, */
/* esperando recibir alguna información sobre el canal cf.  */
            eblock;
         end.
```

Como se puede observar, los elementos pCode son compatibles con los procedimientos (parcialmente instanciados) y los objetos fCode son compatibles con las funciones (parcialmente instanciadas). Ya hemos visto que permiten dar soporte al concepto de subprograma anónimo y, de este modo, ALEPH ofrece una forma práctica de utilización de la *currificación*.

B.1.2 Sintaxis esquemática

1) Declaración de constantes, variables y canales:

En la declaración, se especifica el tipo del elemento que se define con una palabra reservada (**const**, **var**, **channel**). A continuación, se da una lista de identificadores que los nombran y se asocia (opcionalmente y cuando se permita) el atributo (ninguno si es dato, pCode o fCode) que describe la función para la que se va a emplear el elemento: contener un dato, un procedimiento anónimo o una función anónima en el caso de variables, transferir un dato, un procedimiento anónimo o una función anónima en el caso de canales.

Ejemplo 21 *Veamos un ejemplo:*

```
var x; /* Tipo dato, por defecto.  */
channel cf:  fCode; cp:  pCode;
```

2) Declaración de procedimientos y funciones:

Se declara un subprograma empleando **function** o **procedure** para indicar qué clase de subprograma se describe. La lista de parámetros formales constituye una declaración de elementos *variables* (no hay canales entre los parámetros formales) y cada uno de ellos lleva asociado un atributo, siguiendo las mismas normas que las declaraciones globales vistas anteriormente. Se admite la pre-declaración de un subprograma indicando únicamente el nombre y parámetros del mismo, con el fin de emplearlo antes de definir su cuerpo.

Las declaraciones de elementos locales siguen también las mismas normas que los elementos globales. Las sentencias del cuerpo del procedimiento o función se encierran entre **block** y **eblock**. Sin embargo, las sentencias simples encerradas pueden ser tanto secuenciales como paralelas. Las declaraciones locales de canales sirven únicamente para establecer la relación de orden en $\mathcal{L}$. Los canales locales se engloban en el conjunto S_L. No hay generación de código LCEP asociada a las declaraciones de canales locales, contrariamente a lo que sucede con las variables, que introducen un m-proceso en el l-proceso de la función o procedimiento. Los valores de retorno para las funciones son siempre de tipo dato.

Ejemplo 22 *Veamos un ejemplo de procedimiento:*

```
      procedure Presentar;
/* El procedimiento puede ser pre-declarado. */
      procedure Presentar =
        var f:fCode, p:pCode;
/* Las declaraciones locales son como las globales. */
        block /* Las sentencias se encierran entre block y eblock. */
          f <:: cf; print(f(1));
          p <:: cp; p;
        eblock;
```

Ejemplo 23 *Veamos un ejemplo de función:*

```
      function F(x, y; g:fCode) =
/* Los parámetros son, por defecto, de tipo dato. */
        block
          F:=g(x)*y;
/* El valor de retorno de la función se asigna al nombre de ésta. */
        eblock;
```

3) Asignación:

La asignación de valor a una variable debe ser coherente con el atributo asociado a dicha variable. Se puede asignar cualquier expresión a una variable de tipo dato (en una expresión podemos incluir llamadas a funciones anónimas representadas por variables de tipo fCode). A una variable pCode podemos asociarle una llamada a procedimiento parcialmente instanciada, precedida por el operador de inhibición de evaluación @. Lo mismo ocurre con las variables fCode y las funciones.

Ejemplo 24 *Suponiendo las declaraciones del programa del Ejemplo 20, se pueden realizar las siguientes asignaciones:*

```
      x:=sen(3);
/* Se asigna a la variable dato un valor numérico. */
      f:=@G(1);
/* Se asigna a la variable fCode una función parcialmente instanciada. */
      p:=@Presentar;
/* Se asigna a la variable pCode un procedimiento totalmente */
/* instanciado. */
```

4) Llamada a procedimiento y función:

Se introducen algunas ampliaciones en el tratamiento de las llamadas a función y procedimiento. Se refieren a los siguientes aspectos:

* Modelos de parametrización: Se han introducido algunas posibilidades más. Su efecto sobre la definición de τ_Ψ es mínimo.

Tipo de instanciación	**Características**
`Posicional`	`Los parámetros reales se especifican suponiendo la asignación posicional a los parámetros formales.`
`Simbólica`	`Indicamos explícitamente cómo asociar los parámetros reales a los distintos parámetros formales. El orden en que aparecen las asociaciones en la invocación no se considera a la hora de ligar los parámetros formales a los reales. El programador debe conocer los nombres de los parámetros formales de la función.`
`Numérica`	`Indicamos explícitamente cómo asociar los parámetros reales a los formales, pero mediante la posición que el parámetro formal ocupa en la lista de parámetros formales de la declaración de la función.`
`Posicional-Simbólica`	`Indicamos primero una lista de parámetros reales que se asocian según su posición, y luego, explícitamente, mediante etiquetas simbólicas, indicamos la asociación que deseamos realizar entre los restantes parámetros reales y formales.`
`Posicional-Numérica`	`Indicamos primero una lista de parámetros reales que se asocian según su posición, y luego explícitamente, mediante etiquetas numéricas, indicamos la asociación que deseamos realizar entre los restantes parámetros reales y formales.`

Tipo de instanciación	**Ejemplo**
`Posicional`	$F(a, b, \ldots, c)$ x_1 `toma el valor` a x_2 `toma el valor` b ... x_n `toma el valor` c
`Simbólica`	$F(a{-}{>}\, x_2, b{-}{>}\, x_n, \ldots, c{-}{>}\, x_1)$ x_1 `toma el valor` c x_2 `toma el valor` a ... x_n `toma el valor` b
`Numérica`	$F(a{-}{>}\, 2, b{-}{>}\, n, \ldots, c{-}{>}\, 1)$ x_1 `toma el valor` c x_2 `toma el valor` a ... x_n `toma el valor` b
`Posicional-Simbólica`	$F(a, b{-}{>}\, x_n, \ldots, c{-}{>}\, x_2)$ x_1 `toma el valor` a x_2 `toma el valor` c ... x_n `toma el valor` b
`Posicional-Numérica`	$F(a, b{-}{>}\, n, \ldots, c{-}{>}\, 2)$ x_1 `toma el valor` a x_2 `toma el valor` c ... x_n `toma el valor` b

* Inhibición de la evaluación: En el caso de la instanciación de parámetros que se definen como pCode o fCode en la declaración de la función o procedimiento, se debe inhibir la evaluación final[1] de la expresión especificada como parámetro. Esto se lleva a cabo mediante el operador @.

Ejemplo 25 *Ejemplo de inhibición:*

```
print(F(1,2,@G(3->y)));
```

[1]Recordemos que la inhibición de evaluación afecta únicamente a la función más externa de la expresión, es decir, @F(G(x)) implica la evaluación total de G(x) pero la no evaluación de F sobre el resultado obtenido.

* Evaluación paralela de los parámetros pasados a un subprograma:
Se permite al programador indicar explícitamente si desea evaluar los parámetros reales pasados a un procedimiento o función de manera secuencial (como es habitual) o en paralelo. Esto se indica *encerrando el bloque de parámetros reales entre llaves* en lugar de entre paréntesis. En el caso de la evaluación secuencial, los parámetros reales se evalúan en el orden de aparición en la lista de parámetros, independientemente de la asociación que se establezca con los parámetros formales.

Ejemplo 26 *Ejemplo de evaluación paralela:*

```
print(F{1,2,@G(3->y)});
```

Los parámetros de print y G se evalúan en serie, los de F en paralelo.

* Ejecución de objetos xCode: Un objeto pCode puede hacer las veces de un procedimiento. Un objeto fCode puede ocupar el lugar de una función en una expresión. El problema con estos últimos estriba en la posibilidad de que se de una instanciación incompleta de una función anónima, en cuyo caso los resultados de la ejecución son incorrectos. La gestión del problema se deja en manos del programador.

Ejemplo 27 *Ejemplo de ejecución de objetos xCode:*

```
          f:=@G(1);
          p:=@Presentar;
          p; /* Ejecutamos un subprograma anónimo.  */
          print(f(2));
/* Se visualiza el valor devuelto por la llamada a la */
/* función anónima f(2).  */
```

5) Comunicaciones síncronas:

Se ha sustituido el operador -> por ::> y el operador <- por <:: De esta manera, se resalta la diferencia conceptual en un lenguaje de alto nivel entre la especificación etiquetada de parámetros (para la cual se mantiene ->) y las comunicaciones síncronas.

Ejemplo 28 *Ejemplo de comunicaciones síncronas:*

```
          ::> canal; /* Sincronización.  */
          sen(3) ::> cDato;
/* Se envía una función anónima sobre un canal fCode.  */
          @G(1) ::> cf;
/* Se envía una función anónima sobre un canal fCode.  */
          @Presentar ::> cp;
```

```
/* Se envía un procedimiento anónimo sobre un canal pCode.  */
            <::  canal; /* Sincronización.  */
            x <::  cDato; /* Se recibe un dato en la variable x.  */
            f <::  cf;
/* Se recibe una función anónima sobre un canal fCode en una */
/* variable fCode.  */
            p <::  cp; p;
/* Se recibe un procedimiento anónimo sobre un canal pCode */
/* en una variable fCode y, posteriormente, se ejecuta.  */
```

6) Elección indeterminista:

Se introduce una sentencia de elección indeterminista de sentencias sujetas a posible comunicación sobre canales, cuya sintaxis general es:

random
  ? $\texttt{COM}_1$; $\texttt{S}_1$
  ? $\texttt{COM}_2$; $\texttt{S}_2$
  ...
  ? $\texttt{COM}_n$; $\texttt{S}_n$
erandom

donde $\texttt{COM}_i$ puede ser cualquier sentencia de comunicación (tanto de recepción como de envío) sobre cualquier canal. $\texttt{S}_i$ puede ser cualquier sentencia.

B.2 Gramática

A continuación se presenta la gramática completa del lenguaje de programación usando la notación BNF.

a) Elementos sintácticos del lenguaje de programación

a.1 Estructura del programa

⟨Programa⟩ ::=
  program Id ; ⟨Declaraciones⟩
    begin ⟨SentenciasConstruccion⟩ **end.**

⟨Declaraciones⟩ ::=
  ⟨DeclaracionSimbolos⟩⟨DefinicionesSubprg⟩

⟨DeclaracionSimbolos⟩ ::=
⟨DeclaracionClaseSimbolo⟩ ;
⟨DeclaracionSimbolos⟩ | ε
⟨DeclaracionClaseSimbolo⟩ ::=
const ⟨ListaAsocCte⟩ | **var** ⟨ListaVar⟩ |
channel ⟨ListaCanal⟩ | ⟨DeclaracionSubprg⟩

⟨ListaAsocCte⟩ ::=
⟨ListaId⟩ = ⟨Const⟩⟨RestoListaAsocCte⟩ |
⟨ListaId⟩ = ⟨IdConst⟩⟨RestoListaAsocCte⟩

⟨RestoListaAsocCte⟩ ::= ; ⟨ListaAsocCte⟩ | ε

⟨ListaVar⟩ ::= ⟨ListaId⟩⟨Atributo⟩⟨RestoListaVar⟩
⟨RestoListaVar⟩ ::= ; ⟨ListaVar⟩ | ε
⟨Atributo⟩ ::= **: fCode** | **pCode** | ε
⟨ListaCanal⟩ ::= ⟨ListaVar⟩

⟨DeclaracionSubprg⟩ ::=
function Id ⟨ParmSubprg⟩⟨RestoDeclaracionSubprg⟩ |
procedure Id ⟨ParmSubprg⟩⟨RestoDeclaracionSubprg⟩
⟨RestoDeclaracionSubprg⟩ ::= ; ⟨DeclaracionSubprg⟩ | ε

⟨DefinicionSubprg⟩ ::=
function Id ⟨ParmSubprg⟩ =
⟨DeclaracionLocal⟩⟨CuerpoSubprg⟩ |
procedure Id ⟨ParmSubprg⟩ =
⟨DeclaracionLocal⟩⟨CuerpoSubprg⟩

⟨ParmSubprg⟩ ::= **(**⟨ListaVar⟩**)** | ε
⟨CuerpoSubprg⟩ ::=
block ⟨SentenciasConstruccion⟩ **eblock**

En el cuerpo de una función, se debe asignar un valor resultado al identificador de la función.

⟨DeclaracionLocal⟩ ::=
var ⟨ListaVar⟩ ; ⟨DeclaracionLocal⟩ |
channel ⟨ListaCanal⟩ ; ⟨DeclaracionLocal⟩ | ε

⟨ListaId⟩ ::= **Id** ⟨RestoListaId⟩

⟨RestoListaId⟩ ::= **,** ⟨ListaId⟩ | ε

a.2 Construcción de sentencias

⟨Sentencias⟩ ::=
⟨SentenciaSimple⟩ | ⟨SentenciaCompuesta⟩ |
⟨SentenciasEleccion⟩

⟨SentenciaCompuesta⟩ ::=
block ⟨SentenciasConstruccion⟩ **eblock**

⟨SentenciasConstruccion⟩ ::=
⟨SentenciasSecuenciales⟩ | ⟨SentenciasParalelas⟩ |
⟨SentenciasEleccion⟩

⟨SentenciasSecuenciales⟩ ::=
⟨Sentencias⟩⟨RestoSentenciasSecuenciales⟩ | ε

⟨RestoSentenciasSecuenciales⟩ ::=
; ⟨SentenciasSecuenciales⟩ | ε

Una sentencia de construcción vacía es, por definición, una sentencia secuencial.

⟨SentenciasParalelas⟩ ::=
⟨Sentencias⟩ $\|$ ⟨Sentencias⟩⟨RestoSentenciasPar⟩

⟨RestoSentenciasPar⟩ ::=
$\|$ ⟨Sentencias⟩⟨RestoSentenciasPar⟩ | ε

⟨SentenciasEleccion⟩ ::=
random ⟨ListaElecciones⟩ **erandom**

⟨ListaElecciones⟩ ::=
? ⟨Comunicacion⟩ ;
⟨Sentencias⟩⟨RestoListaElecciones⟩

⟨RestoListaElecciones⟩ ::= **;** ⟨ListaElecciones⟩ | ε

a.3 Sentencias básicas

⟨`SentenciaSimple`⟩ `::=`
⟨`Asignacion`⟩ | ⟨`LlamadaProc`⟩ | ⟨`Comunicacion`⟩

a.3.1 Asignación

⟨`Asignacion`⟩ `::=`
⟨`VarDato`⟩ := ⟨`Expresion`⟩ |
⟨`VarpCode`⟩ := @ ⟨`LlamadaGralProc`⟩ |
⟨`VarfCode`⟩ := @ ⟨`LlamadaGralFunc`⟩ |
⟨`RetFunc`⟩ := ⟨`Expresion`⟩

Esta última forma de asignación se permite sólo en el cuerpo de una función, cuyo identificador aparece a la izquierda de la asignación.

⟨`LlamadaGralProc`⟩ `::=`
⟨`Proc`⟩⟨`RestoLlamadaSubprg`⟩ |
⟨`VarpCode`⟩⟨`RestoLlamadaxCode`⟩

Este no terminal acepta que la lista de parámetros especificada esté incompleta.

⟨`LlamadaGralFunc`⟩ `::=`
⟨`Func`⟩⟨`RestoLlamadaSubprg`⟩ |
⟨`VarfCode`⟩⟨`RestoLlamadaxCode`⟩

Este no terminal acepta que la lista de parámetros especificada esté incompleta.

a.3.2 Llamada a procedimiento

⟨`LlamadaProc`⟩ `::=`
⟨`Proc`⟩⟨`RestoLlamadaSubprg`⟩ |
⟨`VarpCode`⟩⟨`RestoLlamadaxCode`⟩

Este no terminal no acepta más que la especificación completa de los parámetros que hayan sido definidos para ⟨`Proc`⟩. En cambio, no analiza los pasados a ⟨`VarpCode`⟩.

⟨`RestoLlamadaSubprg`⟩ `::=`
(⟨`ListaParamReal`⟩**)**|**{**⟨`ListaParamReal`⟩**}**| ε

⟨RestoLlamadaxCode⟩ ::=
(⟨ListaParamRealxCode⟩) | {⟨ListaParamRealxCode⟩} | ε

a.3.3 Comunicación

⟨Comunicacion⟩ ::= ⟨Input⟩ | ⟨Output⟩

⟨Input⟩ ::=
<:: ⟨CanalpCode⟩ | ⟨VarDato⟩ <:: ⟨CanalDato⟩ |
⟨CanalpCode⟩ <:: ⟨CanalpCode⟩ |
⟨VarfCode⟩ <:: ⟨CanalfCode⟩

⟨Output⟩ ::=
::> ⟨CanalpCode⟩ | ⟨Expresion⟩ ::> ⟨CanalDato⟩ |
@ ⟨LlamadaGralProc⟩ ::> ⟨CanalpCode⟩ |
@ ⟨LlamadaGralFunc⟩ ::> ⟨CanalfCode⟩

a.3.4 Condicional

⟨Condicional⟩ ::=
if ⟨Expresion⟩ **then** ⟨Sentencias⟩ |
if ⟨Expresion⟩ **then** ⟨Sentencias⟩ **else** ⟨Sentencias⟩

a.3.5 Bucle

⟨Bucle⟩ ::= **while** ⟨Expresion⟩ **do** ⟨Sentencias⟩

a.4 Parametrización de las llamadas a función y procedimiento

⟨ListaParamReal⟩ ::=
⟨ListaParamSimple⟩ | ⟨ListaParamEtiq⟩
⟨ListaParamSimple⟩ ::=
⟨ParametroReal⟩⟨RestoParamSimple⟩
⟨RestoParamSimple⟩ ::= **,** ⟨ListaParamReal⟩ | ε

Los parámetros pueden empezar a indicarse posicionalmente. Podemos pasar al modo simbólico o numérico.

⟨ListaParamEtiq⟩ ::=
  ⟨ParametroReal⟩ -> ⟨Etiq⟩⟨RestoParamEtiq⟩

Este no terminal controla que el atributo de la etiqueta sea congruente con el definido para el parámetro real.

⟨RestoParamEtiq⟩ ::= , ⟨ListaParamEtiq⟩ | ε

⟨ListaParamRealxCode⟩ ::=
  ⟨ListaParamSimplexCode⟩ | ⟨ListaParamEtiqxCode⟩
⟨ListaParamSimplexCode⟩ ::=
  ⟨ParametroReal⟩⟨RestoParamSimplexCode⟩
⟨RestoParamSimplexCode⟩ ::=
  ,⟨ListaParamRealxCode⟩ | ε
⟨ListaParamEtiqxCode⟩ ::=
  ⟨ParametroReal⟩ -> ⟨EtiqNum⟩⟨RestoParamEtiqxCode⟩

Este no terminal controla que el atributo de la etiqueta sea congruente con el definido para el parámetro real.

⟨RestoParamEtiqxCode⟩ ::= , ⟨ListaParamEtiqxCode⟩ | ε

⟨RestoParamEtiqxCode⟩ ::=
  ⟨Expresion⟩ | **@** ⟨LlamadaGralProc⟩ |
  @ ⟨LlamadaGralFunc⟩

a.5 Construcción de expresiones

⟨Expresion⟩ ::= ⟨Relacion⟩⟨RestoExpresion⟩
⟨RestoExpresion⟩ ::=
  ⟨RestoRelacionAND⟩ | ⟨RestoRelacionOR⟩ |
  ⟨RestoRelacionXOR⟩
⟨RestoRelacionAND⟩ ::=
  and ⟨Relacion⟩⟨RestoRelacionAND⟩ | ε
⟨RestoRelacionOR⟩ ::=
  or ⟨Relacion⟩⟨RestoRelacionOR⟩ | ε
⟨RestoRelacionXOR⟩ ::=
  xor ⟨Relacion⟩⟨RestoRelacionXOR⟩ | ε

⟨Relacion⟩ ::=
⟨ExpresionSimple⟩ **Oprel**
⟨ExpresionSimple⟩ | ⟨ExpresionSimple⟩

⟨ExpresionSimple⟩ ::=
Signo ⟨Termino1⟩ **Opsuma** ⟨Termino2⟩ |
Signo ⟨Termino⟩ |
⟨Termino1⟩ **Opsuma** ⟨Termino2⟩ | ⟨Termino⟩

⟨Termino⟩ ::= ⟨Factor⟩ **Opmult** ⟨Factor⟩ | ⟨Factor⟩
⟨Factor⟩ ::=
⟨FactorPrimario1⟩ $\wedge$ ⟨FactorPrimario2⟩ |
⟨FactorPrimario⟩ | **not** ⟨FactorPrimario⟩

⟨FactorPrimario⟩ ::=
⟨Const⟩ | ⟨IdConst⟩ | ⟨VarDato⟩ |
⟨LlamadaFunc⟩ | **(**Expresion**)**

⟨LlamadaFunc⟩ ::=
⟨Func⟩⟨RestoLlamadaSubprg⟩ |
⟨VarfCode⟩⟨RestoLlamadaxCode⟩

a.6 Elementos sintácticos básicos

⟨Const⟩ ::= **Num** | **Cadena**
⟨IdConst⟩ ::= **Id**
⟨VarDato⟩ ::= **Id**
⟨VarpCode⟩ ::= **Id**
⟨VarfCode⟩ ::= **Id**
⟨Etiq⟩ ::= ⟨EtiqSimb⟩ | ⟨EtiqNum⟩
⟨EtiqSimb⟩ ::= **Id**
⟨EtiqNum⟩ ::= **EnteroPositivo**
⟨Proc⟩ ::= **Id**
⟨Func⟩ ::= **Id**
⟨RetFunc⟩ ::= **Id**
⟨CanalDato⟩ ::= **Canal**
⟨CanalpCode⟩ ::= **Canal**
⟨CanalfCode⟩ ::= **Canal**

Cada vez que se analiza uno de los no terminales básicos, además de calificar el elemento leído como identificador, número, ... se comprueba que existe una declaración adecuada para el identificador en cuestión; es decir, que un identificador encontrado como una variable fCode ($\langle$`VarfCode`$\rangle$) es realmente una variable con el atributo fCode.

b) Elementos léxicos del lenguaje de programación

Se dan las expresiones regulares para los elementos léxicos del lenguaje.

Id = Letra(Letra+Digito)*

Cadena = "(ASCII)***"**

EnteroPositivo = DigitoNoNulo(Digito)*

Digitos = Digito(Digito)*

FraccionOpc = . Digitos+ε

Canal = Id+EnteroPositivo

Num = Digitos FraccionOpc

Letra = Mayuscula+Minuscula

Digito $= \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$

DigitoNoNulo $= \{1, 2, 3, 4, 5, 6, 7, 8, 9\}$

Mayuscula $= \{A, B, \ldots, Z\}$

Minuscula $= \{a, b, \ldots, z\}$

Signo $= \{+, -\}$

Opsuma $= \{+, -\}$

Opmult $= \{*, /\}$

Oprel $= \{=, <>, <, <=, >=, >\}$

B.3 Función de traducción

Presentamos, por último, la función de traducción:

$$\tau_\Psi : \Psi \to \mathcal{M}$$

que transforma un programa ALEPH en un proceso LCEP ejecutable.

a) Declaración de procedimientos y funciones:

$\tau_{\Psi_{1.1}}$

$$\tau_\Psi \left[\begin{array}{l} \textbf{procedure } F(x_1, x_2, \ldots, x_n) = \\ \textbf{var } x, y, \ldots, z; \\ \textbf{channel } c_1, c_2, \ldots, c_k; \\ \textbf{block} \\ S(x_1, x_2, \ldots, x_n, x, y, \ldots, z, u, v, \ldots, w) \\ \textbf{eblock}; \, S' \end{array} \right]$$

$$= !\widehat{_F}\ \Lambda(F, x_1, \ldots, x_n, M_S(x_1, \ldots, x_n, x, \ldots, z, u, \ldots, w)) \parallel \tau_\Psi[S']$$

$\tau_{\Psi_{1.2}}$

$$\tau_\Psi \left[\begin{array}{l} \textbf{function } F(x_1, x_2, \ldots, x_n) = \\ \textbf{var } x, y, \ldots, z; \\ \textbf{channel } c_1, c_2, \ldots, c_k; \\ \textbf{block} \\ S(x_1, x_2, \ldots, x_n, x, y, \ldots, z, u, v, \ldots, w, f) \\ \textbf{eblock}; \, S' \end{array} \right]$$

$$= !\widehat{_F}\ \Lambda(F, x_1, \ldots, x_n, M_S(x_1, \ldots, x_n, x, \ldots, z, u, \ldots, w, f)) \parallel \tau_\Psi[S']$$

donde $\Lambda(F, x_1, \ldots, x_n, M_S(x_1, \ldots, x_n, x, \ldots, z, u, \ldots, w))$ es una forma abreviada de representar la abstracción multicanal que se emplea para definir el contenido del m-proceso asociado al subprograma F. Esta se define recursivamente como:

$$\begin{array}{l} \Lambda(F, x_i, x_{i+1}, \ldots, x_n, \\ M_S(x_1, x_2, \ldots, x_n, x, y, \ldots, z, u, v, \ldots, w, f(*))) = \end{array}$$

$$\begin{cases} \left(\begin{array}{l} \lambda_1 x_i'.\Lambda(F, x_{i+1}, \ldots, x_n, \\ M_S(x_1, x_2, \ldots, x_n, x, y, \ldots, z, u, v, \ldots, w, f(*)))+ \\ +\lambda_{F_{x_i}} x_i'.\Lambda(F, x_{i+1}, \ldots, x_n, \\ M_S(x_1, x_2, \ldots, x_n, x, y, \ldots, z, u, v, \ldots, w, f(*))) \end{array} \right) & \text{si } i \leq n \\ M_S(x_1, x_2, \ldots, x_n, x, y, \ldots, z, u, v, \ldots, w, f(*))) & \text{otro caso} \end{cases}$$

donde:

$$M_S(x_1, x_2, \ldots, x_n, x, y, \ldots, z, u, v, \ldots, w, f(*))) =$$

$$\lambda_\sigma \sigma'.\lambda_{kill} kill'.(kill' \circ \widehat{1}\ \sigma') \parallel$$
$$\widehat{\sigma} \begin{pmatrix} \lambda_{asig} asig'.asig' \circ \widehat{1}\ x_1 \circ \widehat{1}\ x_1' \parallel \cdots \parallel \\ \lambda_{asig} asig'.asig' \circ \widehat{1}\ x_n \circ \widehat{1}\ x_n' \parallel \\ \lambda_{asig} asig'.asig' \circ \widehat{1}\ x \circ \widehat{1}\ 0 \parallel \cdots \parallel \\ \lambda_{asig} asig'.asig' \circ \widehat{1}\ z \circ \widehat{1}\ 0 \parallel \\ \lambda_{asig} asig'.asig' \circ \widehat{1}\ f \circ \widehat{1}\ 0 \quad (*) \\ \tau_\Psi[S(x_1, x_2, \ldots, x_n, x, y, \ldots, u, v, \ldots);\ \mathbf{return}\ f(*)] \end{pmatrix}$$

$(*)$ Sólo en el caso de funciones.

b) Asignación:

$\tau_{\Psi_{2.1}}$

$$\tau_\Psi[x := E; S] =$$

$$\lambda_\sigma \sigma'.\lambda_{asig} asig'.(asig' \circ \widehat{1}\ x \circ \widehat{1}\ \sigma' \parallel \tau_\Psi[S]) \parallel \tau_\Psi[E \to \sigma]$$

$\tau_{\Psi_{2.2}}$

$$\tau_\Psi[u := E; S] = \lambda_\sigma \sigma'.(\widehat{_{global}}(u \circ \sigma') \circ \tau_\Psi[S]) \parallel \tau_\Psi[E \to \sigma]$$

$\tau_{\Psi_{2.3}}$

$$\tau_\Psi[x := @F(R_1, R_2, \ldots, R_m); S] =$$

$$\tau_{\Psi_\lambda}[R_1, R_2, \ldots, R_m] \lambda_F F'.\lambda_{asig} asig'.(asig' \circ \widehat{1}\ x \circ$$
$$\circ \widehat{1}(F' \widehat{_{\tau_\Psi}}[F, R_1, R_2, \ldots, R_m]) \parallel \tau_\Psi[S]) \parallel \tau_\Psi[\tau_{\Psi_\sigma}[R_1, R_2, \ldots, R_m]]$$

$\tau_{\Psi_{2.4}}$

$$\tau_\Psi[u := @F(R_1, R_2, \ldots, R_m); S] =$$

$$\tau_{\Psi_\lambda}[R_1, R_2, \ldots, R_m] \lambda_F F'.(\widehat{_{global}}(u \circ F' \widehat{_{\tau_\Psi}}[F, R_1, R_2, \ldots, R_m]) \circ \tau_\Psi[S]) \parallel$$
$$\parallel \tau_\Psi[\tau_{\Psi_\sigma}[R_1, R_2, \ldots, R_m]]$$

Las definiciones $\tau_{\Psi_{2.3}}$ y $\tau_{\Psi_{2.4}}$ implementan la definición de subprogramas anónimos. Lo que se persigue es la instanciación de m parámetros del subprograma F y, sin evaluar la aplicación de F sobre los parámetros, dejar el proceso obtenido como contenido de un m-proceso con dirección x (o u). Creamos un entorno de ejecución para evaluar cada parámetro de F y, a continuación, aplicamos la función F sobre ellos, pero sin evaluarla. Una vez realizado esto, la dejamos ligada a la variable correspondiente (que es el subprograma anónimo) como contenido del m-proceso de dirección x (o u).

c) Llamada a procedimiento:

τ_{Ψ_3}

$$\tau_\Psi[F(R_1, R_2, \ldots, R_m); S] =$$

$$\lambda_\sigma \sigma'.(0 \circ \tau_\Psi[S]) \parallel \tau_\Psi[F(R_1, R_2, \ldots, R_m) \to \sigma]$$

d) Condicional:

$\tau_{\Psi_{4.1}}$

$$\tau_\Psi[\text{if } E \text{ then } S_1; S] =$$

$$\lambda_\sigma \sigma'.\lambda_{if} if'.(if' \circ \hat{1}\ \sigma' \circ \hat{1}\ \tau_\Psi[S_1; S] \circ \hat{1}\ \tau_\Psi[S]) \parallel \tau_\Psi[E \to \sigma]$$

$\tau_{\Psi_{4.2}}$

$$\tau_\Psi[\text{if } E \text{ then } S_1 \text{ else } S_2; S] =$$

$$\lambda_\sigma \sigma'.\lambda_{if} if'.(if' \circ \hat{1}\ \sigma' \circ \hat{1}\ \tau_\Psi[S_1; S] \circ \hat{1}\ \tau_\Psi[S_2; S]) \parallel \tau_\Psi[E \to \sigma]$$

e) Bucle:

τ_{Ψ_5}

$$\tau_\Psi[\text{while } E \text{ do } S_1; S] =$$

$$\lambda_{\sigma_0} \sigma_0'.\lambda_{while} while'.(while' \circ \hat{1}\ \sigma_0' \circ \hat{1}\ \sigma_0 \circ \hat{1}\ \tau_\Psi[E \to \sigma_0] \circ \hat{1}\ \sigma_1 \circ$$
$$\circ \hat{1}\ \tau_\Psi[S_1; 0 \to \sigma_1] \circ \hat{1}\ \tau_\Psi[S]) \parallel \tau_\Psi[E \to \sigma_0]$$

f) Comunicaciones síncronas:

Recordemos que, con el fin de mejorar la expresividad de ALEPH, hemos diferenciado los operadores de comunicaciones síncronas y parametrización etiquetada de subprogramas. Sin embargo, no deja de ser una adición sintáctica soportada por los operadores de comunicación de LCEP. Por eso, aunque en ALEPH no sea sintácticamente correcto escribir $E-> canal; S$ (dado que la sintaxis correcta es $E ::> canal; S$) permitimos incluir la construcción anterior como un argumento aceptable para la definición de τ_Ψ, empleando únicamente el símbolo primitivo "$->$" para denotar el envío de un mensaje. Puesto que en la representación de la creación de entornos de ejecución para expresiones en las definiciones dadas para τ_Ψ para la asignación, llamada a procedimiento, condicional y bucle se ha empleado la notación primitiva, en la traducción de las sentencias de comunicación síncrona de salida se han dado como simples cambios de símbolo, del nuevo al inicial. Posteriormente, se aborda de modo unificado la traducción empleando "$->$".

f.1 Comunicaciones. Output:

$\tau_{\Psi_{6.1}}$

$$\tau_\Psi[A\ \ ::>\ \ canal; S] = \tau_\Psi[A \to canal; S]$$

$\tau_{\Psi_{6.2}}$

$$\tau_\Psi[::>\ \ canal; S] = \tau_\Psi[0 \to canal; S]$$

$\tau_{\Psi_{6.3}}$

$$\tau_\Psi[Var\ \ ::>\ \ canal; S] = \tau_\Psi[Var \to canal; S]$$

$\tau_{\Psi_{6.4}}$

$$\tau_\Psi[@F(R_1, R_2, \ldots, R_m)\ \ ::>\ \ canal; S] =$$

$$\tau_\Psi[F(R_1, R_2, \ldots, R_m) \to canal; S]$$

f.2 Comunicaciones. Input:

$\tau_{\Psi_{6.5}}$

$$\tau_\Psi[<::\ \ canal; S] = \lambda_{canal} x'.\tau_\Psi[S]$$

$\tau_{\Psi_{6.6}}$

$$\tau_\Psi[x\ \ <::\ \ canal; S] = \lambda_{canal} x'.\lambda_{asig} asig'.(asig' \circ \widehat{}_1\ x \circ \widehat{}_1\ x' \parallel \tau_\Psi[S])$$

$\tau_{\Psi_{6.7}}$

$$\tau_\Psi[u\ \ <::\ \ canal; S] = \lambda_{canal} u'.(\widehat{}_{global}\ (u \circ u') \circ \tau_\Psi[S])$$

g) Tratamiento de las comunicaciones a través de canales:

$\tau_{\Psi_{7.1}}$

$$\tau_\Psi[A \to canal; S] = (\widehat{}_{canal}\ A) \circ \tau_\Psi[S] \qquad A \in C \cup \{0\}$$

$\tau_{\Psi_{7.2}}$

$$\tau_\Psi[F(R_1, R_2, \ldots, R_n) \to canal; S] =$$

$$\tau_{\Psi_\lambda}[R_1, R_2, \ldots, R_n]\lambda_F F'.(\widehat{}_{canal}(F' \widehat{}_{\tau_\Psi}[F, R_1, R_2, \ldots, R_n]) \circ \tau_\Psi[S]) \parallel$$
$$\tau_\Psi[\tau_{\Psi_\sigma}[R_1, R_2, \ldots, R_n]]$$

donde los parámetros reales R_i pueden adoptar una de las siguientes formas:

$$R_i = \begin{cases} E_i & \text{instanciación posicional o estándar} \\ E_i \to j_i & \text{instanciación numérica } j_i \in \{1, 2, \ldots n\} \\ E_i \to x_i & \text{instanciación simbólica } x_i \in X \text{, donde } X \\ & \text{es el conjunto de parámetros formales de } F \end{cases}$$

De acuerdo con esta notación, podemos definir τ_{Ψ_σ}, τ_{Ψ_λ} y $\widehat{\tau_\Psi}$.

g.1 τ_{Ψ_σ}: Obtención del valor evaluado de los parámetros reales:

Esta función describe cómo deben crearse los entornos de ejecución asociados a cada parámetro real especificado en la parametrización.

$$\tau_{\Psi_\sigma}[R_1, R_2, \ldots, R_m] =$$

$$\begin{cases} \tau_{\sigma_{sec}}[R_1, R_2, \ldots, R_m] & \text{si } F(R_1, R_2, \ldots, R_m) \\ \tau_{\sigma_{par}}[R_1, R_2, \ldots, R_m] & \text{si } F\{R_1, R_2, \ldots, R_m\} \end{cases}$$

En ALEPH es posible efectuar la evaluación de los parámetros reales pasados a un subprograma tanto de forma secuencial como en paralelo. Por eso, dependiendo de la modalidad escogida por el programador, se define de una forma u otra la traducción. Esta es la razón por la que usamos $\tau_{\sigma_{sec}}$ y $\tau_{\sigma_{par}}$ para describir τ_{Ψ_σ} en los casos de evaluación secuencial y paralela, respectivamente. Observamos que τ_{Ψ_σ} es una función *parcial* $\tau_{\Psi_\sigma} : \Psi \to \Psi$ que transforma sentencias ALEPH en sentencias ALEPH. Esto se debe a que su utilización auxiliar para definir τ_Ψ incluye la aplicación de τ_Ψ al resultado de aplicar primero τ_{Ψ_σ}. Por tanto, también las funciones $\tau_{\sigma_{sec}}$ y $\tau_{\sigma_{par}}$ son funciones parciales que toman y devuelven valores en Ψ.

g.1.1 $\tau_{\sigma_{sec}}$:

$$\tau_{\sigma_{sec}}[R_i, R_{i+1}, \ldots, R_m] =$$

$$\begin{cases} \varepsilon \quad \text{si } m < i \\ \tau_{\sigma_{sec}}[R_{i+1}, \ldots, R_m] \quad \text{si } E_i \in C \\ \tau_{\sigma_{sec}}[T_{i_1}, \ldots, T_{i_{m'}}]\tau_{\sigma_{sec}}[R_{i+1}, \ldots, R_m] \\ \qquad \text{si } E_i = @G(T_{i_1}, \ldots, T_{i_{m'}}) \\ \leftarrow c_i; \tau_{\sigma_{sec}}[R_{i+1}, \ldots, R_m] \parallel \tau_{\sigma_{par}}[T_{i_1}, \ldots, T_{i_{m'}}]; \to c_i \\ \qquad \text{si } E_i = @G\{T_{i_1}, \ldots, T_{i_{m'}}\} \\ E_i \to \sigma_i; \tau_{\sigma_{sec}}[R_{i+1}, \ldots, R_m] \quad \text{en otro caso} \end{cases}$$

Como puede observarse, la definición presentada para $\tau_{\sigma_{sec}}$ constituye una ampliación de τ_σ. Se ha añadido el tratamiento de la inhibición de la evaluación. Lo que se hace es realizar una llamada recursiva a $\tau_{\sigma_{sec}}$ en la que se intercala la creación de entornos de ejecución para la evaluación de los parámetros de G, cuyo resultado se envía a los operadores de input convenientemente dispuestos por τ_{Ψ_λ}. Debemos reparar también en el tratamiento de la posibilidad de que se haya especificado que los parámetros de G se evalúen en paralelo entre sí. Esto se hace continuando la generación de entornos de ejecución secuenciales con el parámetro R_{i+1} tras haber recibido una señal de sincronismo a través del canal c_i, indicando el final de la evaluación (paralela) de

los parámetros para G. Esta sincronización es necesaria para mantener el orden de evaluación definido en la parametrización, ya que no debemos evaluar R_{i+1} hasta que no haya sido evaluado R_i. Pero la evaluación de R_i implica la evaluación de todos los parámetros de G.

g.1.2 $\tau_{\sigma_{par}}$:

$$\tau_{\sigma_{par}}[R_i, R_{i+1}, \ldots, R_m] =$$

$$\begin{cases} \varepsilon & \text{si } m < i \\ \tau_{\sigma_{par}}[R_{i+1}, \ldots, R_m] & \text{si } E_i \in C \\ \tau_{\sigma_{sec}}[T_{i_1}, \ldots, T_{i_{m'}}] \parallel \tau_{\sigma_{par}}[R_{i+1}, \ldots, R_m] & \\ \quad \text{si } E_i = @G(T_{i_1}, \ldots, T_{i_{m'}}) & \\ \tau_{\sigma_{par}}[T_{i_1}, \ldots, T_{i_{m'}}] \parallel \tau_{\sigma_{par}}[R_{i+1}, \ldots, R_m] & \\ \quad \text{si } E_i = @G\{T_{i_1}, \ldots, T_{i_{m'}}\} & \\ E_i \to \sigma_i \parallel \tau_{\sigma_{par}}[R_{i+1}, \ldots, R_m] & \text{en otro caso} \end{cases}$$

En ambos casos (g.1.1 y g.1.2) debemos mencionar que es fundamental la creación correcta, por parte del traductor, de las etiquetas σ_i y c_i asociadas a la evaluación de los parámetros o a la sincronización.

g.2 Obtención del valor evaluado de los parámetros reales:

Esta función complementa el trabajo realizado por τ_{Ψ_σ}, estableciendo cauces de recepción de los valores de los parámetros reales calculados en los entornos de ejecución.

$$\tau_{\Psi_\lambda}[R_i, R_{i+1}, \ldots, R_m] =$$

$$\begin{cases} \varepsilon & \text{si } m < i \\ \tau_{\Psi_\lambda}[R_{i+1}, \ldots, R_m] & \text{si } E_i \in C \\ \tau_{\Psi_\lambda}[T_{i_1}, \ldots, T_{i_{m'}}]\lambda_G G'.\tau_{\Psi_\lambda}[R_{i+1}, \ldots, R_m] & \\ \quad \text{si } E_i = @G(T_{i_1}, \ldots, T_{i_{m'}}) & \\ \lambda_{\sigma_i}\sigma'_i.\tau_{\Psi_\lambda}[R_{i+1}, \ldots, R_m] & \text{en otro caso} \end{cases}$$

Su definición es casi idéntica a la dada para τ_λ. Unicamente se añade el tratamiento de la inhibición de evaluación de parámetros. A destacar tan solo la inclusión de la recepción de la abstracción en G.

g.3 $\widehat{\tau_\Psi}$: Parametrización de la aplicación del subprograma:

Esta función define cómo debe parametrizarse la instancia de parametrización $P = \{P_1, P_2, \ldots, P_m\}$ obtenida para el subprograma. Implementa la función *param*

a partir de la información contenida en los parámetros reales R_i.

$$\widehat{\ }_{\tau_\Psi}[F, e_1, e_2, \ldots, e_{i-1}, R_i, R_{i+1}, \ldots, R_m] =$$

$$\begin{cases} \varepsilon & \text{si } m < i \\ \circ \widehat{\ }_{P_i} \ \widehat{\ \ }_{\tau_{\Psi_S}}[R_i]\widehat{\ }_{\tau_\Psi}[F, e_1, e_2, \ldots, e_{i-1}, e_i, R_{i+1}, \ldots, R_m] & \text{en otro caso} \end{cases}$$

Se puede observar que la única novedad respecto a la definición de su homóloga $\widehat{\ }_\tau$ radica en la inclusión de $\widehat{\ \ }_{\tau_{\Psi_S}}$ para describir cómo es la instancia de parametrización en cada parámetro formal. La necesidad de $\widehat{\ \ }_{\tau_{\Psi_S}}$ surge para soportar la inhibición de evaluación de parámetros. En el caso de $\widehat{\ }_\tau$, donde no se incluía esta característica, es suficiente con situar una variable ligada conectada con un canal de comunicación para la recepción de la instancia del parámetro ya evaluado. Ahora necesitamos que algunos parámetros no se evalúen por completo. Por lo demás, las modificaciones son mínimas:

$$e_i = \begin{cases} 1 & \text{si } R_i = E_i \\ j_i & \text{si } R_i = E_i \to j_i \\ x_i & \text{si } R_i = E_i \to x_i \end{cases}$$

representa, como en la definición de $\widehat{\ }_\tau$, las etiquetas e_i empleadas en cada parámetro real R_i para especificar el modelo de parametrización (función *param*).

$$P_i = \begin{cases} 1 & \text{si } e_i = 1 \\ o(j_i) & \text{si } e_i = j_i \\ F_{x_i} & \text{si } e_i = x_i \end{cases}$$

donde $o(j_i) = j_i - card\{k : e_k < j_i \ \wedge \ 1 \leq k < i\}$. P_i establece el comportamiento de *param* en función de e_i.

$$\widehat{\ \ }_{\tau_{\Psi_S}}[E_i] =$$

$$\begin{cases} (G' \widehat{\ }_{\tau_\Psi}[G, T_{i_1}, T_{i_2}, \ldots, T_{i_{m'}}]) & \text{si } E_i = @G(T_{i_1}, T_{i_2}, \ldots, T_{i_{m'}}) \\ \sigma'_i & \text{en otro caso} \end{cases}$$

Como puede observarse, en el caso de una inhibición de la evaluación preparamos la aplicación de la abstracción de G (que τ_{Ψ_λ} se encarga de entregar en G') sobre los m' argumentos especificados para G. Sin embargo, puesto que el resultado de esa aplicación se utiliza como uno de los parámetros reales de F, aunque se haya instanciado G totalmente, no se lleva a cabo ninguna computación que involucre procesos del sistema ya que no hay ningún túnel que comunique el entorno de ejecución de G con los procesos del sistema ni con los procesos de alcance global. En el caso de un parámetro real normal, el tratamiento coincide con el definido para $\widehat{\ }_\tau$.

h) Constructores de proceso:

$\tau_{\Psi_{8.1}}$

$$\tau_\Psi[\textbf{random } ?C_1; S_1 \ ?C_2; S_2 \ \dots \ ?C_n; S_n] = \sum_{i=1}^{n} \tau_\Psi[C_i; S_i; S]$$

La traducción de la sentencia de elección indeterminista es directamente la "suma" de la traducción de sus sentencias encadenadas. Si se considera que C_i puede ser una sentencia de envío o recepción de mensajes, es fácil ver que la traducción propuesta hace que la traducción sea un proceso comunicante, donde cada componente $\tau_\Psi[C_i; S_i; S]$ es un emisor elemental o un receptor elemental. Es interesante reparar en el tratamiento de la composición secuencial de la sentencia con S. Se añade S a cada sentencia S_i que acompaña a las sentencias de comunicación síncrona, de modo que, sea cual sea la escogida para proseguir la ejecución de acuerdo con la regla de comunicación paralela, la sentencia S se ejecuta cuando se espera.

$\tau_{\Psi_{8.2}}$

$$\tau_\Psi[(S_1 \parallel S_2 \parallel \cdots \parallel S_n); S] =$$

$$\begin{array}{l} \tau_\Psi[S_1; 0 \to c_S] \parallel \tau_\Psi[S_2; 0 \to c_S] \parallel \cdots \parallel \tau_\Psi[S_n; 0 \to c_S] \parallel \\ \parallel \lambda_{c_S} x'.\lambda_{c_S} x'. \overset{n}{\cdots} .\lambda_{c_S} x'.\tau_\Psi[S] \end{array}$$

i) Otras:

$\tau_{\Psi_{9.1}}$

$$\tau_\Psi[\varepsilon] = 0$$

$\tau_{\Psi_{9.2}}$

$$\tau_\Psi[\textbf{return } f] = \lambda_f f'.f'$$

Recordemos que **return** es una pseudo-sentencia, en el sentido de que no forma parte de la sintaxis de ALEPH.

B.3.1 Funciones y procedimientos primitivos

La presentación de ALEPH realizada cubre los rasgos fundamentales de un lenguaje de programación imperativo orientado a la programación estructurada. El tratamiento de las características esenciales de este tipo de lenguajes está cubierto completamente con lo expuesto hasta aquí. Sin embargo, queda por abordar algunos detalles, como el tratamiento de la entrada/salida, las posibilidades de cálculo aritmético, los operadores relacionales para definir expresiones lógicas básicas, los operadores lógicos para la construcción de expresiones más complejas, etc.

Puesto que la máquina abstracta donde se ejecutan los programas es, en cierto modo, programable a su vez, mediante la definición de reglas de Ψ-evaluación adecuadas,

tenemos un amplio margen de maniobra a la hora de plantear las funciones primitivas del lenguaje. La sintaxis de ALEPH introduce un marco general de trabajo, dentro de la metodología de la programación estructurada, dando soporte a características de la programación paralela, como la composición paralela de sentencias y la comunicación. Las herramientas fundamentales para construir los programas son las funciones y los procedimientos. Respecto a ellas, la gramática establece la sintaxis adecuada para la activación de los subprogramas, bien sea como sentencias o como parte de una expresión. Unicamente se han añadido, como base mínima para la construcción de estas últimas, los operadores aritméticos habituales: suma ($+$), resta ($-$), multiplicación ($*$), división ($/$) y potencia ($\hat{}$). Lo mismo ocurre en el caso de los operadores relacionales ($=, <>, <, <=, >=, >$) y lógicos (and, or, xor). Respecto a la entrada/salida, empleamos en los ejemplos *print* como una función capaz de mostrar por pantalla el término que se le pasa como argumento.

Lo verdaderamente importante para hacer operativo un subprograma primitivo es definir:

- por un lado, un proceso del sistema como una abstracción capaz de instanciar los argumentos que recibe, un l-proceso que contiene una constante $a \in C$ (por ejemplo, *suma*, *resta*, *and*, ...) capaz de activar las reglas de Ψ-evaluación con las que se da soporte real a la acción que deseamos asignar a dicho subprograma, y
- una regla de Ψ-evaluación que trate adecuadamente las ocurrencias de dicha constante y los símbolos que la rodean en el λ_{EP}-término donde se encuentre.

Esta flexibilidad de las últimas características de ALEPH puede presentar grandes posibilidades de explotación, en particular puede permitir su extensión para tratar tipos de datos. De hecho, entendemos ALEPH como un marco genérico de explotación de las características de cierta clase de lenguajes de alto nivel bajo la tutela de la máquina abstracta extendida X_Ψ, cuyo núcleo fundamental es LCEP.

La función de traducción se ha desarrollado sobre un entorno PC utilizando Arity Prolog como lenguaje de implementación. Sobre el producto obtenido se han realizado diferentes y abundantes pruebas, entre las que se incluyen todos los ejemplos presentados en la memoria. Los resultados han sido muy satisfactorios y confirman la utilidad de LCEP como ejecutador de sentencias de un lenguaje de alto nivel que incorpora mecanismos para representar el paralelismo y la comunicación entre los diferentes elementos del sistema.

Índice de Materias

α-conversión, 48
α-renombramiento, 28
β-comunicación, 50
β-comunicación paralela, 50, 57
β-comunicación secuencial, 50
β-conversión, 2, 3
β-reacción, 9
β-reducción, 27, 28, 40
β_P-comunicación paralela, 53
γ-cálculo, 4, 9, 11
λ-cálculo, 1–3, 9, 10, 16, 19, 26, 27, 39, 40, 42, 48, 63, 64, 131
λ-cálculo Etiquetado, 3, 25, 27–29, 39
λ-cálculo Etiquetado Paralelo, 25, 60, 125
λ_{EP}-términos, 41, 52
π-cálculo, 4, 19, 22, 23, 29, 32, 35, 52, 53, 58, 60, 62, 67, 71
π-cálculo monádico, 19
π-cálculo poliádico, 22
ξ-conversión, 2
kill, 114
l-proceso, 87, 99, 102, 103, 159, 180
m-proceso, 87, 91, 93, 97, 99, 103, 111, 116, 159
s-proceso, 87, 99
$\mathcal{H}$-evaluación, 92, 103, 114, 119
$\mathcal{T}$-comunicabilidad, 77
$\mathcal{T}$-comunicabilidad sobre un canal, 57
$\mathcal{T}$-comunicación, 87
$\mathcal{T}$-comunicación potencial, 56

árbol sintáctico, 44–46, 50, 55, 91
árbol sintáctico simétrico, 46

abstracción, 2, 9, 22, 23, 25, 27, 41, 63, 91, 99, 116
abstrayente, 26, 28
agente, 11
alcance, 91, 111, 112, 114, 148
ALEPH, 4, 62, 85, 125, 131, 141, 155
aplicación, 25, 26, 92
aplicador, 26, 28, 64
arcos, 45
aridad, 43
asignación, 87, 115, 142, 149
axiomas de reducción, 50
axiomas de reordenación, 42

bisimilaridad, 24
bisimulación, 4, 14, 15, 18, 21, 24
bucle, 87, 150

call by name, 148
call by value, 147
camino, 46, 53
camino elemental, 46, 55
camino elemental contiguo, 89
canal, 3, 10, 11, 26, 28–30, 35, 39, 40, 42, 53, 58, 85, 99, 121
canal privado, 93
canales activos, 50
canales de entrada, 50
canales de salida, 50
canales numéricos, 61

canales simbólicos, 91
canales soporte, 49
CCS, 3, 9, 10, 16, 34
CHAM, 4, 7, 9, 10, 29, 30, 34, 60
CHOCS, 4, 16, 17, 29, 33, 37, 60
comportamiento, 28
comportamiento observacional, 18
comportamiento operacional, 21
composición, 20, 64
composición paralela, 3, 16, 17, 64, 87, 115
composición secuencial, 48, 115, 116
compromisos, 23
computación concurrente, 19
computación concurrente asíncrona, 7
computación funcional, 67
comunicación, 2, 3, 9, 10, 12, 14, 17, 20, 23, 26, 35, 39, 41, 52, 53, 85, 92, 131
comunicación paralela, 108
comunicación síncrona, 120
comunicación secuencial, 42
concreción, 22, 23
concurrencia, 2, 3, 10, 28, 131
condición de reemplazamiento, 68
condicional, 87, 150
confluencia, 28
congruencia, 11, 12, 15, 20, 22–25, 48, 52, 64, 81
conjunto de etiquetas de un túnel, 57
constante, 91, 116, 143
contextos, 10, 21
continuación, 23
cooperación, 10, 13, 14
CSP, 2, 4
cuerpo, 26
currificación, 26, 39, 51

datos, 144
declaración, 143
dinámica, 23

efecto lateral, 114, 117
elección no determinista, 16, 17, 26, 30, 40, 41, 64
emisión, 3
emisor, 9, 53
emisor elemental, 49
entorno de ejecución, 94, 99, 112, 114, 116, 117, 119, 121, 123, 173
envío, 10, 16
equivalencia, 15, 18, 21
equivalencia de reducción, 21
equivalencia estructural, 9
equivalencia observacional, 15, 18
equivalencias de tests, 10
esquemas de reglas, 8
estado estable, 9, 10
estructuras de control, 144
etiquetas, 3, 25
etiquetas numéricas, 25–28, 39, 40, 51, 61, 62, 146
etiquetas simbólicas, 25–28, 39, 40, 51

filtro, 58, 63, 88, 91, 93
flujo de control, 142
forma normal, 10, 68, 92, 113
función de determinación simbólica, 46
función de reemplazamiento, 68
función de traducción, 5
funciones primitivas, 102

grafo, 45, 46, 78
grafo de $\mathcal{T}$-comunicabilidad, 80
gramática, 5

ión, 7, 9, 30
input, 3, 16, 42, 43
instrucción elemental, 142
interleaving, 10, 11, 13

lógica combinatoria, 1
LCEP, 3, 4, 29, 39, 40, 52, 59, 67, 85, 131, 141, 155
lenguaje Γ, 7
ley de interacción, 3, 11, 13
ley de la membrana, 8
ley de la válcula, 8
ley de reacción, 8
ley química, 8
leyes estructurales, 7
ligadura, 11–13, 17, 19
llamada a procedimiento, 87, 149
localización, 23
localización de un túnel, 55

máquina de Turing, 1
membrana, 7, 9, 10
memoria abstracta, 91, 114, 115
modelo, 19
modelo de Plotkin, 16
molécula, 7–9, 30
monoide conmutativo, 20, 48
movilidad, 19
movimiento browniano, 7

no determinismo, 16
nodos, 45
nombre, 10, 11, 17, 19, 20, 22, 26, 42, 53
nombres libres, 20
nombres ligados, 20

objeto, 19
observabilidad, 4, 21
observable, 21
ocurrencia, 44
operación, 142
orden de ejecución, 142, 150
orden de precedencia, 17
orden de prioridad, 41
output, 3, 16, 42, 43

par complementario, 49
par comunicante, 49
parámetros formales, 41, 42, 99, 144
parámetros reales, 41, 42, 107, 121, 145
paralelismo, 17, 19, 23, 40, 41, 143
preórdenes, 18
prefijo, 16, 17, 19, 21, 63
preorden de prueba, 10
proceso, 1, 3, 11, 16, 19, 21, 26, 41, 44, 53, 58, 87, 99, 155
proceso comunicante, 49, 52, 87
proceso comunicante elemental, 49
proceso comunicante maximal, 78
proceso del sistema, 88
proceso inactivo, 16
proceso nulo, 40, 41, 63, 99, 104, 114, 120, 121
procesos básicos, 91
procesos concurrentes, 10
procesos del sistema, 102
procesos normales, 19, 22, 23
programa, 7, 141
programación estructurada, 142

reacción, 31
recepción, 3, 10, 11, 16
receptor, 9, 53
receptor elemental, 49, 91
redes de Petri, 4
redex, 92
reducción, 21, 23, 26, 92
reemplazamiento, 68
reescritura, 67
reescritura sensible al contexto, 67
reglas de calentamiento, 8
reglas de enfriamiento, 8
reglas de evaluación, 102, 113
reglas de inferencia, 20, 52
reglas de reacción, 7–9
reglas de reducción, 27

reglas de transformación, 7–9
reglas locales, 26
regularidad, 68
relación de $\mathcal{T}$-comunicabilidad, 78
relación de $\mathcal{T}, P$-comunicabilidad, 77
relación de derivación, 16
relación de equivalencia, 4
relación de orden, 42, 91, 110
relación de reducción, 20, 52, 64, 67
relación de reescritura en un paso, 68
relación de reescritura sensible al contexto, 4
relación de transición, 12, 13, 15
renombramiento, 16–18
reordenación, 28, 41
reordenamiento, 27
replicación, 19, 40, 41, 48, 52, 64, 87
restricción, 10, 16, 17, 19, 20, 23, 58, 64

s-túnel, 55
secuencialidad, 2, 40, 41
semántica declarativa, 21, 24
semántica operacional, 4, 16–18, 24
signatura, 45, 46
signatura homogénea, 43
simulación, 14
sincronización, 17
sintaxis, 4, 9, 11, 17, 19, 22, 40, 157
sistema de reducción, 28
sistema de transición, 16
sistema de transición de estados, 17
sistema de transición etiquetado, 4, 16, 23
sistemas de inferencia, 67
sobrecarga, 148
solución inerte, 9, 10
soluciones, 7
soluciones químicas, 7
subproceso comunicante, 49
subprograma, 91, 98, 144
subprograma anónimo, 107, 108
subprogramas recursivos, 108
subtérmino, 45
sujeto, 19
sup-adyacencia de una ocurrencia, 88
sustitución, 11
sustitutividad, 12

término cerrado, 12
túnel, 40, 46, 50, 53, 55, 59, 85, 87, 91, 123
túnel P-transparente, 57
túnel transparente, 108
teoría de autómatas, 4
tipos, 143
transiciones etiquetadas, 12

válvula, 8, 9
variables, 143
variables globales, 99, 115, 144
variables libres, 11, 42
variables ligadas, 12, 16, 48, 114, 116
variables locales, 99, 107, 115, 144