MAPiCO: Máquina Abstracta para el Cálculo PiCO

ANTAL ALEXANDER BUSS MOLINA
MAURICIO HEREDIA RAMIREZ

PONTIFICIA UNIVERSIDAD JAVERIANA
FACULTAD DE INGENIERIA
INGENIERIA DE SISTEMAS Y COMPUTACION
SANTIAGO DE CALI
1999

MAPiCO: Máquina Abstracta para el Cálculo PiCO

ANTAL ALEXANDER BUSS MOLINA
MAURICIO HEREDIA RAMIREZ

Tesis de grado para optar el título de
Ingeniero de Sistemas y Computación

Director
GABRIEL TAMURA MORIMITSU
Ingeniero de Sistemas y Computación

PONTIFICIA UNIVERSIDAD JAVERIANA
FACULTAD DE INGENIERIA
INGENIERIA DE SISTEMAS Y COMPUTACION
SANTIAGO DE CALI
1999

Santiago de Cali, Febrero 25 de 1999

Ingeniero
ANDRES JARAMILLO BOTERO
Decano Académico de la Facultad de Ingeniería
Pontificia Universidad Javeriana
Ciudad

Certifico que el presente proyecto de grado, titulado "MAPiCO: Máquina Abstracta para el Cálculo PiCO" realizado por ANTAL ALEXANDER BUSS MOLINA y MAURICIO HEREDIA RAMIREZ, estudiantes de Ingeniería de Sistemas y Computación, se encuentra terminado y puede ser presentado para sustentación.

Atentamente,

Ing. GABRIEL TAMURA MORIMITSU
Director del Proyecto

Santiago de Cali, Febrero 25 de 1999

Ingeniero
ANDRES JARAMILLO BOTERO
Decano Académico de la Facultad de Ingeniería
Pontificia Universidad Javeriana
Ciudad

Por medio de ésta, presentamos a usted el proyecto de grado titulado "MAPiCO: Máquina Abstracta para el Cálculo PiCO" para optar el título de Ingeniero de Sistemas y Computación.

Esperamos que este proyecto reúna todos los requisitos académicos y cumpla el propósito para el cual fue creado, y sirva de apoyo para futuros proyectos en la Universidad Javeriana relacionados con la materia.

Atentamente,

ANTAL ALEXANDER BUSS MOLINA

MAURICIO HEREDIA RAMIREZ

ARTICULO 23 de la Resolución No 13 del 6 de Julio de 1946
del Reglamento de la Pontificia Universidad Javeriana.

"La Universidad no se hace responsable por los conceptos emitidos por sus alumnos en sus trabajos de Tesis. Sólo velará porque no se publique nada contrario al dogma y a la moral Católica y porque las Tesis no contengan ataques o polémicas puramente personales; antes bien, se vea en ellas el anhelo de buscar la Verdad y la Justicia"

Nota de Aceptación:

Aprobado por el comité de Trabajo de Grado en cumplimiento de los requisitos exigidos por la Pontificia Universidad Javeriana para optar el título de Ingeniero de Sistemas y Computación.

ANDRES JARAMILLO BOTERO
Decano Académico de la Facultad de Ingeniería

PABLO GRECH MAYOR
Director de la Carrera de Ingeniería de Sistemas y Computación

GABRIEL TAMURA MORIMITSU
Director de Tesis

CAMILO RUEDA
Jurado

ANDRES JARAMILLO
Jurado

Antal Alexander Buss Molina

A mi familia y a todos ustedes, mis amigos.

Mauricio Heredia Ramírez

A mis padres y mi hermano.

A mis familiares y amigos.

Agradecimientos

Los autores expresan sus agradecimientos:

- A Gabriel Tamura M. profesor de la carrera de Ingeniería de Sistemas y Computación de la Universidad Javeriana, director del proyecto, por sus ideas y ayuda en la elaboración del mismo.
- A Camilo Rueda, Profesor de la carrera de Ingeniería de Sistemas y Computación de la Universidad Javeriana, asesor del proyecto por sus comentarios acerca del presente trabajo.
- A Frank Valencia, por su ayuda en las etapas iniciales.
- Al grupo AVISPA.
- A todas aquellas personas que de una u otra forma colaboraron en la realización del presente trabajo.

Indice General

Indice de Tablas

Indice de Figuras

Indice de Anexos

Resumen

El cálculo PiCO (A Calculus of Concurrent Constraint Objects for Musical Applications) [ADQ+98] es un cálculo que integra objetos concurrentes y restricciones como elementos básicos, en contraste con algunos cálculos como TyCO [Vas94a] y el cálculo π [Wal91, MPW92]. En PiCO, el modelo de objetos es extendido con el manejo de restricciones de una manera ortogonal (es decir que la noción de objetos sin restricciones es derivable) adicionando la noción de sistema de restricciones encontrado en el cálculo ρ [NM95] y la noción de delegación de mensajes. La extensión del cálculo pretende adoptar una manera natural de expresar comportamientos de comunicación con el uso estándar de la sincronización por el método de paso de mensajes.

El objetivo de este trabajo es crear una máquina abstracta para dicho cálculo, diseñando el manejo de memoria, almacenamiento y la reducción de los programas, sin pretender profundizar en el sistema de restricciones; manteniendo una equivalencia semántica entre la máquina y el cálculo de tal manera que la especificación y la reglas de reducción de la máquina abstracta sigue las del cálculo PiCO.

INTRODUCCION

Al definir un lenguaje de programación, lo ideal es que él tenga una base teórica formal tal como un cálculo computacional, para poder demostrar formalmente propiedades tales como correctitud y completitud por medio de la matemática.

El grupo AVISPA está desarrollando un lenguaje de programación visual (CORDIAL) [QRT97b, QRT97a] que integra los paradigmas de restricciones, concurrente y orientado a objetos. También ha desarrollado un cálculo computacional llamado PiCO para ser usado como base formal del lenguaje.

Se diseñó y desarrolló la implementación de una máquina abstracta para el cálculo PiCO, definiendo las estructuras de soporte, el formato de las instrucciones, la codificación y el comportamiento interno.

En el capítulo 1 del presente trabajo se presenta un breve vistazo a algunos conceptos básicos y se explica qué son y para qué sirven las máquinas virtuales y abstractas, además de las diferencias entre algunas máquinas implementadas. También, se hace un recuento de algunos cálculos computacionales, sus máquinas abstractas y los lenguajes que los utilizan. En el capítulo 2, se da una descripción de Cordial, que es un lenguaje visual que integra los paradigmas de restricciones, concurrente y orientado a objetos, lenguaje que utiliza y es en cierta medida la justificación de nuestro trabajo de grado. En el capítulo 3 se expone el cálculo PiCO, su sintaxis, semántica y aplicación; se hace un mayor énfasis en este cálculo ya que es el formalismo matemático que la máquina debe respetar. En el capítulo 4, se da una descripción del diseño de la máquina, lo que es un programa para ella, cómo hace el almacenamiento y el manejo de memoria, la reducción de un programa y se nombra la utilización del sistema de restricciones. En el capítulo 5 se da algunos detalles

de implementación como: estructuras de datos usadas, estructuras de los procesos, recolectores de basura, lenguaje de programación usado, algoritmos y complejidad entre otros. Por último en el capítulo 6 se describen algunas pruebas realizadas a la implementación de la máquina, donde se muestra el resultado de la ejecución y se comprueba que el estado deseado y el estado arrojado por la máquina después de la ejecución de un programa es el mismo.

Los archivos fuente del documento fueron desarrollados en LaTeX 2_ε[BOP+96, Lam94], para tal efecto se ha desarrollado una propuesta de una clase de documento de tal manera que sirva como estándar en la presentación de tesis de grado y de anteproyectos de la carrera de Ingeniería de Sistemas y Computación en la Pontificia Universidad Javeriana de Cali. Este tipo de documento está acorde con los estándares internacionales para tesis de grado en áreas tecnológicas.

1 GENERALIDADES

Este capítulo muestra los cálculos computacionales en los cuales es basado en parte el cálculo PiCO, dando un breve vistazo (en algunos casos) a la implementación y especificación de su máquina abstracta; además, se especifica qué lenguajes de programación la usan. También en este capítulo se dan algunos conceptos básicos necesarios para el buen entendimiento de este trabajo.

1.1 CONCEPTOS BASICOS

1.1.1 Traductores y Software de Simulación de Computadores

En teoría es posible construir un computador hardware o firmware para ejecutar directamente programas escritos en cualquier lenguaje de programación particular, y así construir un computador LISP, APL, o PL/1, pero no es normalmente económico construir tal máquina [Pra84]. Consideràciones practicas tienden a favorecer los computadores actuales con un lenguaje de máquina de más bajo nivel, ganando con esto velocidad, flexibilidad y costo. La Programación, por supuesto, es más a menudo hecha en un lenguaje de alto nivel alejado del lenguaje de máquina mismo. La pregunta que resulta es: Cómo tener programas en un lenguaje de alto nivel que se ejecuten en un computador, sin considerar el lenguaje de máquina?

Hay dos soluciones básicas: Traducción (compilación) y software de simulación (interpretadores).

Un traductor podría ser diseñado para traducir programas del lenguaje de alto nivel

a uno equivalente en el lenguaje de máquina del computador.

Un software de simulación o interpretador simula una máquina, cuyo lenguaje es de alto nivel, en otra de más bajo nivel. Para hacer esto, se construye un conjunto de programas en el lenguaje de máquina del computador en el que se va a trabajar los cuales representan los algoritmos y las estructuras de datos necesarias para la ejecución en el lenguaje de alto nivel.

1.1.2 Máquinas Virtuales e Implementación de Lenguajes

Si los lenguajes de programación fueran originalmente definidos en términos de su máquina real, cada lenguaje se asociaría en principio con una única máquina.

Cuando un lenguaje de programación está siendo implementado en una máquina en particular, el desarrollador primero determina la máquina virtual que representa la semántica de su lenguaje para luego escoger una vía particular de construir esa máquina virtual fuera de los elementos de hardware y software provistos por el computador real.

1.1.3 Procesos Móviles

Los procesos móviles [MPW90] son procesos cuyas estructuras pueden cambiar, es decir que no sólo los componentes computacionales de un sistema están arbitrariamente enlazados sino que una comunicación entre componentes vecinos puede llevar información que cambia sus enlaces.

Los modelos desarrollados más formalmente como las redes de petri, CSP, ACP, CCS pueden expresar esta movilidad indirectamente. Por otro lado tenemos modelos que expresan movilidad directa pero aún esperan un análisis matemático de sus conceptos básicos, como el modelo de Actores de Hewitt. En dichos modelos la movilidad es dada permitiendo que los procesos mismos se pasen como valores en una comunicación. Estos modelos están fuera del alcance del presente trabajo.

1.2 CALCULOS COMPUTACIONALES, MAQUINAS VIRTUALES Y ABSTRACTAS

1.2.1 Cálculos Computacionales

A mediados de los 60's se pudo observar que un lenguaje de programación no trivial puede ser entendido en términos de un "lenguaje núcleo" o lenguaje base simple que captura el mecanismo esencial de una colección de estilos de programación, junto con una colección de "formas derivadas" o acciones cuyo comportamiento se define mediante traducción al lenguaje núcleo.

Los cálculos computacionales a diferencia de los lenguajes núcleo, modelan matemática, sólida y formalmente los lenguajes de programación de tal forma que se acomoden, adapten y sigan la estructura de dicho cálculo.

Un cálculo computacional está caracterizado por su sintaxis, que son las reglas que prescriben la estructura de una fórmula bien formada, y su semántica, que son formalismos que interpretan o reducen fórmulas.

Usualmente, para asegurar que un lenguaje sea completo, consistente y formal, se debe modelar matemáticamente con un cálculo computacional, de tal forma que dicho cálculo sea un soporte para este lenguaje o su lenguaje núcleo.

A continuación se consideran varios cálculos computacionales que han servido de base para algunos lenguajes de programación.

1.2.2 Cálculo λ

El cálculo λ o cálculo lambda [Nie96, Pie95] es un sistema formal en el cual todas las computaciones son reducidas a operaciones de definición de funciones y aplicaciones. Desde los 60's el cálculo λ ha sido ampliamente usado en la especificación de características de lenguajes de programación, diseño e implementación de lenguajes y en el estudio de sistemas de tipos.

La abstracción procedimental es la característica principal de la mayoría de los

lenguajes de programación. En el cálculo λ todo es una función (ver la tabla 1.1), los argumentos aceptados por las funciones son a su vez funciones y el resultado retornado por una función es otra función.

L, M, N	$::=$	x	Variable
		MN	Aplicación
		$\lambda x.M$	Abstracción

Tabla 1.1: Sintaxis del Cálculo Lambda

En su forma pura, el cálculo λ no tiene constantes ni operadores (e.d. no tiene números, operaciones aritméticas, registros, ciclos, secuenciación, entrada y salida, etc). Los números, por ejemplo, son codificados como funciones especiales que pueden ser interrogados para saber que número representan.

Como un ejemplo de una reducción en el cálculo λ tenemos:

Inicialmente se definen las expresiones booleanas $Verdadero$ y $Falso$:

$$Verdadero = \lambda t.\lambda f.t$$
$$Falso = \lambda t.\lambda f.f$$

Ambas expresiones son combinaciones, ninguno contiene una variable libre. Esto quiere decir que son inertes respectos con la sustitución: $[N/x]Verdadero = Verdadero$ sin importar que sean N o x. La única forma de "interactuar" con las combinaciones es la aplicación de estas en otras expresiones. Por ejemplo en la definición combinatoria del If con la propiedad de $If\ L\ M\ N$ reduce a M cuando $L = Verdadero$ y reduce a N si $L = Falso$.

$$If = \lambda l.\lambda m.\lambda n.\ l\ m\ n$$

El valor booleano L es en si la condición. Si se tiene la expresión $If\ Verdadero\ M\ N$ se reduce como sigue:

$IfVerdaderoM\ N$	$=$	$\underline{(\lambda l.\lambda m.\lambda n.\ l\ m\ n)\ Verdadero\ M\ N}$	Por definición
	$\rightarrow$	$\underline{(\lambda m.\lambda n.\ Verdadero\ m\ n)\ M}\ N$	Reducción
	$\rightarrow$	$\underline{(\lambda n.\ Verdadero\ M\ n)\ N}$	Reducción
	$\rightarrow$	$\underline{Verdadero\ M\ N}$	Reducción
	$=$	$\underline{(\lambda t.\lambda f.t)\ M}\ N$	Por definición
	$\rightarrow$	$\underline{(\lambda f.M)\ N}$	Reducción
	$\rightarrow$	M	Reducción

1.2.3 Cálculo π, π^+ y la Máquina Abstracta PICT

El Cálculo π [Mil91, Pie96, Tur95, Wal91, Wal95, Pie95] es un modelo formal de computación concurrente basada en la noción de nombres.

Análogamente al cálculo λ donde todo es una función, en el cálculo π todas las expresiones denotan un proceso (una actividad computacional, se ejecuta concurrentemente con otros procesos y que posiblemente contiene muchos subprocesos independientes). Dos procesos pueden interactuar intercambiando un mensaje por un canal. La comunicación entre canales es el único mecanismo de computación, de igual manera que la aplicación de funciones lo es para el cálculo λ. La única cosa que puede ser observada acerca del comportamiento de los procesos es su habilidad de enviar y recibir mensajes.

P, Q, R	$::=$	O	Proceso Nulo
		$x(y)P$	Lector
		$xy.P$	Escritor
		$P\|Q$	Composición Paralela
		$(vx)P$	Restricción
		$!P$	Replicación

Tabla 1.2: Sintaxis del Cálculo Pi

La expresión más simple en el π cálculo síncrono es el "proceso nulo" **0** (ver la tabla 1.2), el cual denota un proceso que no tiene comportamiento. Además, si P es un proceso, entonces la expresión $x(y).P$ denota un proceso que espera para leer un valor y de un canal x; habiendo recibido éste, se comporta como $P[y/a]$ donde a son los parámetros actuales. Similarmente, $\bar{x}y.P$ denota un proceso que primero espera

a enviar el valor de y hacia el canal x y entonces luego de que y ha sido aceptado por algún proceso de entrada, se comporta como P.

$P|Q$ denota un proceso compuesto de dos subprocesos, P y Q, ejecutándose concurrentemente.

Poniendo un operador de ligadura (νx) antes de un proceso P, asegura que x es un canal libre en P. (e.d. que los mensajes enviados y recibidos por P en x nunca serán mezclados con mensajes enviados o recibidos en algún otro canal de nombre x creado en otra parte). El nombre alfabético del canal no es importante: el proceso $(\nu x)\bar{y}x.\mathbf{0}$ es completamente equivalente a $(\nu z)\bar{y}z.\mathbf{0}$) (ambos introducen un canal libre, diferentes de todos los otros canales y envían este en y).

Finalmente, el "proceso replicado" $!P$ se entiende como un infinito número de copias de P, todas corriendo en paralelo. Típicamente, sólo unas pocas copias estarán haciendo algo en un momento determinado; $!P$ podría realmente concebirse como un simple dispositivo notacional para describir procesos con secuencia de comportamiento de longitud infinita.

La definición de variables libres y sustitución en el cálculo π son similares a la correspondiente definición en el cálculo λ. La expresión $x(y).P$ y $(\nu x)P$ ata la variable x en el cuerpo P; como en el cálculo λ, se deben renombrar las variables cuando sea necesario. La sustitución es más simple aquí, porque sólo se sustituyen variables por variables (no se sustituyen procesos por otros procesos).

Por ejemplo, los valores booleanos $Verdadero$ y $Falso$ son codificados por procesos replicados (sobre el canal b) que acepta un par de canales t y f y responde mandando un mensaje a t o f según sea el caso.

$$\begin{aligned} Verdadero(b) &= !b(t,f).\bar{t}() \\ Falso(b) &= !b(t,f).\bar{f}() \end{aligned}$$

Si se define el proceso $Test(b)$ como:

$$Test(b) = (\nu t)(\nu f)\bar{b}(t,f).(t().P \mid f().Q)$$

Luego se realiza la composición de procesos $Verdadero(b) \mid Test(b)$ que reduce a $Verdadero(b) \mid P \mid (vt)(vf)(O \mid f().Q)$, mientras que $Falso(b) \mid Test(b)$ reduce a $Falso(b) \mid Q \mid (vt)(vf)(O \mid t().P)$ donde el tercer subproceso queda sin posibilidad de participar en futuras acciones.

El cálculo π^+ [VDR97a, VDR97b] es una extensión ortogonal del cálculo π con restricciones, donde la noción de agentes de restricciones es adicionado al agente estándar del cálculo π. En el cálculo π^+ también se extiende la noción de equivalencia de reducción [VDR97b] (barbed bisimulation [MS92]) para definir la equivalencia comportamental en el cálculo extendido y entonces usa la noción extendida para caracterizar algunos comportamientos equivalentes derivados de los agentes de restricciones.

La sintaxis del cálculo π^+ adiciona agentes de restricciones a los agentes estándar del cálculo π (ver la tabla 1.3). Los agentes de restricciones desarrollan las operaciones ask y tell del lenguaje CCP (Concurrent Constraint Programing).

La semántica del cálculo π^+ es definida operacionalmente siguiendo el sistema de transición para el modelo cc (concurrent constraint) definido en [Sar93].

Los agentes de restricciones son un nuevo tipo de agentes cuyo comportamiento depende de un almacenamiento o área de trabajo global. Un área de trabajo contiene información dada por las restricciones. El agente "tell" adiciona una restricción al sistema; el agente "ask" por su parte activa un proceso si una restricción dada es una consecuencia lógica de la información en el área de trabajo.

El cálculo π^+ está parametrizado en un sistema de restricciones y es independiente de un dominio de restricciones particular.

PICT [PT94, Pie96, PT96] es una máquina abstracta que implementa el cálculo π donde la ejecución concurrente es simulada.

Pict compila el código fuente a código C, en lugar de un código nativo, permitiendo generar código eficiente sin sacrificar portabilidad además de poder incrustar código en C dentro de programa Pict para así poder utilizar las librerías de funciones disponibles en C.

Procesos Normales: M, N	::=	O	Proceso nulo
	|	$\pi.P$	Prefijo
	|	$M + N$	Suma
Proceso de restricción: R	::=	$!\phi.P$	Proceso *tell*
	|	$?\phi.P$	Proceso *ask*
Procesos: P, Q	::=	$(\upsilon x)P$	Variable nueva x en P
	|	$(\upsilon a)P$	Nombre nuevo a en P
	|	N	Procesos normales
	|	$P \mid Q$	Composición
	|	$*P$	Procesos replicados
	|	R	Proceso de restricción
Canales: C	::=	a	Nombre
	|	v	Valor
	|	x	Variable
Prefijo: π	::=	$C?[x_1 \ldots x_n]$	Prefijo lector
	::=	$C![C_1 \ldots C_n]$	Prefijo escritor

Tabla 1.3: Sintaxis de Pi$^+$

Aunque el cálculo π es no determinístico, la implementación de él (PICT), es determinístico, debido a que este comportamiento surge naturalmente a causa de la dependencia del tiempo entre la máquina abstracta y el sistema operativo (por ejemplo, durante la entrada/salida o el manejo de interrupciones).

Para la implementación de PICT se han hecho dos simplificaciones al cálculo π. Primero en el operador de replicación $*P$ el proceso P puede ser sólo un proceso de entrada (lectores) es decir, *x(v)P*. Esto para hacer la implementación significativamente más simple ya que es más fácil detectar cuando se necesita crear una nueva copia de un proceso replicado. La segunda simplificación es deshabilitar la operación suma "+" encontrada en el cálculo para hacer más simple la implementación de la comunicación. La sola presencia de la suma en el cálculo puede duplicar la cantidad de almacenamiento requerido para los canales. La experiencia presentada con el lenguaje de programación PICT sugiere que el uso de la suma es poco frecuente,

y de requerirla se obtiene por medio de una librería.

Las colas de canales son el componente principal de la máquina. Los elementos suspendidos en una cola de canales pueden ser lectores, escritores o lectores replicados.

Un estado de máquina se caracteriza por un par conformadò por un "heap" y una cola de ejecución. El "heap" guarda los canales que han sido creados y cualquier proceso que esté esperando comunicarse con esos canales. La cola de ejecución guarda los procesos que son actualmente ejecutables.

1.2.4 Lenguaje Oz y su Máquina Abstracta

Oz [HM94] es un sistema de programación por restricciones concurrentes de alto orden desarrollado por el centro DFKI (Deutsches Forschungszentrum für Künstliche Intelligenz), que reune los beneficios de la programación lógica y orientada a objetos en un solo lenguaje.

Las dos principales innovaciones de Oz son la posibilidad de programar con el paradigma orientado a objetos y la posibilidad de programación de alto orden, esto es posible gracias a modificaciones en los fundamentos semánticos. La semántica de Oz está especificada por un nuevo modelo matemático, llamado el cálculo de Oz, que técnicamente está inspirado en [Mil91, MPW92]

La forma en que Oz permite la programación de alto orden es por medio de la denotación e igualdad de variables que son capturadas solamente por la lógica de primer orden. De hecho, la denotación de variables y la facilidad para programación en alto orden son conceptos totalmente ortogonales en Oz.

La comunicación concurrente es asíncrona e indeterminística. Los eventos de comunicación reemplaza dos símbolos complementarios de comunicación con una ecuación que enlaza los parámetros de la comunicación.

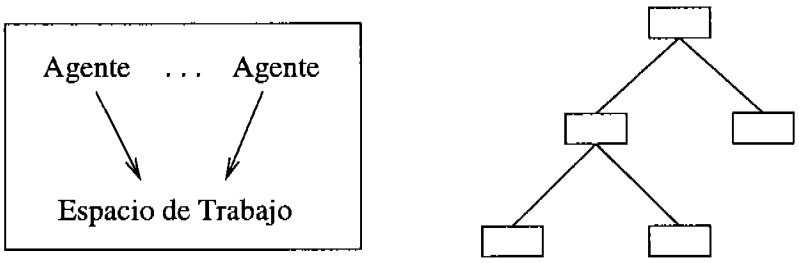

Figura 1.1: Modelo Computacional

1.2.4.1 El Cálculo Oz

La semántica operacional de Oz está definida por un modelo matemático llamado el cálculo Oz [Smo94].

La noción básica de Oz es un espacio computacional. Un espacio computacional consiste en un número de agentes conectados a un espacio de trabajo (Ver figura 1.1). Cada agente lee del espacio de trabajo y se reduce una vez el espacio de trabajo tenga la suficiente información que el agente esperaba. La información del área de trabajo crece monotónicamente. Cuando un agente se reduce puede adicionar información al espacio de trabajo y crear nuevos procesos. Cada agente por si mismo puede tener uno o varios espacios computacionales.

Los agentes de un espacio computacional son espacios a un micro-nivel. Estos se usan para programar agentes a un macro-nivel. Una forma interesante de macro-agentes son los objetos.

Formalmente un estado computacional es una expresión, (véase la figura 1.2). Restricciones, abstracciones y símbolos de comunicación residen en el área de trabajo. Las aplicaciones y condicionales son agentes. La composición y los cuantificadores son usados para unir los agentes y el área de trabajo dentro de un espacio computacional. Las abstracciones pueden ser vistas como definición de procedimientos y las aplicaciones como llamados a procedimientos.

Las clausulas en un condicional están desordenadas. Las guardas, por ejemplo, σ en $\exists\tilde{x}(\sigma$ **then** $\tau)$, constituye un espacio de computación local. Cabe anotar que

x, y, z	:	variables
σ, τ, μ	::=	
	ϕ	restricción
	$x : \tilde{y}/\sigma$	abstracción
	$x!y$	agregar símbolo
	$x?y$	obtener símbolo
	$x\tilde{y}$	aplicación
	if w_1 ... w_n **else** σ	condicional
	$\sigma \wedge \tau$	composición
	$\exists x\sigma$	cuantificación
w	::= $\exists\tilde{x}(\sigma$ **then** $\tau)$	clausula
ϕ, ψ	::= $\bot \mid \top \mid s \doteq t \mid r(\tilde{s}) \mid \phi \wedge \psi$	

Figura 1.2: Expresiones del Cálculo de Oz

cualquier expresión puede ser una guarda; se habla de guardas planas si la guarda es una restricción.

Hay dos formas de ligar variables: por medio de la cuantificación $\exists x\sigma$, que liga a x con el ambiente σ y la abstracción $x : \tilde{y}/\sigma$ une las variables en $\tilde{y}$ con el ambiente σ.

La computación está definida como una reducción (i.e., reescritura) de expresiones. Un paso de reducción se desarrolla por la aplicación de una regla de reducción a una subexpresión que satisface las condiciones de la aplicación de la regla. No se presenta backtracking.

La reducción "$\sigma \rightarrow \tau$" está definida como un módulo estructural congruente "$\sigma \equiv \tau$" de expresiones, que satisfacen la siguiente regla de congruencia:

$$\frac{\sigma \equiv \sigma' \quad \sigma' \rightarrow \tau' \quad \tau' \equiv \tau}{\sigma \rightarrow \tau}$$

La congruencia estructural es una igualdad abstracta para estados computacionales, transformando objetos puramente sintácticos a objetos semánticos. La congruencia estructural permite la asociación y composición en la comunicación, renombre de variables ligadas, movilidad de cuantificadores ($\exists x\sigma \wedge \tau \equiv \exists x(\sigma \wedge \tau)$ si x no ocurre

libre en τ), simplificación de constantes y propagación de información desde áreas de trabajo globales a áreas de trabajo locales.

Restricciones

Las restricciones (ϕ, ψ en la figura 1.2) son fórmulas de lógica de predicados de primer orden. La conjunción lógica de restricciones coincide con la composición de expresiones. Las restricciones expresan información parcial acerca de los valores de las variables. La semántica de restricciones está definida por la teoría de primer orden Δ e impuesta por la ley de congruencia.

$$\phi \equiv \psi \quad \text{si } \Delta \models \phi \leftrightarrow \psi$$

Esta ley encierra el área de trabajo con la deducción de restricciones (desde $\Delta \models \phi \rightarrow \psi$ sii $\Delta \models \phi \leftrightarrow \phi \wedge \psi$). La ley de congruencia $x \doteq y \wedge \sigma \equiv x \doteq y \wedge \sigma[y/x]$

si y es libre para x en σ impone la igualdad en el área de trabajo hacia el resto del espacio computacional ($\sigma[y/x]$ es obtenida por remplazar cada ocurrencia libre de x por y en σ). Igualdad de variables está dada estrictamente por primer orden: Dos variables x, y son iguales si de las restricciones en el área de trabajo se puede deducir $x \doteq y$, y diferente si se deduce $\neg(x \doteq y)$. En un área de trabajo inconsistente se deducen ambas $x \doteq y$ y $\neg(x \doteq y)$.

Aplicación

Un agente de aplicación $x\tilde{y}$ espera hasta que una abstracción referenciada por x aparezca en el área de trabajo para luego reducirse:

$$x\tilde{y} \wedge x : \tilde{z}/\sigma \rightarrow \exists \tilde{z}(\tilde{z} \doteq \tilde{y} \wedge \sigma) \wedge x : \tilde{z}/\sigma$$
$$\text{si } \tilde{z} \text{ y } \tilde{y} \text{ son disyuntos y de igual longitud.}$$

Hay que anotar que el área de trabajo $y : \tilde{z}/\sigma \wedge x \doteq y$ contiene una abstracción para x dada la ley de congruencia anterior.

Comunicación de restricciones

La semántica de la comunicación de dos símbolos está definida por la regla de comunicación:

$$x!y \wedge z?y \rightarrow x \doteq z.$$

La aplicación de esta regla permite que se presente el indeterminismo dentro del área de trabajo reemplazando dos símbolos de comunicación complementarios con una restricción de igualdad. La regla de comunicación es sólo para eliminar elementos del área de trabajo.

Condicionales

La semántica de un agente condicional está definida por:

$$\textbf{if } \exists\tilde{x}_1(\sigma_1 \textbf{ then } \tau_1) \ ... \ \exists\tilde{x}_n(\sigma_n \textbf{ then } \tau_n) \textbf{ else } \mu.$$

La guarda σ_i de las clausulas, determinan un espacio computacional local que se reduce concurrentemente. Es esencial para los espacios computacionales locales que la información que se encuentra en el área de trabajo global sea visible para las áreas de trabajo locales.

Esto es logrado por medio de la ley de propagación (hay que tener en cuenta que las clausulas están desordenadas):

$$\pi \wedge \textbf{if } \exists\tilde{x}(\sigma \textbf{ then } \tau)\tilde{w} \textbf{ else } \mu$$
$$\equiv$$
$$\pi \wedge \textbf{if } \exists\tilde{x}(\pi \wedge \sigma \textbf{ then } \tau)\tilde{w} \textbf{ else } \mu$$

Siendo π una restricción o una abstracción y sin que haya variables en $\tilde{x}$ que aparezcan libres en π

Leyendo de izquierda a derecha, la ley crea una copia de la información desde el área de trabajo global, al área de trabajo local. Leyendo de derecha a izquierda, la ley elimina información que está presente globalmente. Un ejemplo del uso de la ley de propagación en ambas direcciones (como también de simplificación de restricciones) es:

$$x \doteq 1 \wedge \textbf{if } (x \doteq 1 \textbf{ then } \sigma) \ (x \doteq 2 \textbf{ then } \tau) \textbf{ else } \mu$$
$$\equiv x \doteq 1 \wedge \textbf{if } (\top \textbf{ then } \sigma) \ (\bot \textbf{ then } \tau) \textbf{ else } \mu$$

En el ejemplo se asume que de la teoría de restricciones se deduce que 1 y 2 son diferentes.

Operacionalmente, la simplificación de restricciones y la ley de propagación puede ser realizada con un co-llamado relativo al procedimiento de simplificación.

Hay dos formas distinguidas en que las guardas de las clausulas pueden reducirse, satisfacible y falla. Si la guarda de una clausula se satisface, la condición se puede reducir:

$$\textbf{if } \exists\tilde{x}(\sigma \textbf{ then } \tau)\ \tilde{w} \textbf{ else } \mu \ \rightarrow\ \exists\tilde{x}(\sigma \wedge \tau) \quad \text{si } \exists\tilde{x}\sigma \equiv \top.$$

La reducción adiciona tanto σ como τ en el área de trabajo global y libera el cuerpo de la clausula.

Una clausula falla si la restricción en el área de trabajo es insatisfecha. Si la guarda de una clausula falla, la clausula es simplemente descartada.

$$\textbf{if else } \mu \ \rightarrow\ \mu.$$

1.2.4.2 Máquina Abstracta para Oz

La máquina abstracta que implementa a Oz es llamada Amoz [MSS95]. Amoz implementa árboles de relaciones para las restricciones, procedimientos de primera clase, guardas profundas e hilos, dejando de lado los estados mutables de los objetos, registro de restricciones, como también dominios finitos de restricciones y búsquedas encapsuladas.

La noción de espacio computacional es vista como un número de tareas conectadas a un área de almacenamiento (store).

La computación procede por la reducción de tareas respecto a la información contenida en el store. Una tarea se reduce tan pronto exista suficiente información en el "store". Cuando una tarea es reducida nueva información puede ser adicionada al store o nuevas tareas pueden ser creadas. Las tareas son de vida corta: dejan

de existir una vez se reducen. Algunas tareas pueden expandir espacios locales de computación. Espacios locales pueden ser creados y espacios existentes pueden ser eliminados o unidos con los espacios padres.

El Store se divide en un store de restricciones y un store de procedimientos. El store de restricciones acepta restricciones del estilo $x = y \mid x = f(\overline{y})$ en forma normal. El store es siempre monotónicamente creciente. Las restricciones son interpretadas como una estructura fija de primer orden, llamado "Universo". El universo contiene un árbol relacional. El store de procedimientos contiene las ligaduras (binding) de nombres a procedimientos.

El árbol de espacios computacionales locales conoce las restricciones de los espacios globales. Una restricción se impone sobre un espacio local y sobre todos los store siguientes en el árbol de espacios computacionales. Si el espacio computacional falla, todos los espacios debajo fallan y todos las tareas involucradas son descargadas.

Las tareas se pueden dividir en dos clases, elaboradoras (reducciones) y tareas de condicional. Una reducción es la tarea que ejecuta una expresión. Las expresiones se muestran en la figura 1.3.

E, F, G	::=	$x = y \mid x = f(\overline{y})$	Restricciones
	|	`local` x `in` E `end`	Declaraciones
	|	EF	Composición
	|	`proc` $x\overline{y}$ E `end`	Definición de procedimientos
	|	$x\overline{y}$	Aplicación de procedimientos
	|	`if` $\overline{x}$ `in` E `then` F `else` G `fi`	Condicional

Figura 1.3: Expresiones de AMOz

Reducción de una restricción: Impone una restricción $x = y \mid x = f(\overline{y})$ en el store.

Reducción de una declaración: `local` x `in` E`end`. Crea una nueva variable local en el espacio computacional y una reducción para E. Dentro de E se referencia la nueva variable (x). El espacio es llamado "home" de x. La Declaración `local` $x\overline{y}$ `in` E`end` es la abreviación de `local` x `in local` $\overline{y}$ `in` E`end end`

Reducción de composición: EF Es la reducción por separado para E y F.

Reducción de la definición de procedimientos: `proc` $x\overline{y}$ E `end`. Escoge un nuevo nombre a, liga $a : \overline{y}/E$ y lo escribe en el store de procedimientos y crea un elaborador para la restricción $x = a$. Un nombre es una constante en el Universo. Hay un infinito número de nombres en el Universo. Un nombre sólo referencia a un solo procedimiento.

Reducción de una aplicación de procedimiento: $x\overline{y}$ Espera hasta que exista un nombre a, que restrinja $x = a$ y el store de procedimiento contenga una ligadura $a : z_1...z_n/E$. Cuando este es el caso, una reducción para la expresión $E/[y_1/z_1, ..., y_n/z_n]$ es creado, donde los parámetros formales son remplazados por los parámetros actuales.

Reducción de un condicional: `if` $\overline{x}$ `in` E `then` F `else` G `fi`. Se realiza en dos pasos. Primero se considera el caso especial `if` $\overline{y}$ `in` $\overline{x} = f(\overline{y})$ `then` F `else` G `fi`, donde las variables $\overline{y}$ son excluyentes. En este caso se crea una tarea de condicional. La tarea espera hasta que del store se pueda deducir $\exists\overline{y}x = f(\overline{y})$, en este caso se hace una reducción de `local` $\overline{y}$ `in` $x = f(\overline{y})F$ `end`. En el caso de que $\neg\exists\overline{y}x = f(\overline{y})$, se reduce a G.

En general el condicional `if` $\overline{x}$ `in` E `then` F `else` G `fi` se asume como el caso simplificado anterior. La reducción crea una tarea que expande un espacio de computación local. A la expresión $\overline{x}$ `in` E se le llama guarda. Una guarda es profunda si E no es una restricción $(x = y \mid x = f(\overline{y}))$.

Los hilos de ejecución son una secuencia de tareas no vacía. Cada tarea puede pertenecer únicamente a un solo hilo de ejecución. Cuando el espacio de computación falla, todas las tareas asociadas son descargadas, removiéndolas del hilo de ejecución que las contiene.

Un hilo de ejecución corre si su primera tarea es reducible, en caso contrario se mueve la primera tarea a un nuevo hilo de ejecución. La reducción de una tarea en un hilo de ejecución significa reducir la tarea y reemplazarla con posibles secuencias vacías de tareas.

Si el hilo de ejecución contiene una tarea no reducible, queda en estado *s*uspendido, en caso contrario está en estado *l*isto (Figura 1.4). Después de la creación de una

tarea, esta se encuentra suspendida, luego pasa al estado listo para intentar reducirla.

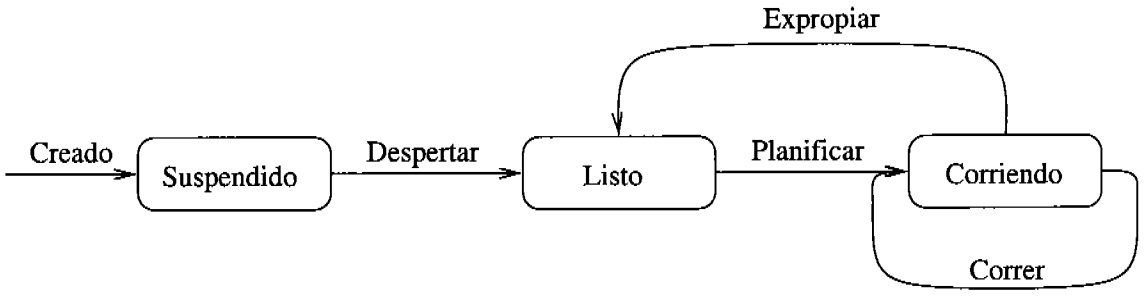

Figura 1.4: Estados de los Hilos de Ejecución de AMOz

AMOz es secuencial, está basado en un solo trabajador, pero con múltiples hilos de ejecución con panificación por expropiación. Sólo un hilo puede ejecutarse a la vez. La planificación no se basa en tiempos.

La elaboración de una expresión E corresponde a la ejecución de sus correspondientes instrucciones $\mathcal{C}[\![E]\!]$ dados por el compilador. Se consideran en adelante sólo expresiones cerradas (Sin variables libres) y renombradas (Variables declaradas sólo una vez).

Para compilar una expresión E, el conjunto V de variables declaradas en E es computado en el procedimiento que definió la variable. Para cada variable $x \in V$ se crea un índice $\mathcal{A}[\![x]\!]$, así Amoz puede referenciar a la variable x por $\mathcal{A}[\![x]\!]$.

Amoz necesita varios registros. El contador de programa `PC` apunta a la instrucción que está siendo ejecutada. El ambiente `E` es un arreglo donde se proyectan los índices $\mathcal{A}[\![x]\!]$ de la variable x a los nodos de la variable donde es almacenada: `E`$[\mathcal{A}[\![x]\!]]$. El emulador de ciclo mostrado en la figura 1.5 contiene una sola instrucción `ALLOCATE(`n`)`.

La ejecución de `local` x `in` E `end` crea un nodo de variable "nueva" en el almacenamiento y una referencia a este, en el ambiente `E`$[\mathcal{A}[\![x]\!]]$ (ver figura 1.6).

Una composición se compila como una concatenación de las respectivas secuencia de instrucciones: $\mathcal{C}[\![E\ F]\!] \equiv \mathcal{C}[\![E]\!]\mathcal{C}[\![F]\!]$.

La restricción de igualdad es traducida como un llamado al algoritmo de unificación

```
#define DISPATCH { PC++; goto emulate; }
engine() {
  emulate: switch (*PC) {
             case ALLOCATE(n):
                  E = new Node*[n];
                  DISPATCH;
                  /* Las proximas instrucciones se colocan aqui */
           }
  fail: /* Maneja los problemas en la unificacion */
}
```

Figura 1.5: Ciclo de Emulación en Amoz

$\mathcal{C}[\![$ `local` x `in` E `end` $]\!] \equiv$	
CREATE_VAR($\mathcal{A}[\![x]\!]$)	`case CREATE_VAR(i):`
$\mathcal{C}[\![E]\!]$	`  E [i] = new Node (tag:  VAR);`
	`  DISPATCH;`

Figura 1.6: Traducción de la Instrucción Create en Amoz

(Ver la figura 1.7).

$\mathcal{C}[\![x = y]\!] \equiv$	
UNIFY(A[x],A[y])	`case UNIFY(i,j):`
	`  if (unify(E[i],E[j])==False) goto Fail;`
	`  DISPATCH;`

Figura 1.7: Traducción de la Restricción de Igualdad en Amoz

La construcción de la tupla $x = f(y)$ se realiza en 3 pasos. Primero se reserva memoria para el nodo donde se guarda la función y en el registro `S` se guarda una referencia a dicho nodo. Segundo, los argumentos son construidos y referenciados en la tupla. El ultimo paso es unificar la tupla con x. (ver la figura 1.8)

Hilos (Threads) y Emparejamiento (Matching)

Se presentan los hilos de control en forma de expresiones en condicionales que implementan pattern-matching (apareamiento de patrones). Los condicionales son considerados de la forma `if` y `in` $x = f(x)$ `then` E `else` F `fi`, donde las variables y son una a una distintas. Estos pueden ser reducidos, si una y sólo una de las

$\mathcal{C}[\![x = f(y_1 \ldots y_n)]\!] \equiv$

CREATE_TUPLE(f/n)	`case CREATE_TUPLE(f/n):`
PUT_ARG(1,$A[y_1]$)	`  S = new Node (tag:TUPLE,`
...	`  tuple:(label:f/a,args:new Nodes[n]));`
PUT_ARG($n, A[y_1]$)	`  DISPATCH;`
UNIFY_S(A[x])	`case PUT_ARGS(i,j):`
	`  S->tuple.args[i] = E[j];`
	`  DISPATCH;`
	`case UNIFY_S(i):`
	`  if (unify(S,E[i])==False) goto Fail;`
	`DISPATCH;`

Figura 1.8: Traducción de una Tupla en Amoz

variables x está instanciada (bounded). En el caso de que x no esté instanciado se introduce un hilo de ejecución en Amoz.

Los hilos en Amoz son pilas de tareas. Tiene el tipo `Thread` y tiene las funciones comunes de pilas, `push`, `pop` y `isEmpty`. Una *t*area `TASK(l)` apunta a una instrucción de la máquina abstracta etiquetada por `l`. Amoz contiene tres registros: `running` (de tipo `Thread`) para tener referencia al hilo que se ejecuta actualmente, `runnable` (de tipo `ThreadQueue`) para almacenar la cola de hilos por ejecutar, y `timeOver` la cual es una bandera que marca el fin de un hilo después de cierto tiempo.

Adicionar una tarea al hilo que actualmente se está ejecutando se realiza por un `PUSH`. La instrucción `RETURN` trata de ejecutar la tarea más arriba del hilo actual en ejecución. Si no hay tareas por ejecutar, se cambia al siguiente hilo. (Preemption) es revisado solamente en la instrucción `PUSH`.

El ciclo emulado es extendido con las nuevas referencias como se muestra en la figura 1.9. Hay que tener en cuenta que sólo se trabajan con los hilos `runnable`.

1.2.5 Máquina Abstracta de TyCO

En los últimos años, se han echo muchos esfuerzos para proveer una semántica de objetos con variaciones y extensiones del cálculos π visto en la sección 1.2.3; TyCO [Vas94a] (Typed Concurrent Objects) es uno de esos esfuerzos en el que el cálculo

```
#define DISPATCH { PC++; goto emulate; }
engine() {
  emulate: switch (*PC) {
             case ALLOCATE(n):
                  E = new Node*[n];
                  DISPATCH;
                  /* Las proximas instrucciones se colocan aqui */
           }
  fail: /* Maneja los problemas en la unificacion */
  run:
      ...
  schedule:
      ...
  preempt:
      ...
}
```

Figura 1.9: Ciclo de Emulación en Amoz con Hilos

plantea formalmente la interacción de objetos en una comunicación asíncrona utilizando mensajes, los cuales son las entidades fundamentales junto con los objetos. Las plantillas (templates) que son especificaciones de procesos abstractos en una secuencia de variables, permiten modelar clases para instanciar objetos en tiempo de ejecución.

TyCO tiene el mismo poder computacional del cálculo π y retoma el concepto de Abadi y Cardelli en el cálculo ζ en el que los objetos son conjuntos de métodos etiquetados donde "self" es interpretado como el canal donde el objeto está localizado, y los mensajes son invocaciones sincrónicas de métodos (ver la tabla 1.4).

Los nombres son denotados por a, b,..., y también v, x, y,.... Secuencias de nombres de la forma $x_1 \ldots x_n$, para $n > 0$ son abreviados con $\tilde{x}$. Los agentes variables son denotados por $X, Y, \ldots$, y las etiquetas por $l, l_1, l_2, \ldots$.

El sistema de reglas de reescritura de TyCO es realmente pequeño, sólo hay dos reglas: la primera hace reducción estructural y la segunda hace reducción cuando un objeto recibe un mensaje que va dirigido a él.

Procesos: P, Q	::=	$a \lhd l : \tilde{v}$	Mensaje
	|	$a \rhd [l_1 : A_1 \& \ldots \& L_n : A_n]$	Objeto
	|	P, Q	Composición concurrente
	|	$\|X\|P$	Variable local
	|	$X(\tilde{v})$	Reescritura de agente
	|	$A(\tilde{v})$	Reescritura de agente
	|	**let** $X = A$ **in** P	Instanciación local
Agentes: A	::=	$(\tilde{x})P$	Procesos simples
	|	**rec** $X.A$	Procesos recurrentes

Tabla 1.4: Sintaxis del Cálculo TyCO

$$\text{STRUCT } \frac{P' \equiv P \quad P \longrightarrow Q \quad Q \equiv Q'}{P' \longrightarrow Q'}$$

$$\text{COMM: } v\tilde{x}(a \rhd M, a \lhd C, \tilde{Q}) \longrightarrow v\tilde{x}(M \bullet C, \tilde{Q})$$

donde C representa la etiqueta de un método con sus argumentos actuales y M representa la colección de métodos del objeto a. La regla COMM expresa el hecho que dos procesos, uno objeto y otro mensaje sobre este objeto, reducen a la aplicación del método solicitado por el mensaje.

La máquina abstracta de TyCO está compuesta de dos áreas de memoria. La primera es donde se almacena la parte estática: Instrucciones de programas, Canales locales, tablas de métodos y arreglos estáticos no inicializados. La segunda área es un "heap" para estructuras dinámicas, incluyendo la cola de ejecución y las colas de comunicación. Véase la figura 1.10. La máquina abstracta opera con dos tipos básicos internos: palabra (word) que es la mínima área de memoria direccionable y marco (frame) que es usado para apuntar a un bloque de palabras de máquina.

1.2.5.1 Datos estáticos

Hilos: Secuencia de instrucciones. Un programa es una colección de hilos. La ejecución comienza con el hilo especial T0. En una reducción pueden generarse más hilos para ser puestos en la cola de ejecución; cuando el hilo actual finaliza, se ejecuta el siguiente en la cola. Los hilos son ejecutados dentro de un ambiente que enlaza

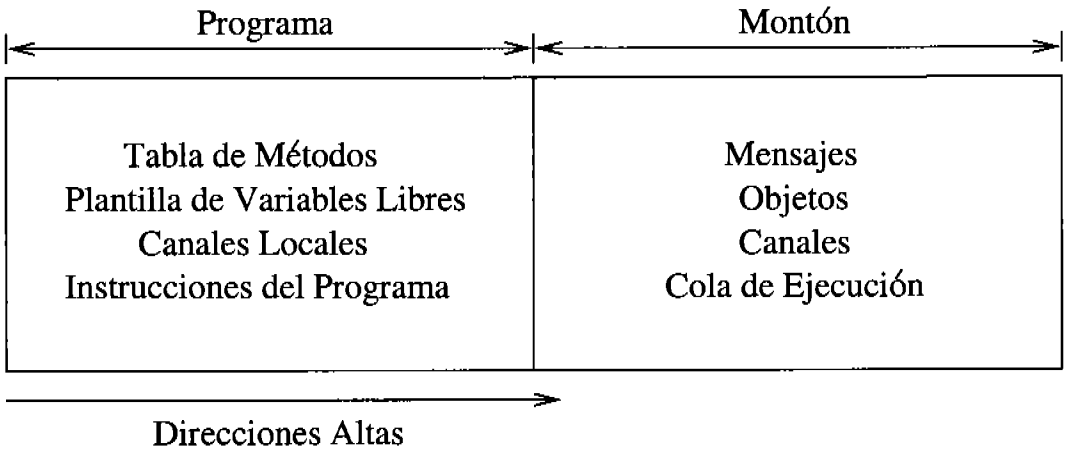

Figura 1.10: Capa de Memoria de la Máquina de TyCO

las variables con una posición de memoria.

Canales locales: Típicamente un hilo en ejecución crea un nuevo canal. Estos canales unen variables locales al hilo o a un subconjunto de instrucciones de este.

Tablas de métodos: es un marco con al menos una palabra (objetos sin métodos, no tiene tablas de métodos) Cada palabra contiene la dirección de la primera instrucción de un método dentro del objeto. Las palabras dentro de la tabla puede ser visto como las etiquetas de los métodos.

Plantillas de variables libres: Las plantillas son objetos parametrizados en una secuencia de variables distintas. Las variables libres de una plantilla son sus valores cuando la plantilla está definida en el programa. Los enlaces (binding)[ASU90] de las variables libres de una plantilla, son guardados en marcos estáticos. El tamaño de estos marcos son conocidos en tiempo de compilación.

1.2.5.2 Datos Dinámicos

Heap: El heap es un espacio direccionable de memoria donde la máquina abstracta pone las estructuras dinámicamente. El heap provee las siguientes instrucciones:

New : Nat $\mapsto$ Frame

Free : Frame $\mapsto$ Void

New recibe un número natural para especificar el tamaño del bloque requerido y retorna un puntero a un marco de ese tamaño. *Free* recibe un puntero a un marco y libera la memoria para que sea reusado o reciclado.

Colas: Son usadas para implementar la cola de ejecución y los canales de comunicación. Las colas proveen las siguientes operaciones

Empty : Queue $\mapsto$ Bool

NewQueue: Void $\mapsto$ Queue

Enqueue: Queue x Frame $\mapsto$ Void

Dequeue: Queue $\mapsto$ Frame

Empty dice si la cola está vacía. *NewQueue* inicializa una cola vacía. *Enqueue* recibe una cola y un marco y pone el marco en (el final de) la cola. *Dequeue* recibe una cola y retorna un marco (el del tope) de la cola sacando el marco de la cola.

Cola de ejecución: Los items en al cola de ejecución son marcos de tamaño tres que representan un hilo ejecutable. Las palabras de cada marco tienen los siguientes nombres: código, variables libres y argumentos. Cuando una reducción ocurre, un nuevo marco es puesto en la cola de ejecución. La máquina usa el entero "etiqueta" en el marco mensajes y el puntero a la tabla de métodos en el marco objeto para buscar cuál hilo correr. La dirección de este hilo es puesto en la palabra "código" del nuevo marco. La palabra "variables libres" (argumentos) tiene el puntero al objeto (mensaje), marco donde los enlaces de las variables libres (parámetros) están puestos.

Marco objetos: Los marcos objetos tienen al menos 2 palabras. La primera es llamada continuación y es usada en manipulación de colas; la segunda es llamada tabla porque es un puntero a una tabla de métodos (o nulo si el objeto no tiene métodos). Palabras adicionales pueden existir como el enlace de las variables libres.

Marco mensajes: Los marcos mensajes también tienen al menos 2 palabras. La primera es igual al primer campo de los marcos objeto, y la segunda es llamada

"etiqueta" y es un número natural, índice de algunas tablas de métodos. Puede tener palabras adicionales, los cuales son los argumentos del mensaje.

Canales: Los canales implementan nombres para intercambiar mensajes entre procesos. Un canal es una cola de objetos o marcos mensaje extendida con la siguiente función.

State : Queue $\mapsto$ vacío, mensajes, objetos

Si la cola está vacía, *State* retorna *vacío*, si ella contiene al menos un marco mensaje, retorna *mensaje* o *objeto* si la cola contiene al menos un marco objeto. Un canal no puede tener mezclas de objetos o mensajes.

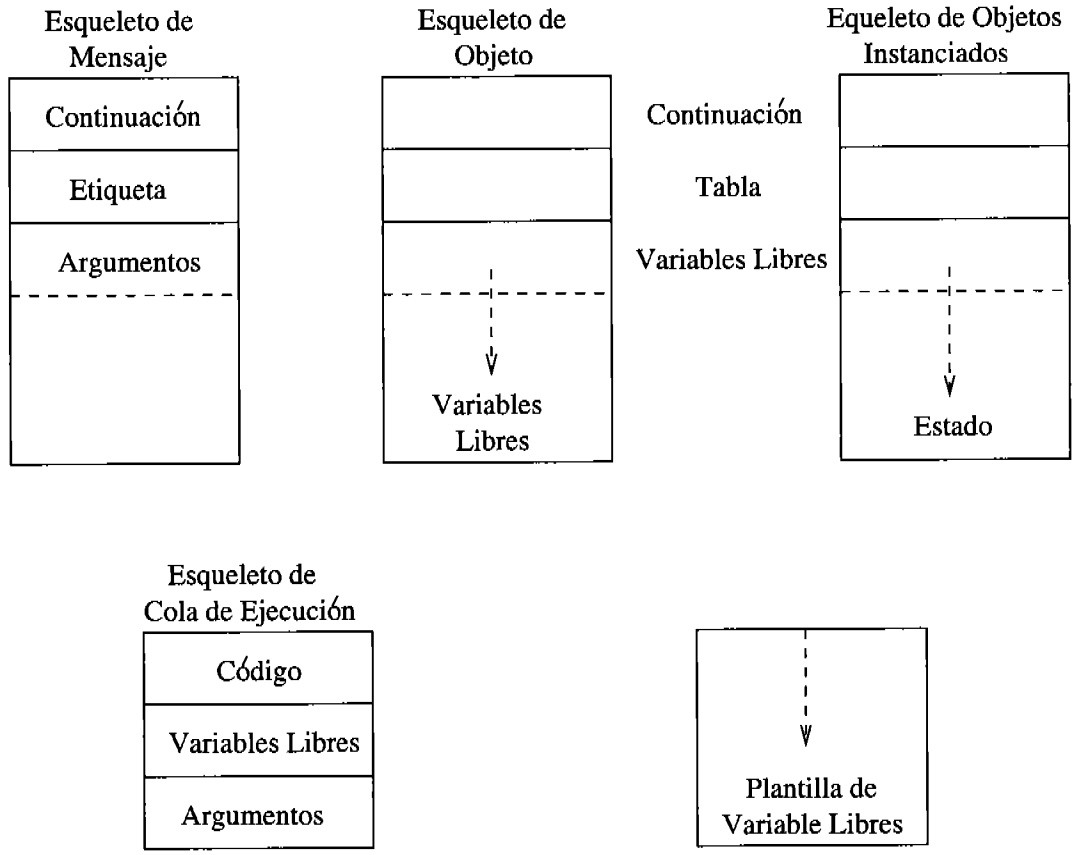

Figura 1.11: Marcos de Mensajes, Objetos y Colas de Ejecución de la Máquina de TyCO

1.2.5.3 Registros

La máquina abstracta usa un pequeño conjunto de registros para controlar el flujo de control de un programa, para guardar punteros y canales locales, así como también manejar la cola de ejecución.

PC(program counter): Puntero a la instrucción en ejecución. Este registro es inicializado con la dirección del hilo T0, que es el primero en ser invocado por todos los programas.

CC(current channel): Apunta al canal el cual está siendo usado para hacer la reducción.

CF(current frame): Guarda marcos (objetos o mensajes) temporalmente hasta que ellos son encolados o usados en una reducción.

OF(other frame): Usado para operaciones temporales internas.

NT(new thread): Marcos para hilos ejecutables son puestos temporalmente en este registro, durante la reducción del frame, cuando un nuevo hilo está siendo construido y también cuando un nuevo hilo es sacado de la cola de ejecución para ser ejecutado.

FB(free variable bindings) y AB(argument bindings): Son usados para guardar el enlace (binding) de las variables libres y de los parámetros respectivamente.

LC(local channel): Es un puntero a la primera palabra disponible en el marco direccionado estáticamente para canales locales (NC).

1.2.6 Máquina Virtual de Java

La Máquina Virtual de Java fue diseñada para soportar el lenguaje de programación Java, sin embargo, la máquina no conoce nada de este lenguaje.

Como una máquina computacional normal, la máquina virtual de java tiene varias áreas de memoria, un manejo de datos y un conjunto de instrucciones.

1.2.6.1 Tipos de Datos

La Máquina Virtual de Java opera con dos clases de tipos de datos: tipos primitivos y referencias a tipos.

Esta máquina virtual no hace chequeo de tipos, ya que espera que el compilador del lenguaje de alto nivel que la use haga el chequeo de tipos en tiempo de compilación; pero los operandos del conjunto de instrucciones si deben tener un tipo determinado, por ejemplo: `iadd` es la adición para valores de tipo entero.

La JVM[1], contiene soporte explicito para objetos. Un objeto es un arreglo o una instancia de clase direccionada dinámicamente. Referencias a objetos se consideran referencias a tipos.

Los tipos primitivos son los siguientes:

- byte: 8 bit con signo complemento a 2.
- short: 16 bit con signo complemento a 2.
- int: 32 bit con signo complemento a 2.
- long: 64 bit con signo complemento a 2.
- char: 16 bit sin signo usando codificación Unicode versión 1.1.5.
- float: 32 bit números de punto flotante IEEE 754.
- double: 64 bit números de punto flotante IEEE 754.

Respecto a las referencias a tipos, hay tres géneros:

- Tipos de clase.
- Tipos de interfaz.
- Tipos de arreglo.

[1]Máquina Virtual de Java por sus siglas en ingles.

Sus valores son referencias de clase, arreglos (que implementan o no interfaces) creados dinámicamente.

Un valor de referencia puede ser una referencia especial "null". La referencia null inicialmente no tiene tipo pero se le puede hacer "casting[2]" a cualquier tipo.

En la JVM, se define una noción abstracta de una palabra. El tamaño de palabra es del tamaño de un puntero en la plataforma en que esté implementada. Así pues en una plataforma de 32 bits, una palabra es de 32 bits y los punteros son de 32 bits.

A diferencia de una implementación de 32 bits que puede guardar un valor de tipo long o double en dos palabras, en una implementación de 64 bits, sólo se necesita una palabra para estos valores.

1.2.6.2 Areas de Datos En Tiempo de Ejecución

Un registro PC (program counter): La JVM tiene un PC por cada hilo de ejecución. El PC contiene la dirección de la instrucción que será ejecutada dentro del hilo. El ancho del PC es una palabra.

Una pila privada (Java stack): Se crea cuando inicia el hilo. En esta pila se guardan los marcos (sección 1.2.6.3). Aquí, se guardan variables locales, resultados parciales y hace parte del la invocación y el retorno de un método. El stack sólo puede ser manipulado con "pop" y "push" de marcos. La especificación de la JVM permite que la pila sea de tamaño fijo o que varíe dinámicamente.

Una pila compartida (heap): Es una pila compartida entre todos los hilos de ejecución. El heap es el área de memoria donde todas las instancias y arreglos son direccionados. El heap es creado cuando la máquina virtual arranca. El almacenamiento de los objetos es liberado por un manejador automático ("garbage collector"). En la especificación no hay un tipo especial de garbage collector, esto depende de los requerimientos.

Area de métodos: El área de métodos es compartida entre todos los hilos de ejecución. Contiene, por clase, estructuras como el conjunto de constantes (men-

[2]Usualmente llamado en programación a la conversión de tipos

cionada más adelante), campos y datos de los métodos y el código de los métodos y constructores, incluyendo los métodos especiales usados en la inicialización de clases, instancias e interfaces. Al igual que el heap, esta área es creada cuando la máquina arranca. En ella no es necesario un "garbage collector", aunque se deja a decisión de la implementación su posible inclusión. La máquina permite que también esta área sea de tamaño fijo o que varíe dinámicamente.

Conjunto de constantes: Hay un conjunto (pool) de constantes por cada clase o interfase. Este contiene muchas clases de constantes, rango de literales numéricos de métodos y referencias conocidos en tiempo de compilación. Cada conjunto de constantes es localizado en el área de métodos.

Pila de métodos nativos: Una implementación de la JVM puede usar pilas convencionales para apoyar métodos nativos y métodos escritos en otros lenguajes. Implementaciones que no provean código nativo, no necesitan de esta pila. Si proveen métodos nativos, las pilas son comúnmente direccionadas cada una por un hilo base. El tamaño puede ser fijo o variar dinámicamente.

1.2.6.3 Marcos o Frames

Un marco (o frame) se usa para guardar datos y resultados parciales así como también para desarrollar enlaces dinámicos, retornar valores de los métodos y disparar excepciones.

Cada que se invoca un método, se crea un nuevo marco. Este se destruye cuando el método se termina. Los marcos están guardados en la pila privada (java stack) del hilo de ejecución. Cada marco tiene su propio conjunto de variables locales y su propia pila de operandos. El espacio de memoria de cada marco puede ser separado en el momento de la creación pues el tamaño de las variables locales y de la pila son conocidas en tiempo de compilación.

Sólo el marco del método que se está ejecutando está activo al tiempo en un hilo. Este marco es llamado el *current frame*, y el método es conocido como el *current method*; la clase en la que está definido este método se le conoce como *current class*. Un método deja de ser *current method* si este invoca otro método o termina su

ejecución (normal o anormalmente).

Un método termina normalmente si la invocación no causa una excepción de tal manera que puede retornar un valor.

Note que un marco creado por un hilo es local a él y no puede ser directamente referenciado por algún otro hilo.

Variables locales: En cada invocación de un método, la JVM direcciona un marco que contiene un arreglo de palabras conocido como variables locales direccionables con un desplazamiento y la base del arreglo.

Pila de operandos: La mayoría de las instrucciones en la JVM toma valores de la pila de operandos del *current frame*, opera en él y retorna resultados en la misma pila, también usado para pasar argumentos a los métodos y recibir resultados.

Enlace dinámico: Un marco contiene una referencia al "pool" de constantes para el tipo del *current method* de modo que facilite el enlace dinámico del código del método.

Una de las propuestas examinadas durante el desarrollo de este trabajo fue extender la Máquina Virtual de Java y usarla para PiCO. Pero al extender la JVM se encuentran algunos problemas (aparte del reducido "bytecode" disponibles para futuras extensiones) como son: perdida de portabilidad al tener que compilar la máquina virtual para cada una de las plataformas y existe la posibilidad de que las extensiones a la máquina no sean ortogonales con respecto a las instrucciones originales [Shi96].

1.2.7 Máquina "Spineless Tagless G"

La máquina G es una máquina abstracta diseñada para soportar lenguajes funcionales de alto nivel y con evaluación perezosa, es decir que los valores son evaluados sólo cuando se necesitan.

1.2.7.1 La representación de clausuras

La máquina tiene una pila (heap) la cual contiene dos tipos de objetos: valores o *head normal forms* y no evaluados o *thunks*. Adicionalmente, los valores pueden ser clasificados en dos tipos: valores de función y valores de datos. Un valor puede contener *thunks* por dentro.

funciones: Cualquier implementación de un lenguaje de alto nivel deben proveer la vía para representar un valor de una función. Estas, se comportan como una computación suspendida. Cuando el valor es aplicado a sus argumentos, la computación se ejecuta.

La forma más compacta de representar un valor de una función es un bloque de código estático, junto con los valores de las variables libres, comúnmente llamado clausura.

La representación física más directa para esto, es un puntero a un bloque continuo de memoria localizada en la pila, con un apuntador a código seguido de los valores de las variables libres. Véase la figura 1.12

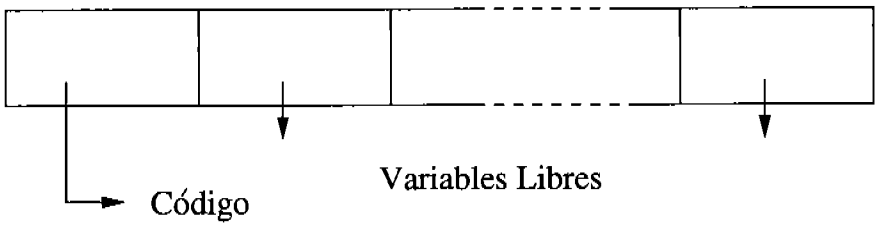

Figura 1.12: Valor de una Función en la Máquina STG

Para desarrollar la computación, un registro llamado "*enviroment pointer*" apunta a la clausura (figura 1.12) y el código es ejecutado. El código puede acceder a las variables libres por desplazamiento[3] con el registro y a los argumentos por algún paso de parámetros estándar. (registro de activación, pila, etc.).

"thunks": En un lenguaje con evaluación perezosa. Los valores son pasados a funciones o guardados en estructuras de datos en una forma no evaluada, y sólo son

[3]desplazamiento (offset): para acceder direcciones de memoria a partir de una dirección base

evaluados cuando los valores son realmente requeridos. Como las funciones, estas formas no evaluadas capturan una computación suspendida y pueden ser representadas de la misma manera que las funciones.

Un "thunk", puede ser (en principio) implementado como una simple función, pero esto es ineficiente, ya que se necesita evaluar repetidamente. Este trabajo repetido es eliminado con una implementación llamada *lazy* que es algo así como: cuando un thunk es forzado la primera vez, esta es físicamente actualizada con sus valores.

Hay tres estrategias principales para hacer la actualización en implementaciones *lazy* [Pey93]: el modelo de reducción inocente, el modelo celda y el modelo autoactualizable que es usado por la máquina STG[4] (de donde saca su nombre "tagless"). Estas estrategias no son descritas en este trabajo pues está fuera del alcance del mismo.

[4]Spineless Tagless G-machine

2 CORDIAL: UN LENGUAJE DE PROGRAMACION VISUAL

En este Capítulo se dará una breve explicación del lenguaje *CORDIAL* ("Constraint-ed Objects Relations Defining an Iconic Applicative Language") [QRT97c], que se fundamenta en el cálculo PiCO (capítulo 3) y la implementación de la máquina abstracta como núcleo; hay que tener en cuenta que el diseño de la máquina tiene que ver con el cálculo y no con Cordial directamente.

2.1 QUE ES CORDIAL

Cordial es un lenguaje visual de propósito general distinguido por tres características: un modelo visual jerárquico que permite programación basada en grafos e iconos, un formalismo visual de base para la interpretación de programas visuales y un modelo semántico basado en un cálculo formal que integra los paradigmas de programación por restricciones y programación orientada a objetos.

Sintácticamente, un programa en *Cordial* es una disposición de formas en la pantalla tales como iconos, segmentos de líneas y flechas que se usan para representar diferentes elementos de un programa y su ejecución. Iconos o formas definidos por el usuario son usados para representar objetos, mientras que las flechas y líneas que los tocan definen relaciones. *Cordial* permite diferentes niveles de abstracción en las representaciones visuales para un programa. Cada forma en el programa tiene una interpretación en un nivel inferior. El último nivel de abstracción es una representación en el formalismo visual de base. La noción de un programa en *Cordial*

incluye todas las formas junto con las funciones que asocian cada forma con sus vistas expandidas. Para más información, véase [RAQ+98].

2.2 SINTAXIS DE CORDIAL

Cordial un lenguaje visual orientado a objetos, con pocas construcciones de flujo de control, intrínsecamente concurrente y con un modelo de ejecución subyacente enmarcado en el paradigma de restricciones, soportado todo por un modelo formal. En este trabajo sólo se introducirán los elementos y construcciones básicas junto con su representación.

2.2.1 Elementos Visuales Básicos

El objetivo de *Cordial* es representar visualmente los conceptos básicos de programación orientada a objetos tales como clase, objeto, encapsulación, método, herencia y polimorfismo; combinándolos con algunos elementos de otros estilos de programación, tales como restricciones y paso de funciones como argumentos. Un programa es un conjunto de clases (atributos, métodos y restricciones iniciales) en el que todo "flujo de control" debe especificarse dentro de un método. Por lo tanto, la ejecución del "programa" empezará con la invocación de uno de los métodos sobre una clase distinguida.

2.2.2 Clases

Las clases tienen dos estados visuales, minimizado y maximizado. Se representan minimizadamente mediante un rectángulo con un contorno de linea sencilla, en el interior se encuentra su nombre y su icono asociado. Para especificar los atributos y métodos de la clase, se usa su representación maximizada, es decir, dos rectángulos: en el superior está el nombre y el icono asociado, en el interior del rectángulo inferior contiene, en regiones bien delimitadas, tanto atributos como métodos y restricciones iniciales. Los atributos se especifican mediante su nombre y clase; los métodos mediante su nombre y signatura. Véase la figura 2.1.

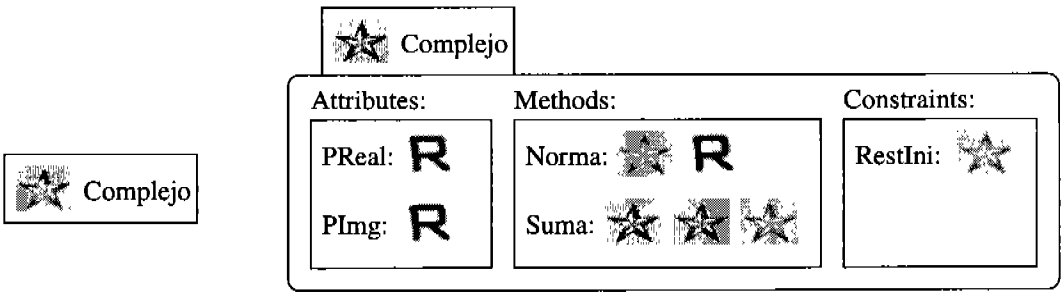

Figura 2.1: Representación Minimizada y Maximizada de una Clase en Cordial

Los métodos se interpretan como relaciones, por lo que la signatura de los mismos está dada por el producto cartesiano de las clases de los argumentos de la relación y la primera clase corresponde a la clase donde se define el método.

Las clases deben definirse enteramente de manera estática, de manera que en ejecución no se podrá ni añadir o eliminar nuevos atributos ni métodos a clases existentes, ni crearse nuevas clases.

2.2.3 Herencia

El lenguaje sólo permite herencia simple y se considera que existe una super(meta) clase *object*, de la que deben heredar todas las demás clases, incluyendo, de manera particular, a las (meta)clases Método y Clase.

Véase la representación visual de herencia entre clases en la figura 2.2.

2.2.4 Objetos o Instancias de Clase

La representación de los objetos es similar a la representación de las clases: se realiza a través de un rectángulo y tiene una representación minimizada y una maximizada. La representación maximizada se diferencia de las clases en que sólo aparecen los atributos del objeto minimizados. Véase la figura 2.3

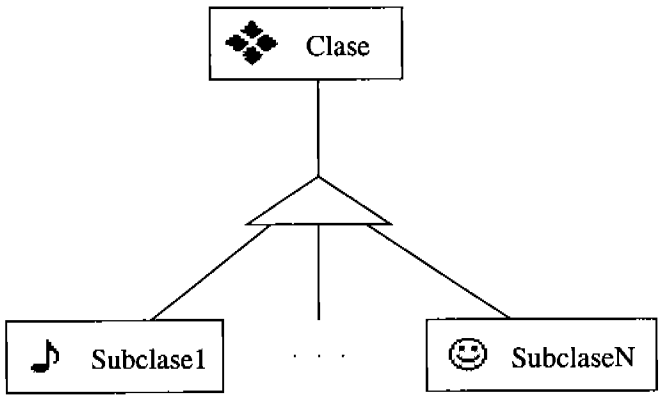

Figura 2.2: Representación de la Herencia entre Clases en Cordial

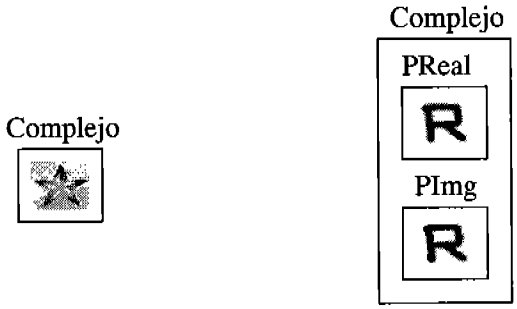

Figura 2.3: Representación Minimizada y Maximizada de una Instancia de la Clase Complejo en Cordial

2.2.5 Métodos o Relaciones

Los métodos están representados por una región rectangular. Dentro de esta región se especifica el cuerpo del método que está constituido por formas de representante de instancia, invocaciones de métodos sobre estas y construcciones condicionales.

Las formas de representantes de instancia que pueden aparecer en el cuerpo de un método pueden ser de tres tipos: (1) *self*, una instancia única y obligatoria, pues representa la instancia receptora del mensaje y por lo tanto es de la clase en la que se define el método; (2) cero o varias instancias de otros argumentos; y (3) cero o varias instancias locales. La instancia *self* se etiqueta con la palabra *self* y con un número de argumento 0. Los otros argumentos se pueden etiquetar con cualquier

otro nombre y con un consecutivo empezando con 1 en el sentido de las manecillas del reloj y las instancias locales se pueden etiquetar con cualquier otro nombre, para propósitos de documentación y de identificación. Además, las instancias de los argumentos deben estar conectadas al contorno del método mediante una línea sencilla y el self mediante una línea doble.

Las restricciones iniciales son métodos con algunas limitaciones como: no puede tener parámetros y a excepción del self, no puede tener instancias ni mensajes donde ocurra una instancia de la clase que se está restringiendo.

2.2.6 Mensajes (Invocación de Métodos)

En el contexto de orientación a objetos, un mensaje involucra un emisor, un receptor y unos argumentos; su interpretación en este lenguaje, es que el mensaje establece la relación especificada por el cuerpo del método correspondiente, entre los objetos involucrados; esta relación puede también verse como una restricción entre dichos objetos.

Los mensajes se representan visualmente mediante una región rectangular en forma de sobre de correspondencia conectado con el objeto receptor mediante una línea doble y con los argumentos (si los tiene) mediante líneas sencillas conectados a unos círculos llamados puertos los cuales están etiquetados según el método, aunque estas etiquetas se puede suprimir si no hay ambigüedad; el sobre está también etiquetado en la parte inferior izquierda y con el icono de la clase en la parte superior izquierda. Véase la figura 2.4

Si se desea que un método, para poder ejecutarse, requiera que uno (o varios) de sus argumentos (incluido el *self*) esté determinado (fijo en un valor), se debe usar en su invocación una línea dirigida en lugar de una línea no dirigida del argumento respectivo hacia el buzón; igualmente puede hacerse en la especificación del método, la línea que conecta el parámetro con el entorno, deberá estar dirigida del entorno hacia la forma de instancia respectiva.

En la figura 2.5 el *self* y el parámetro 1 se requiere que estén determinados (o totalmente instanciados).

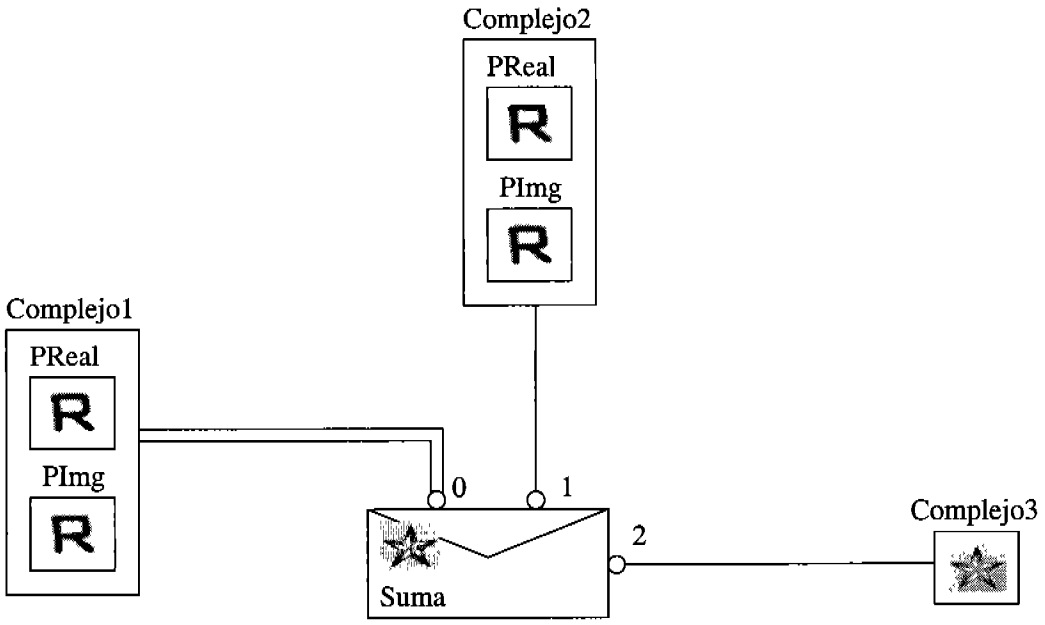

Figura 2.4: Invocación de un Método (Mensaje) en Cordial

2.2.7 Mensajes como Argumentos

Dado que todo método es una instancia de la metaclase método, puede recibir como argumento un mensaje, es decir, la potencial invocación de otro método sobre una instancia, siempre y cuando este método ya esté definido.

2.2.8 Condicionales

El lenguaje ofrece una construcción condicional que permite establecer alternativas guarda-acción (casos) encerradas en un contorno rectangular, como aparece en la figura 2.6.

Para un caso (un par guarda acción), tanto en la región de la guarda como en la de la acción puede aparecer instancias e invocaciones de métodos pero en las guardas no pueden aparecer otros condicionales, además entre casos no puede existir ni invocaciones de métodos ni uso de instancias (i.e. no pueden cruzarse líneas). Las invocaciones de métodos en guardas deben entenderse como chequeos de si las

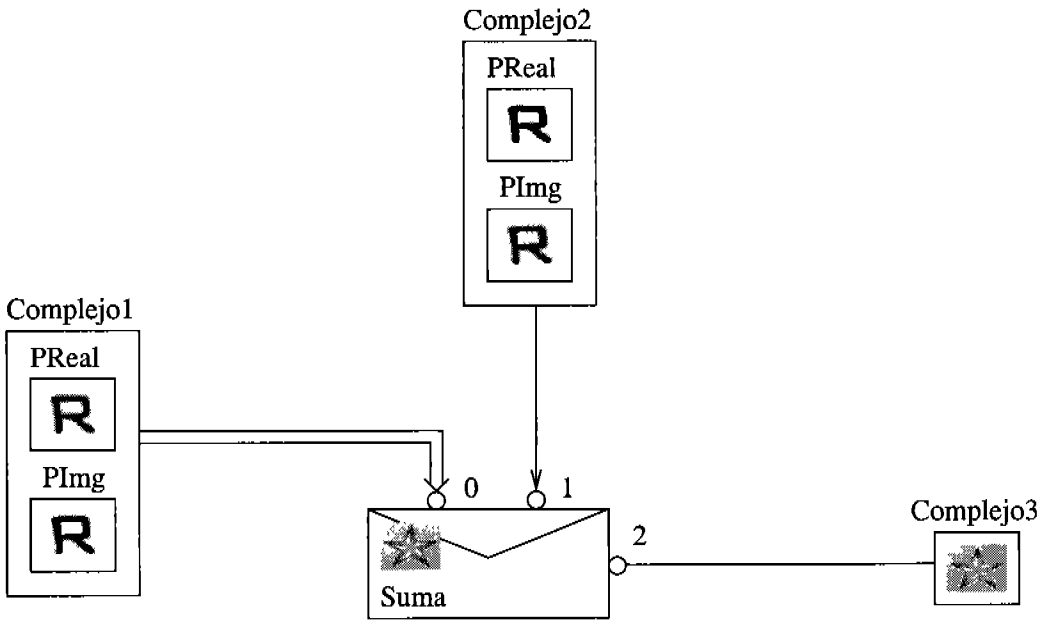

Figura 2.5: Invocación de un Método (Dos Parámetros Determinados) en Cordial

relaciones que establecen si dichos métodos se satisfacen o no. Los casos no se consideran ordenados; la interpretación de esta construcción es que todas las guardas, deben evaluarse concurrentemente, y si alguna (o varias) se satisface(n), se escoge alguna de las acciones correspondientes para ser ejecutada. Si ninguna guarda se puede satisfacer, el programa queda bloqueado.

2.2.9 Concurrencia, Secuenciación y Métodos Especiales

Aunque el lenguaje es, en principio, concurrente, ofrece mecanismos para establecer secuenciación en la "ejecución" de los mensajes. Estos mecanismos se resumen básicamente en el uso de líneas dirigidas, que hacen que la computación (la ejecución de un mensaje) se retrase hasta que las instancias que están en el origen de las mismas estén determinadas y típicamente se hacen necesarios cuando se requiere el uso de métodos que, más que establecer una relación, producen un efecto secundario, tales como la impresión del valor de una variable en la pantalla; en este caso el método que la representa se entiende como la relación entre una instancia (que contiene

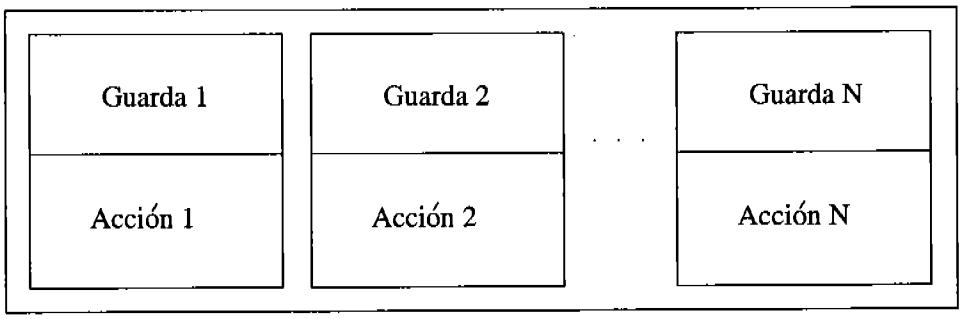

Figura 2.6: Un Condicional en Cordial

el valor que se quiere imprimir), el estado de la pantalla antes de que se imprima y el estado de la misma después de que se imprimió; la lectura es la relación entre una variable (a la que se le envía el mensaje) y el estado del teclado antes de la lectura, el estado del teclado después de la lectura y una nueva variable con el valor leído; análogamente, la asignación es una relación entre un valor, el estado de la variable (antes de la asignación) a la que se le envía el mensaje, y el estado de la variable después de la asignación (que queda ligada a una nueva variable con el valor que se quiere asignar).

2.3 SEMANTICA DE CORDIAL

Cordial es un lenguaje de programación que integra concurrencia, restricciones y programación orientada a objetos. PiCO es un cálculo que integra objetos concurrentes y restricciones donde los objetos y las restricciones son nociones primitivas (véase el capítulo 3).

La traducción de CORDIAL está definida como un conjunto de reglas que representan las estructuras de cordial en términos de tuplas. Esta traducción permite asociar cualquier programa con un proceso PiCO dando una semántica precisa a los programas visuales [QRT98].

3 CALCULO PiCO

3.1 DESCRIPCION GENERAL

PICO [ADQ+98] es un cálculo que integra objetos concurrentes y restricciones como elementos básicos, en contraste con algunos cálculos como TyCO y el cálculo π en los que los elementos básicos son procesos (véase la sección 1.2.3 y la sección 1.2.5). El modelo de objetos es extendido con el manejo de restricciones de una manera ortogonal, adicionando la noción de sistema de restricciones encontrado en el cálculo ρ además de la noción de delegación de mensajes. La extensión del cálculo proviene de una manera natural de expresar más sofisticados comportamientos de comunicación con el uso estándar de la sincronización por el método de paso de mensajes.

Entre los objetivos que se tienen para el cálculo está el ofrecer un modelo computacional que pueda ser adoptado como una herramienta en la construcción de composiciones musicales. Se pueden tener objetos musicales en varias formas, como acordes, patrones armónicos, secuencias de ritmos, melodías, etc.

Actualmente se han propuesto varios cálculos de objetos concurrentes como TyCO visto en la sección 1.2.5, Objetos en el cálculo π visto en la sección 1.2.3 y cálculo de Cardelli [AC96], en los que se presenta un modelo de procesos concurrentes u objetos que son sincronizados principalmente por dos mecanismos: el uso de canales y el paso de mensajes. En TyCO un objeto $a \triangleright [l : (\tilde{y}).P]$ puede ser visto como un proceso P que se encuentra suspendido hasta que algún mensaje es enviado al objeto a con nombre de método l.

De otro lado, las restricciones pueden ser usadas para definir interacciones con procesos concurrentes como se puede ver en el modelo CC [Sar93]. Las operaciones

básicas de *ask* y *tell* permiten definir esquemas complejos de sincronización a través de variables comunes en los procesos. En el lenguaje Oz [HM94], procedimientos de primera clase y celdas de primera clase son usados para simular objetos dentro de un ambiente de restricciones concurrentes, por lo tanto los objetos no son primitivos, de hecho, el modelo sólo interactúa con el paradigma por restricciones. El poder del cálculo de restricciones de Oz es el de permitir otros modelos de programación como el funcional y el de objetos dentro del modelo de Oz.

Dentro de PiCO se tiene una aproximación diferente. Se trata de mantener hasta donde sea posible la independencia entre los modelos de objetos y restricciones al nivel del cálculo. Primero para reflejar mejor las dos aproximaciones de los compositores al construir sus estructuras musicales (Transformaciones sobres las estructuras de datos musicales y planteamiento de reglas sobre las estructuras). Segundo, para permitir adaptar fácilmente diferentes nociones de objetos y diferentes sistemas de restricciones.

Suponga que se desea definir ciertas condiciones sobre la localización de un objeto. Por ejemplo $?(x \in \{a, b, c\}).x \rhd [l : (\tilde{y}).P]$ puede verse como un proceso P el cual está suspendido hasta que un mensaje es enviado a la localización x donde x puede ser uno cualquiera de a,b o c. También puede ser interpretado como un objeto capaz de recibir a través de x, mensajes desde tres diferentes localizaciones. Este ejemplo también es codificable en el cálculo π: $a?[\tilde{y}].P + b?[\tilde{y}].P + c?[\tilde{y}].P$. Sin embargo cuando se quiere definir condiciones arbitrarias para ejecutar P, enviar mensajes o localización de objetos, se necesitan mejores formas de expresar la comunicación. Estas condiciones pueden expresarse de una manera más natural usando restricciones. Por ejemplo:

- $!(x \in \{a_1, a_2, \ldots\}).x \lhd l : (\tilde{J})$ significa "Haga *tell* (Imponga la restricción) que sólo los objetos identificados por $a_1, a_2, \ldots$ se les envía el mensaje $x \lhd l : (\tilde{J})$"
- $*x \rhd [l : (\tilde{y})?(x \in \{a, b\}).P)]$ significa "Ejecute el proceso (método) P solamente si el mensaje es enviado a una localización llamada por a o b".

La interacción con objetos puede modelarse de dos maneras. Primero, con la noción de objetos concurrentes donde la ejecución sincronizada es simulada por cambios en

los objetos reales [Mil91], y segundo, con el uso de restricciones para "cambiar" el estado del objeto redefiniendo el conocimiento parcial que se tiene de cada atributo de los objetos.

3.2 SINTAXIS

Dentro de la sintaxis de PiCO (tabla 3.1), existen tres tipos básicos de procesos: *Mensajes, Objetos* y *Restricciones.*

Procesos Normales: N	::=	O	Proceso nulo
	|	$I \lhd m.P$	Mensaje a I
	|	$(I, J) \rhd M$	Objeto I con delegación a J
Proceso de restricción: R	::=	$!\phi.P$	Proceso *tell*
	|	$?\phi.P$	Proceso *ask*
Procesos: P, Q	::=	$(\upsilon x)P$	Variable nueva x en P
	|	$(\upsilon a)P$	Nombre nuevo a en P
	|	N	Procesos normales
	|	$P \mid Q$	Composición
	|	$*P$	Procesos replicados
	|	R	Proceso de restricción
Identificadores de Objetos: I, J	::=	a	Nombre
	|	v	Valor
	|	x	Variable
Colección de Métodos M	::=	$[l_1 : (\widetilde{x_1})P_1 \& \ldots \& l_m : (\widetilde{x_m})P_m]$	
Mensaje m	::=	$l : [\widetilde{I}]$	

Tabla 3.1: Sintaxis de PiCO

En adelante se denotara la secuencia $t_1, \ldots t_k$, como $\widetilde{t}$ con longitud $|\widetilde{t}| = k$.

El proceso nulo es aquel que no hace nada. Un proceso $(I, J) \rhd M$ donde M es una colección $[l_1 : (\widetilde{x_1})P_1 \& \ldots \& l_m : (\widetilde{x_m})P_m]$ puede ser visto como un objeto direccionado por I (Localizado en I, Llamado o identificado por I) cuyos métodos

$(\widetilde{x_1})P_1 \& \ldots (\widetilde{x_m})P_m$ son identificados por un conjunto de nombres distintos, e.d. $Labels(M) = \{l_1 \ldots l_m\}$. El identificador J, es la dirección de delegación y representa la dirección donde el mensaje debe ser reenviado si no existe el método apropiado al realizar un llamado. Objetos de la forma $(I, J) \rhd M$ representan objetos con delegación, en general cuando $I = J$ el objeto no tiene delegación y por abreviación se escribirá $I \rhd M$.

En un método $l : (\tilde{x})P$, $\tilde{x}$ representa los parámetros formales y P el cuerpo del método. *Nombre, variables* y *valores primitivos* pueden ser usados como identificadores de objetos.

Un proceso $I \lhd : [\tilde{J}].P$ especifica un mensaje dirigido al objeto I con argumentos $\tilde{J}$. A los mensajes se le permite que tengan continuación P, a diferencia de los objetos. La etiqueta l es usada para seleccionar el método correspondiente en el objeto al cual se dirige el mensaje. El resultado de la interacción del mensaje y el objeto es el cuerpo del método seleccionado, reemplazando los argumentos formales por sus correspondientes argumentos actuales contenidos en el mensaje, siempre y cuando el método seleccionado exista en la definición del objeto; en caso contrario, el mensaje es delegado a la dirección de delegación si el objeto tiene declarada la delegación. Este comportamiento se utiliza para codificar subclases.

El proceso $(va)P$ restringe el uso del nombre a al proceso P. Otra forma de ver es que $(va)P$ declara un único (nuevo) nombre a distinto de todos los nombres externos que son usados en P. Similarmente $(vx)P$ declara una nueva variable (lógica), distinta de todas las variables externas usadas en P.

La composición de procesos $P|Q$ denota la ejecución concurrente de procesos P y Q. El proceso $*P$ (replicación) expresa la composición $P|P \ldots$ (tantas copias sean necesarias). Una instancia común de la replicación es $*I \rhd M$, un objeto que se puede reproducir cuando se realiza una comunicación sobre el objeto I. Replicación es comúnmente usado para codificar procesos recursivos.

Los procesos de restricciones son un nuevo tipo de procesos cuyo comportamiento depende de una memoria de restricciones (store). Un store contiene información proporcionada por las restricciones. Esta memoria es usada en PiCO para controlar todas las comunicaciones. El proceso *tell* (escrito $!\phi.P$) significa "Adicione ϕ a

la memoria de restricciones y luego ejecute P". Esto es, el proceso tell es usado para adicionar información a la memoria de restricciones y así influenciar el comportamiento de otros procesos. El proceso *ask* $(?\phi.P)$ significa "Ejecute el proceso P si la restricción ϕ es una consecuencia lógica de la información almacenada en la memoria de restricciones o elimine a P si $\neg\phi$ lo es. En caso contrario, suspenda a $?\phi.P$ hasta que el store contenga suficiente información para reducirlo".

3.3 SEMANTICA OPERACIONAL

3.3.1 Sistema de Restricciones

PiCO está parametrizado en un *sistema de restricciones* [Sar93, Smo94]. Se utilizan los predicados de primer orden para especificar las sentencias usadas en el sistema de restricciones.

Un sistema de restricciones consiste de:

- Una signatura Σ (Un conjunto de funciones, restricciones y símbolos de predicado con igualdad) incluyendo un conjunto infinito distinguido $\mathcal{N}$, de constantes llamado *nombres* denotados por $a, b, \ldots, u$. Otras constantes, llamadas *valores*, son escritas como $v_1, v_2, \ldots$. Estos son considerados como objetos primitivos en el cálculo y usados como identificadores de objetos.
- Una teoría consistente Δ (un conjunto de sentencias sobre Σ teniendo un modelo) que satisface dos condiciones:
  1. $\Delta \models \neg(a = b)$ para cada par a,b distintos.
  2. $\Delta \models \phi \leftrightarrow \psi$ para cada par ϕ, ψ sobre Σ tal que ψ puede ser obtenida de ϕ por permutación de nombres.

A menudo Δ puede ser dado como un conjunto de todas las sentencias validas en cierta estructura (por ejemplo, árboles finitos, enteros, racionales). Dado un sistema de restricciones, los símbolos $\phi, \psi, \ldots$ denotan formulas de primer orden en Σ, de

aquí en adelante llamadas *restricciones*. Se dice que ϕ *deriva* a ψ en Σ, escrito $\phi \models \psi$, si y sólo si $\phi \rightarrow \psi$ es verdad en todos los modelos de Δ. Se dice que ϕ es *equivalente* a ψ en Σ, escrito $\phi \models|_{\Delta} \psi$, si y sólo si, $\phi \models \psi$ y $\psi \models \phi$. Se dice que ϕ es *satisfacible* en Δ, si y sólo si, $\phi \not\models_{\Delta} \bot$. Se usará $\bot$ para las restricciones que siempre son falsas y $\top$ para las restricciones que siempre son verdaderas. Un sistema de restricciones particular debe tener una relación de derivación decidible y eficiente.

Se usan $x, y, \ldots \in \mathcal{V}$ para denotar variables lógicas, designando algunos elementos fijos pero desconocidos en el dominio bajo consideración. El conjunto $fv(\phi) \subset \mathcal{V}$ y $bv(\phi) \subset \mathcal{V}$ denota el conjunto de variables libres y ligadas respectivamente en ϕ. Finalmente, $fn(\phi) \subset \mathcal{N}$ es el conjunto de nombres que aparecen en ϕ.

Como se dijo antes, los procesos de restricciones actúan relativos a un "store". Un "store" está definido en términos del sistema de restricciones:

Definición 3.3.1 (Store) *Un store* $S = \phi_1 \wedge \phi_2 \wedge \ldots \wedge \phi_r$ *(con* $r \geq 0$*) es una restricción en* Σ. *Cuando* $r = 0$, *se dice que* S *es un store vacío (i.e.* $S = \top$*). Cuando* $S \models_{\Delta} \bot$, *se dice que el store* S *es insatisfacible.*

La semántica operacional de PiCO será definida en términos de una relación de equivalencia, $\equiv_P$, sobre una configuración que describe los estados computacionales y un paso de relación de reducción, $\longrightarrow$ describiendo las transiciones de estas configuraciones. Una configuración es la tupla $\langle P; S \rangle$ considerando el proceso P y el store S.

3.3.2 Congruencia Estructural y Equivalencia de Configuraciones

Primero, en PiCO se distingue un operador para ligar: el operador para ligar nombres es $(va)P$ que declara un nuevo nombre a en P. Hay sólo dos operadores para ligar variables: $(vx)P$ el cual liga a x en P y $(x_1 \ldots x_n).P$ y declara parámetros formales $x_1, \ldots, x_n$ en P. También se definen *nombres libres* $fn(P)$, *ligaduras de nombres* $bn(P)$, *variables libres* $fn(P)$ y *ligaduras de variables* $bv(P)$ en un proceso P de

igual manera. El conjunto de variables que aparecen en P, $v(P)$ son $fv(P) \cup bv(P)$ y similarmente el conjunto de nombres que aparecen en P, $n(P)$, es $fn(P) \cup bn(P)$.

Se define la congruencia estructural de la misma forma como se realiza en el cálculo π en [Mil91].

Definición 3.3.2 (Congruencia estructural) *Se define la congruencia estructural* $\equiv$, *como la más pequeña relación de congruencia sobre la satisfacción de procesos con los siguientes axiomas:*

- *Los procesos son idénticos si sólo difieren por un cambio de variables ligadas o nombres ligadas* ($\alpha - conversion$).
- $(\mathcal{P}/\equiv, |, O)$ *es un monoide simétrico, donde* $\mathcal{P}$ *es un conjunto de procesos.*
- $(I, J) \rhd M \equiv (I, J) \rhd M'$ *si* M' *es una permutación de* M.
- $*P \equiv P \mid *P$.
- $(va)O \equiv O$, $(vx)O \equiv O$, $(va)(vb)P \equiv (vb)(va)P$, $(vx)(vy)P \equiv (vy)(vx)P$, $(va)(vx)P \equiv (vx)(va)P$.
- *Si* $a \notin fn(P)$ *entonces* $(va)(P \mid Q) \equiv P \mid (va)Q$.
- *Si* $x \notin fv(P)$ *entonces* $(vx)(P \mid Q) \equiv P \mid (vx)Q$.
- *Si* $\phi \models|_{\Delta} \psi$ *y* $P \equiv Q$ *entonces* $!\phi.P \equiv !\psi.Q$ *y* $?\phi.P \equiv ?\psi.Q$

Definición 3.3.3 (Relación P-equivalente) *Se Dirá que* $\langle P_1; S_1\rangle$ *es una* P*-equivalencia de* $\langle P_2; S_2\rangle$, *lo cual se escribe* $\langle P_1; S_1\rangle \equiv_P \langle P_2; S_2\rangle$, *si* $P_1 \equiv P_2$, $S_1 \models|_{\Delta} S_2$, $fn(S_1) = fn(S_2)$ *y* $fv(S_1) = fv(S_2)$. $\equiv_P$ *se dice que es una relación* P*-equivalente sobre configuraciones.*

El comportamiento de un proceso P está definido por transiciones desde una configuración inicial $\langle P; \top\rangle$. Una transición, $\langle P; S\rangle \longrightarrow \langle P'; S'\rangle$, significa que $\langle P; S\rangle$ puede ser transformado en en $\langle P'; S'\rangle$ en un paso de reducción. Por simplicidad se asume que todas las variables y nombres son declarados en la configuración inicial.

3.3.3 Relación de Reducción

La relación de reducción, $\longrightarrow$, está definido sobre configuraciones y se aplica si al menos satisface una de las reglas que aparece en la tabla 3.2.

$$\text{COMM: } \frac{S \models_{\Delta} I = I' \qquad |\widetilde{K}| = |\widetilde{x}|}{\left\langle I' \lhd l{:}[\widetilde{K}].Q \mid (I,J) \rhd [l{:}(\widetilde{x})P \& \ldots]; S \right\rangle \longrightarrow \left\langle Q \mid P\left\{\widetilde{K}/\widetilde{x}\right\}; S \right\rangle}$$

$$\text{DELEG: } \frac{S \models_{\Delta} I' = I \wedge I' \neq J \qquad l \notin Labels(M)}{\left\langle I' \lhd l{:}[\widetilde{K}].Q \mid (I,J) \rhd M; S \right\rangle \longrightarrow \left\langle J \rhd l{:}[\widetilde{K}].Q \mid (I,J) \rhd M; S \right\rangle}$$

$$\text{TELL: } \langle !\phi.P; S \rangle \longrightarrow \langle P; S \wedge \phi \rangle$$

$$\text{ASK: } \frac{S \models_{\Delta} \phi}{\langle ?\phi.P; S \rangle \longrightarrow \langle P; S \rangle} \quad , \quad \frac{S \models_{\Delta} \neg\phi}{\langle ?\phi.P; S \rangle \longrightarrow \langle O; S \rangle}$$

$$\text{PAR: } \frac{\langle P; S \rangle \longrightarrow \langle P'; S' \rangle}{\langle Q \mid P; S \rangle \longrightarrow \langle Q \mid P'; S' \rangle}$$

$$\text{DEC-V: } \frac{x \notin fv(S),\ \langle P; S \gg \{x\} \rangle \longrightarrow \langle P'; S' \rangle}{\langle (\upsilon x)P; S \rangle \longrightarrow \langle P'; S' \rangle}$$

$$\text{DEC-N: } \frac{a \notin fn(S),\ \langle P; S \gg \{a\} \rangle \longrightarrow \langle P'; S' \rangle}{\langle (\upsilon a)P; S \rangle \longrightarrow \langle P'; S' \rangle}$$

$$\text{EQUIV: } \frac{\langle P_1; S_1 \rangle \equiv_P \langle P_1'; S_1' \rangle \qquad \langle P_2; S_2 \rangle \equiv_P \langle P_2'; S_2' \rangle \qquad \langle P_1; S_1 \rangle \longrightarrow \langle P_2; S_2 \rangle}{\langle P_1'; S_1' \rangle \longrightarrow \langle P_2'; S_2' \rangle}$$

Tabla 3.2: Sistema de transición de PiCO

COMM describe la interacción entre un mensaje $I' \lhd l : [\widetilde{K}].Q$ y el objeto $(I, J) \rhd [l : (\widetilde{x})P\& \ldots]$. El store es usado para decidir si el mensaje y el objeto se pueden comunicar. El proceso $P\{\widetilde{K}/\widetilde{x}\}$ se obtiene al remplazar, en paralelo, cada ocurrencia de variables $\widetilde{x}$ por identificadores (valores, nombres o variables) de $\widetilde{K}$. La continuación del mensaje Q, se activa cuando el mensaje es recibido.

DELEG describe la delegación de un mensaje. Sea $I \lhd l : [\widetilde{K}].Q$ un mensaje enviado al objeto $(I, J) \rhd M$ y se supone que el método l no existe en M. En este caso el mensaje es enviado ahora por delegación al objeto direccionado por J especificado por el objeto.

Las reglas ASK y TELL describen la interacción entre los procesos de restricciones

y el store. TELL es la manera de adicionar información al store. ASK es el método para obtener información del store. La regla $?\phi.P$ especifica que P puede ser activado si del store actual S, se puede deducir ϕ, o descargado si se deduce $\neg\phi$. Si del store actual S no se puede deducir ϕ el proceso queda suspendido por S.

PAR dice que la reducción puede ocurrir dentro de una composición (se puede reducir primero un proceso y luego el otro en una construcción paralela). DEC-V y DEC-N son la manera de introducir nuevas variables y nombres respectivamente. La regla EQUIV simplemente dice que configuraciones P-equivalentes tienen la misma reducción.

3.4 DEFINICIONES RECURSIVAS

Es a menudo conveniente definir procesos recursivos. La definición de procesos recursivos tienen la forma $D(x_1, \ldots, x_n) \stackrel{def}{=} P$, donde P puede tener ocurrencias de D. La aplicación de una definición tiene la forma $D(I_1, \ldots, I_n)$.

La creación de definiciones y sus aplicaciones no son primitivos en PiCO, pero se pueden codificar fácilmente con el uso de replicación. La idea es remplazar la definición $D(x_1, \ldots, x_n) \stackrel{def}{=} P$ por un objeto replicado $*D \rhd [l : (x_1, \ldots, x_n)P]$ y cada aplicación $D(I_1, \ldots, I_n)$ por el mensaje $D \lhd l : [I_1, \ldots, I_n]$.

3.5 CLASES Y SUBCLASES

3.5.1 Clases de Objetos

En PiCO las clases y atributos son representados con el uso de definiciones recursivas y restricciones. Los atributos son representados en PiCO por variables restringidas con información parcial dado por una operación de tell. Las clases pueden ser vistas como objetos generadores planteando unas condiciones iniciales o restricciones de clase sobre los atributos. Las restricciones de clase se deben satisfacer así los atributos cambien y definen las propiedades fundamentales que tiene la clase. Las clases pueden ser codificadas como $C(self, x_1, \ldots, x_n) \stackrel{def}{=} !\phi[x_1, \ldots, x_n].self \rhd M$ donde C

denota el nombre (o identificador) para la clase, la variable $self$ denota el identificador del objeto y $x_1, \ldots, x_n$ los atributos del objeto. La operación $!\phi[x_1, \ldots, x_n]$ impone a cada objeto de la clase una restricción ϕ sobre sus atributos.

Usando la codificación recursiva la representación de clase se puede escribir de la forma $*C \triangleright [new : (self, x_1, \ldots, x_n)!\phi[x_1, \ldots, x_n].self \triangleright M']$, donde M' resulta del reemplazo en M de cada ocurrencia de $C(self, \ldots)$ por $C \triangleleft new : [self, \ldots]$.

3.5.2 Subclases

Las subclases pueden ser definidas como una extensión de una clase predefinida (u otra subclase) conocida como *superclase*. La extensión involucra nuevos atributos y nuevos métodos con redefinición posible por medio de la delegación.

$SB(self, \widetilde{x}, \widetilde{y}) \stackrel{def}{=} !\phi[\widetilde{x}, \widetilde{y}].(vsuper)((self, super) \triangleright M \mid SP(super, \widetilde{x}))$

Donde SB identifica a la subclase, SP a la superclase, M representa la colección de nuevos métodos y $\widetilde{y}$ representa los nuevos atributos. La expresión $!\phi[\widetilde{x}, \widetilde{y}]$ restringe los atributos nuevos y viejos.

4 DISEÑO DE LA MAQUINA ABSTRACTA MAPiCO

En este capítulo se presentan las especificaciones de diseño de una máquina abstracta para el cálculo PiCO. Las reglas de transición de este cálculo han sido adaptadas para formar el núcleo de la máquina abstracta mientras que se mantiene como parámetro el sistema de restricciones. También se muestran las estructuras de soporte de la máquina, el formato de las instrucciones y codificación, seguido de un ejemplo simple y de una descripción de cómo sería su comportamiento interno.

4.1 POSIBLES ALTERNATIVAS

Dentro de las posibles alternativas que se tuvieron en cuenta para el diseño de la máquina abstracta están:

Definir una estructura de memoria para almacenar los objetos que no pueden comunicarse o están replicados, tener también dos colas de ejecución las cuales funcionan de forma alternada, es decir, se parte con una cola y se pasan a la otra los procesos que no han podido ser reducidos. Para controlar el final de la ejecución se cuenta con una bandera que marca si al terminar la cola que se esta procesando se realizó alguna reducción de procesos, de lo contrario, la ejecución termina.

Otra alternativa que se tuvo en cuenta es como en la alternativa anterior, tener una estructura para almacenar los objetos que no pueden comunicarse o están replicados al igual que una estructura para almacenar los mensajes que no se pueden comunicar y tener una cola de ejecución donde se encuentran los procesos por ejecutar. Cuando

se encuentra un proceso *ask* que no se puede deducir del Store, se encola nuevamente. La ejecución termina cuando todos los procesos en la cola de ejecución son procesos *ask* o la cola está vacía.

Se presentaron modificaciones de las alternativas para mejorar algunas características tales como, el número de veces que se trataba de reducir un proceso, la facilidad en la reducción de procesos, la memoria utilizada, velocidad de ejecución, etc. hasta llegar a la alternativa expuesta a continuación.

4.2 DESCRIPCION GENERAL

Aunque la especificación y la reglas de reducción de la máquina abstracta MAPiCO sigue las del cálculo PiCO, muchas alternativas fueron examinadas para construir la especificación más eficiente posible para la máquina.

MAPiCO está compuesta por dos áreas de memoria, cuatro colas para los diferentes estados de ejecución de un proceso, un control donde se encuentran los registros internos (sección 4.6) y un sistema de restricciones que provee un almacenamiento de restricciones y las operaciones *ask* y *tell* (Ver figura 4.1).

El código que MAPiCO ejecuta es una secuencia de instrucciones llamada código de bytes (*bytecode*). La máquina asume que este código es correcto y por lo tanto no desarrolla ningún tipo de prueba o chequeo de integridad.

4.3 ESTRUCTURAS DE SOPORTE

Las estructuras de soporte se usan para almacenar información requerida en la ejecución de un programa dentro de la máquina. Entre estas estructuras están las áreas de memoria donde se guarda el código de un programa, las definiciones de objetos y los elementos transitorios.

Los elementos básicos en el cálculo PiCO son: los objetos, los mensajes, las restricciones, las variables y los nombres. Estos deben ser modelados en la máquina.

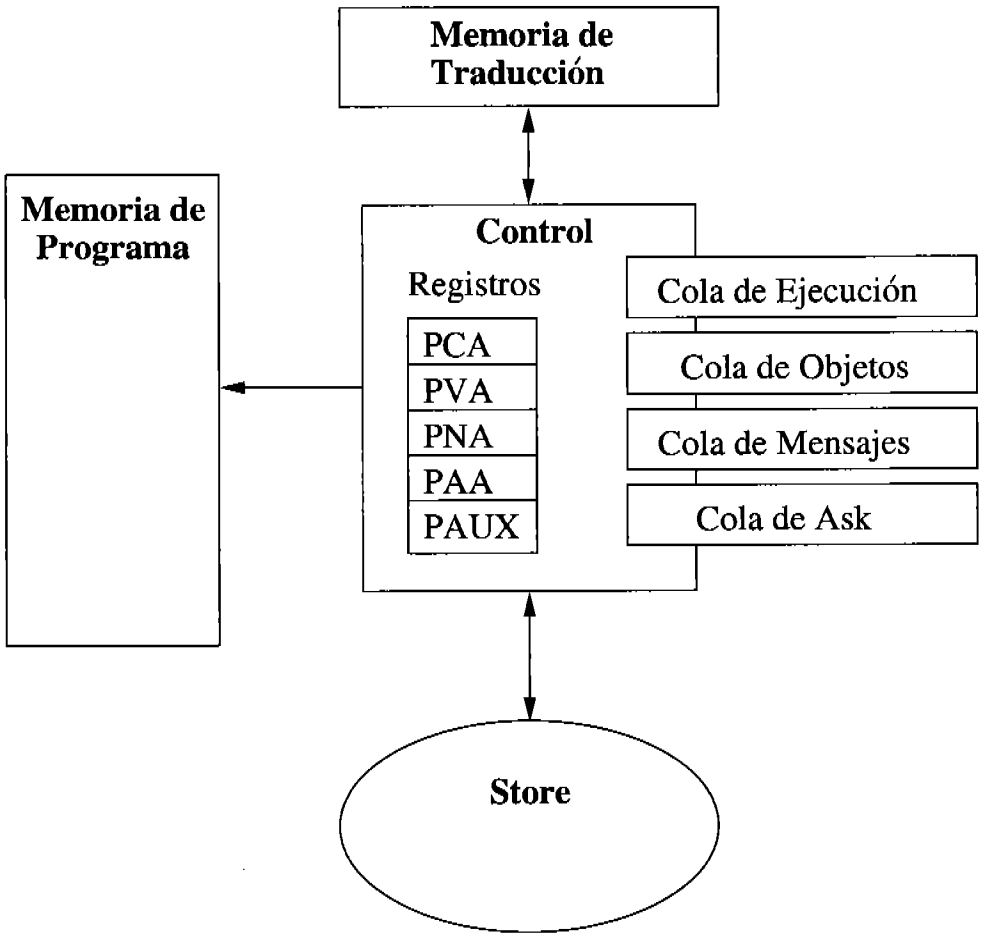

Figura 4.1: Diagrama de bloques de MAPiCO

Se tienen cuatro colas de procesos, tres de las cuales son para los procesos que quedan suspendidos, esto es, se tiene una cola para los objetos que no se han comunicado aún o son objetos replicados (tabla 3.1), otra de mensajes donde están los mensajes replicados o los mensajes donde los objetos destinos no existen o no se puede deducir del *store* que un objeto existente sea el destino de dicho mensaje y una tercera cola para los *ask* que no se pueden deducir de la información que se encuentra en el *store*. La cuarta cola es la cola de procesos que están a la espera de ser ejecutados.

El sistema de restricciones es monotónicamente creciente, e.d. una vez una restricción se puede deducir, esta siempre será deducida en adelante. Hay dos operaciones que interactúan con el *store*: el "ask" para hacer preguntas y el "tell" para adicionar

información a él.

El programa se ubica en una memoria estática (Program memory) donde se almacenan las instrucciones que van a ser ejecutadas.

Por último se encuentra un área de memoria dinámica que se utiliza para tener las traducciones o ligaduras de variables y nombres al store, también se almacenan allí los parámetros para el llamado de métodos y se crea el árbol sintáctico que describe una restricción.

De lo anterior se tiene las siguientes estructuras de datos:

- RunQ: Cola de ejecución. (Procesos listos para ejecución)
- ObjQ: Cola de Objetos suspendidos.
- MsgQ: Cola de mensajes suspendidos.
- AskQ: Cola de procesos Ask suspendidos.
- S: Store. (Almacenamiento de las restricciones)
- Area de Memoria de traducción o ligadura, donde están almacenados las pilas de los procesos y el árbol sintáctico con las restricciones que van a comunicarse con el Store.
- Area de memoria del programa (Program memory) donde están representadas las instrucciones por ejecutar. Esta memoria es estática (no cambia durante la ejecución).
- Registros Internos. (Registros internos del máquina abstracta, ver la sección 4.6).

En adelante una cola vacía será representada con "•". Se usará el signo "::" para representar la concatenación de dos colas o de un elemento y una cola, según el caso. Igualmente el signo "$\oslash$" representa la no existencia de ligadura o enlaces de variables y T representa un "store" vacío.

El estado de la máquina está representado por la tupla:

$$\langle Hilo, HBind, HAux, ObjQ, MsgQ, AskQ, RunQ, Store \rangle$$

Donde Hilo, HBind y HAux representan el proceso que está siendo ejecutado (ver la sección 4.5).

El área de memoria del programa no se toma en cuenta dentro del estado de la máquina ya que esta es estática y no cambia durante la ejecución.

4.4 ESPECIFICACION FORMAL DE LA MAQUINA

Se ha hecho una simplificación en el cálculo PiCO para hacer que la máquina sea más eficiente: Se ha restringido el operador de replicación $*P$ (tabla 3.1) restringiendo P solamente a procesos normales. (Véase la tabla 4.1).

Para poder simplificar la sintaxis de PiCO, se realizó una aproximación respecto a los procesos replicados (ecuación 4.1). Cabe anotar que para que esta aproximación sea valida, la máquina debe tener un orden de ejecución de tal manera que se ejecute P antes de hacer la siguiente comunicación. Con esta aproximación se logra un mejor balance entre la sintaxis del cálculo y la ejecución de la composición de procesos.

$$*P \approx (va)(*(a,a) \rhd [rep : ()P|a \lhd rep[]] \mid a \lhd rep[]) \tag{4.1}$$

En general, se elimina la replicación de la creación de: variables, la creación de nombres, de los procesos restrictivos y de la composición de procesos y sólo se permite la replicación de procesos normales, similar a lo planteado en [PT96], donde se elimina la replicación de escritores.

La secuencia $\vec{P} = P_1|...|P_k$ para $k = 0$ es designado con la palabra *nil*.

A continuación se presenta una descripción formal de cada uno de los elementos de la máquina.

Se define un conjunto de ligaduras de nombres y variables a referencias dentro del Store como *HBind* (en forma corta B) como un conjunto de ligaduras de la forma

N	::=	O	Proceso nulo
	|	$I \lhd m.\vec{P}$	Mensaje a I
	|	$(I, J) \rhd M$	Objeto I con delegación a J
R	::=	$!\phi.\vec{P}$	Proceso *tell*
	|	$?\phi.\vec{P}$	Proceso *ask*
P, Q	::=	$(\upsilon x)\vec{P}$	Variable nueva x en P
	|	$(\upsilon a)\vec{P}$	Nombre nuevo a en P
	|	N	Procesos normales
	|	$*N$	Procesos normales replicados
	|	R	Proceso de restricción
I, J	::=	a	Nombre
	|	x	Variable
M	::=	$[l_1 : (\widetilde{x_1})\vec{P}_1 \& \ldots \& l_n : (\widetilde{x_n})\vec{P}_n]$	
m	::=	$l : [\tilde{I}]$	
$\vec{P}$	::=	$P_1 \mid \ldots \mid P_k$	Hilo de ejecución
l	$\in$	Etiquetas	

Tabla 4.1: Sintaxis modificada de PiCO

$I_1 \mapsto L_1, ..., I_n \mapsto L_n$, teniendo en cuenta que $L \in StoreRef$ donde $StoreRef$ son el conjunto de variables que ocurren dentro del Store.

Para pasar las restricciones al sistema de restricciones o para la comunicación entre mensajes y objetos se define a $HAux$ (en forma corta H) que se define como un conjunto de ligaduras de nombres y variables o una restricción en términos de L: $HAux ::= HBind \mid \phi(\tilde{L})$.

Se puede definir al *Store* (de forma corta S) como una conjunción de restricciones dadas en términos de L: $Store = \{ \phi_1(\tilde{L}) \wedge ... \wedge \phi_m(\tilde{L}) \}$. Por ejemplo $L_1 = 5$, $L_2 > 3$.

Las colas se usan para implementar la cola de ejecución como también la cola de

objetos, mensajes y ask. La definición de cola está dada de forma paramétrica:

$$Cola(X) ::= X :: ... :: X \mid \bullet$$

La cola de objetos $ObjQ$ (Oq) se define como una cola de la estructura de ligaduras, parámetros y un proceso objeto: $ObjQ = Cola(HBind \times HAux \times Obj)$ donde $Obj ::= (I, J) \rhd M \mid *(I, J) \rhd M$.

La cola de mensajes $MsgQ$ (Msq) se define como una cola de la estructura de ligaduras, parámetros y un proceso de tipo mensaje: $MsgQ = Cola(HBind \times HAux \times Msg)$ donde $Msg ::= I \lhd m.\vec{P} \mid *I \lhd m.\vec{P}$.

La cola de ask $AskQ$ (Aq) se define como una cola de la estructura de ligaduras, una restricción y un proceso Ask: $AskQ = Cola(HBind \times HAux \times Ask)$ donde $Ask ::= ?\phi.\vec{P}$.

Cada elemento de la cola de ejecución $RunQ$ (Rq) contiene la estructura de ligaduras, parámetros o una restricción y un hilo de ejecución $(\vec{P})$: $RunQ = Cola(HBind \times HAux \times Hilo)$.

4.5 PROCESOS

Los procesos son los elementos básicos del cálculo. En la máquina los procesos se modelan como el conjunto de información requerida para la ejecución del proceso PiCO.

Se puede definir a un proceso en MAPiCO como una estructura conformada por los siguientes campos:

- PC (Puntero a código): Este registro apunta al código de la instrucción (en la memoria del programa) que está próxima a ser ejecutada en el proceso.
- PV (Puntero a Variables): Apunta al primer elemento (en la memoria de traducción) de una lista donde están la ligadura de las variables dentro del *Store*. Esta lista cambia con las instrucciones *NEWV* o *POP* (véase la sección 4.11).

- PN (Puntero a Nombres): Apunta al primer elemento (en la memoria de traducción) de una lista donde está la ligadura de los nombres dentro del *Store*. Esta lista cambia con las instrucciones *NEWN* o *POP* (véase la sección 4.11).

- PA (Puntero Auxiliar): Similar a PV y PN, se utiliza la memoria de traducción. Este puntero tiene dos usos. Primero se utiliza para apuntar a una lista con los argumentos y el objeto receptor en el momento de hacer una comunicación. Para manipular esta lista, se usa *PUSHV*, *PUSHN* y *POP* (véase la sección 4.11). Segundo, el PA también se utiliza para guardar referencia a nodos en la creación de un árbol sintáctico que define una restricción (véase la sección 4.11.2).

Se Puede entonces, denotar a un proceso por la tupla $< PC, PV, PN, PA >$. La máquina en su estado inicial comienza con la tupla $< 0, \oslash, \oslash, \oslash >$, llamada proceso inicial, que es un proceso que apunta a la primer instrucción en la memoria de programa y todos los demás apuntadores son nulos.

4.6 REGISTROS DE LA MAQUINA

La máquina tiene unos registros de uso específico que guardan información de la memoria y de los procesos; ella modifica esta información para luego actualizar los procesos y las colas.

- PCA (Puntero a código Actual): Este registro apunta a la instrucción que está siendo ejecutada en la máquina.

- PVA (Puntero a Variables Actual): Apunta al PV del proceso en ejecución.

- PNA (Puntero a Nombres Actual): El PNA Apunta al PN del proceso en ejecución.

- PAA (Puntero a Auxiliar Actual): El PAA apunta al PA del proceso que se está ejecutando.

- PAUX (Puntero Auxiliar): Usado para manipulación y ejecución de procesos. Véase la sección 4.12.

4.7 MEMORIA DE TRADUCCION

La memoria de traducción es un espacio de memoria dinámica donde están almacenados los enlaces o "bindings" de las variables y las restricciones que serán enviadas al sistema de restricciones.

Dentro de esta memoria de traducción o ligadura se pueden almacenar tres tipos de información durante la ejecución de un programa como son: la ligadura de las variables, los argumentos y las restricciones.

4.8 MARCOS (ESTRUCTURAS) EN LA MEMORIA DE TRADUCCION

La información de la ligadura de un nombre o variable tiene la estructura presentada en la figura 4.2.

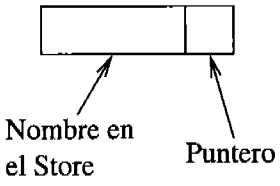

Figura 4.2: Estructura de Traducción de Ligaduras en MAPiCO

El campo nombre es de 16 bits y es usado para guardar la ligadura que tendrá una variable o nombre en un ambiente determinado.

Con estas estructuras la máquina forma árboles ascendentes que son árboles donde los nodos apuntan a su padres (véase la sección 4.12.2).

Para los argumentos en la comunicación de mensajes, se tiene el esquema de la figura 4.3.

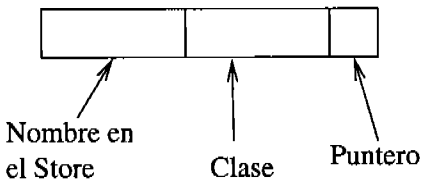

Figura 4.3: Estructura de Argumentos en MAPiCO

La clase en este caso se refiere a si el argumento es una variable o un nombre. Con esta estructura de datos, se implementan listas simplemente encadenadas que simulan una pila.

El árbol sintáctico que se crea para representar restricciones tiene el esquema de la figura 4.4.

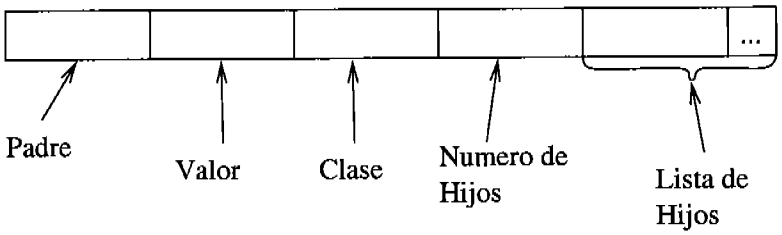

Figura 4.4: Estructura de Restricciones en MAPiCO

El campo Clase indica si el nodo es un término, átomo o sentencia (ver la sección 4.11.2). Con esta estructura, la máquina forma árboles n-arios cuyo número de hijos depende de la aridad de la función en caso de que la estructura esté apuntando a una función.

4.9 REGLAS DE REDUCCION DE LA MAQUINA

El comportamiento de MAPiCO, como el de otras máquinas que implementan cálculos computacionales [LV97, PT96], sigue una semántica operacional, en este caso la semántica del cálculo PiCO. Se han especificado reglas de transición entre configuraciones o estados, que son una mejora a las reglas propuestas en [ABH98] ya que son presentadas de una manera más particular.

Estas reglas son de la forma:

$$\left\langle \vec{P}, B, H, Oq, Msq, Aq, Rq, S \right\rangle \longrightarrow \left\langle \vec{P}', B', H', Oq', Msq', Aq', Rq', S' \right\rangle$$

Cada regla examina el proceso conformado por el hilo de ejecución y el ambiente actual y si se aplica alguna regla (considerando el contexto), transforman la configuración inicial en una configuración final, reduciendo el proceso.

La parte superior de la regla es un condicional que se debe cumplir con la configuración inicial de la máquina.

El proceso SCHED remueve el proceso actual en ejecución, permitiendo que el siguiente proceso en la cola de ejecución sea considerado para reducir. En esta regla, se hace necesario liberar la memoria ocupada por H y en algunos casos la ocupada por B o utilizar un recolector de basura:

$$\text{SCHED:}\left\langle nil, B, H, Oq, Msq, Aq, (B', H', \vec{P}) :: Rq, S \right\rangle \longrightarrow \left\langle \vec{P}, B', H', Oq, Msq, Aq, Rq, S \right\rangle$$

El proceso $(vx)\vec{P}$ o $(va)\vec{P}$ crea un nuevo enlace de x a L donde L es realmente una nueva variable o nombre (generada por el sistema):

$$\text{NEW-REF} : \frac{L \ Nueva}{\left\langle (vx)\vec{P}, B, H, Oq, Msq, Aq, Rq, S \right\rangle \longrightarrow \left\langle \vec{P}, B+(x \mapsto L), H, Oq, Msq, Aq, Rq, S \right\rangle}$$

Donde $B + (x \mapsto L)$ denota la adición de una nueva referencia al store dentro de la memoria de traducción.

Cuando hay una composición paralela $(P|Q)$, se crea un nuevo proceso Q, poniéndolo al final de la cola de ejecución (para ser ejecutado luego) y continuando con la ejecución de P:

$$\text{PARALLEL:} \ \frac{}{\langle (P|Q), B, H, Oq, Msq, Aq, Rq, S \rangle \longrightarrow \langle P, B, H, Oq, Msq, Aq, Rq :: (B, H, Q), S \rangle}$$

Para reducir un proceso tell $(!\phi.\vec{P})$, la restricción $\phi(\tilde{L})$ es puesta en el *store*, los procesos suspendidos de las colas Aq y Msq pasan a la cola de ejecución Rq para intentar ser reducidos nuevamente y el hilo $\vec{P}$ continua con su ejecución:

$$\text{TELL} : \frac{\forall L_i \in \widetilde{L}\ L_i \in Dom(B)}{\langle !\phi.\vec{P}, B, \phi(\widetilde{L}), Oq, Msq, Aq, Rq, S\rangle \longrightarrow \langle \vec{P}, B, \varnothing, Oq, \bullet, \bullet, Rq :: Msq :: Aq, S \wedge \phi(\widetilde{L})\rangle}$$

Para reducir un proceso *ask* $(?\phi.\vec{P})$, se requiere del sistema de restricciones. Hay tres posibles casos:

1. Si el *store* deduce $\phi(\widetilde{L})$ la máquina continúa con la ejecución del hilo $\vec{P}$:

$$\text{ASK1} : \frac{S \models_{\Delta} \phi(\widetilde{L}) \quad \forall L_i \in \widetilde{L}\ L_i \in Dom(B)}{\langle ?\phi.\vec{P}, B, \phi(\widetilde{L}), Oq, Msq, Aq, Rq, S\rangle \longrightarrow \langle \vec{P}, B, \varnothing, Oq, Msq, Aq, Rq, S\rangle}$$

2. Si el *store* deduce $\neg\phi(\widetilde{L})$ la máquina elimina el proceso actual:

$$\text{ASK2} : \frac{S \models_{\Delta} \neg\phi(\widetilde{L}) \quad \forall L_i \in \widetilde{L}\ L_i \in Dom(B)}{\langle ?\phi.\vec{P}, B, \phi(\widetilde{L}), Oq, Msq, Aq, Rq, S\rangle \longrightarrow \langle nil, B, \phi(\widetilde{L}), Oq, Msq, Aq, Rq, S\rangle}$$

3. Si el *store* no puede deducir $\phi(\widetilde{L})$ ni tampoco $\neg\phi(\widetilde{L})$, el proceso es suspendido pasándolo a la cola Aq:

$$\text{ASK3} : \frac{S \not\models_{\Delta} \phi(\widetilde{L}) \quad S \not\models_{\Delta} \neg\phi(\widetilde{L}) \quad \forall L_i \in \widetilde{L}\ L_i \in Dom(B)}{\langle ?\phi.\vec{P}, B, \phi(\widetilde{L}), Oq, Msq, Aq, Rq, S\rangle \longrightarrow \langle nil, B, \phi(\widetilde{L}), Oq, Msq, Aq :: (B, \phi(\widetilde{L}), ?\phi.\vec{P}), Rq, S\rangle}$$

Como se puede ver en la sintaxis (Tabla 4.1) hay la posibilidad de que los objetos o los mensajes estén replicados. Por abreviación en las siguientes reglas sólo se tomarán objetos y mensajes sin replicación. La replicación asegura la persistencia de ellos, razón por la cual siempre serán llevados a las colas Oq o Msq, respectivamente. Si el proceso por reducir está replicado, la máquina intentará comunicar dicho proceso con todos elementos en la cola Msq para objetos y Oq para mensajes antes de ser encolados.

Si se intenta reducir un paso de mensaje y el objeto destino no se encuentra en la cola Oq, este mensaje será suspendido en la cola Msq:

$$\text{MsgEnQ:} \frac{S \models_{\Delta} I = I' \quad (I', J') \rhd M \notin Oq}{\langle I \lhd l_i : [\widetilde{K}].\vec{P}, B, \widetilde{x_i} \mapsto \widetilde{L}, Oq, Msq, Aq, Rq, S\rangle \longrightarrow \langle nil, B, \widetilde{x_i} \mapsto \widetilde{L}, Oq, Msq :: (B, \widetilde{x_i} \mapsto \widetilde{L}, I \lhd l_i : [\widetilde{K}].\vec{P}), Aq, Rq, S\rangle}$$

Si el proceso es un mensaje $I \lhd l_i : [\widetilde{K}].\vec{P}$ a un objeto que se encuentra en la cola Oq y la etiqueta del mensaje l_i coincide con una etiqueta en el conjunto de métodos del objeto, el objeto es eliminado de la cola de objetos y el cuerpo del método $\vec{P}_i$ con el nuevo enlace para $\widetilde{K}$ es puesto al final de la cola de ejecución Rq para ser ejecutado luego; la ejecución continua con $\vec{P}$:

$$\text{MsgComm:}\frac{S \models_\Delta I=I' \quad o \leq i \leq m \quad |\widetilde{K}|=|\widetilde{x_i}| \quad (\widetilde{K} \mapsto \widetilde{L}) \in B}{\left\langle I \lhd l_i:[\widetilde{K}].\vec{P},B,\widetilde{x_i} \mapsto \widetilde{L},Oq::(B',\emptyset,(I',J') \rhd [l_1:(\widetilde{x_1})\vec{P}_1,\ldots,l_m:(\widetilde{x_n})\vec{P_m}]),Msq,Aq,Rq,S\right\rangle \longrightarrow}$$
$$\left\langle \vec{P},B,\emptyset,Oq,Msq,Aq,Rq::(B'+(\widetilde{x_i} \mapsto \widetilde{L}),\emptyset,\vec{P}_i),S\right\rangle$$

Si el objeto $(I', J') \rhd [l_1 : (\widetilde{x_1})\vec{P}_1, \ldots, l_m : (\widetilde{x_n})\vec{P_m}]$ existe en la cola Oq pero no hay una etiqueta l_i para comunicarse con $I \lhd l_i : [\widetilde{K}].\vec{P}$ y el objeto tiene una dirección de delegación J' con $(S \models_\Delta I' \neq J')$, el receptor del mensaje se cambia por la dirección de delegación. e.d el proceso es cambiado por $J' \lhd l_i : [\widetilde{K}].\vec{P}$ y el objeto se deja en la cola Oq:

$$\text{MsgDel:}\ \frac{S \models_\Delta I=I' \quad S \models_\Delta I' \neq J' \quad i \geq m \quad (\widetilde{K} \mapsto \widetilde{L}) \in B}{\left\langle I \lhd l_i:[\widetilde{K}].\vec{P},B,\widetilde{x_i} \mapsto \widetilde{L},Oq::(B',\emptyset,(I',J') \rhd [l_1:(\widetilde{x_1})\vec{P}_1,\ldots,l_m:(\widetilde{x_n})\vec{P_m}]),Msq,Aq,Rq,S\right\rangle \longrightarrow}$$
$$\left\langle nil,B,\widetilde{x_i} \mapsto \widetilde{L},Oq::(B',\emptyset,(I',J') \rhd [l_1:(\widetilde{x_1})\vec{P}_1,\ldots,l_m:(\widetilde{x_n})\vec{P_m}]),Msq,Aq,Rq::(B,(\widetilde{x_i} \mapsto \widetilde{L}),J' \lhd l_i:[\widetilde{K}].\vec{P}),S\right\rangle$$

De igual manera si el objeto $(I', J') \rhd [l_1 : (\widetilde{x_1})\vec{P}_1, \ldots, l_m : (\widetilde{x_n})\vec{P_m}]$ existe en Oq y no hay una etiqueta l_i para comunicarse con el mensaje $I \lhd l_i : [\widetilde{K}].\vec{P}$ pero el objeto no tiene una dirección de delegación $(S \models_\Delta I' = J')$, este mensaje será suspendido en la cola Msq:

$$\text{MsgWODel:}\ \frac{S \models_\Delta I=I' \quad S \models_\Delta I'=J' \quad i \geq m \quad (\widetilde{K} \mapsto \widetilde{L}) \in B}{\left\langle I \lhd l_i:[\widetilde{K}].\vec{P},B,\widetilde{x_i} \mapsto \widetilde{L},Oq::(B',\emptyset,(I',J') \rhd [l_1:(\widetilde{x_1})\vec{P}_1,\ldots,l_m:(\widetilde{x_n})\vec{P_m}]),Msq,Aq,Rq,S\right\rangle \longrightarrow}$$
$$\left\langle nil,B,\widetilde{x_i} \mapsto \widetilde{L},Oq::(B',\emptyset,(I',J') \rhd [l_1:(\widetilde{x_1})\vec{P}_1,\ldots,l_m:(\widetilde{x_n})\vec{P_m}]),Msq::(B,\widetilde{x_i} \mapsto \widetilde{L},I \lhd l_i:[\widetilde{K}].\vec{P}),Aq,Rq,S\right\rangle$$

En esta regla no hay ningún mensaje en la cola de mensajes Msq para comunicarse con $(I, J) \rhd [l_1 : (\widetilde{x_1})\vec{P}_1, \ldots, l_m : (\widetilde{x_n})\vec{P_m}]$, por tanto el objeto es pasado a la cola de objetos Oq para su posterior reducción:

$$\text{ObjEnQ}:\ \frac{S \models_\Delta I=I' \quad I' \lhd m.\vec{P} \notin Msq}{\left\langle (I,J) \rhd [l_1:(\widetilde{x_1})\vec{P}_1,\ldots,l_m:(\widetilde{x_n})\vec{P_m}],B,\emptyset,Oq,Msq,Aq,Rq,S\right\rangle \longrightarrow}$$
$$\left\langle nil,B,\emptyset,Oq::(B,\emptyset,(I,J) \rhd [l_1:(\widetilde{x_1})\vec{P}_1,\ldots,l_m:(\widetilde{x_n})\vec{P_m}]),Msq,Aq,Rq,S\right\rangle$$

En este caso, hay un mensaje $I' \lhd l_i : [\widetilde{K}].\vec{P}$ en la cola de mensajes Msq para comunicarse con el objeto $(I,J) \rhd [l_1 : (\widetilde{x_1})\vec{P}_1, \ldots, l_m : (\widetilde{x_n})\vec{P_m}]$; adicionalmente, hay una etiqueta l_i en la lista de métodos del objeto; el mensaje es eliminado de la cola de mensajes Msq, y la continuación del mensaje $\vec{P}$ se pone al final de la cola de ejecución Rq, seguido por el cuerpo del método $\vec{P}_i$ con el nuevo enlace para $\widetilde{K}$:

$$\text{ObjComm} : \frac{S \models_\Delta I = I' \quad 0 \leq i \leq m \quad |\widetilde{K}| = |\widetilde{x_i}| \quad (\widetilde{K} \mapsto \widetilde{L}) \in B'}{\left\langle (I,J) \rhd [l_1 : (\widetilde{x_1})\vec{P}_1, \ldots, l_m : (\widetilde{x_n})\vec{P_m}], B, \emptyset, Oq, Msq :: (B', (\widetilde{x_i} \mapsto \widetilde{L}), I' \lhd l_i : [\widetilde{K}].\vec{P}), Aq, Rq, S \right\rangle \longrightarrow}$$
$$\left\langle nil, B, \emptyset, Oq, Msq, Aq, Rq :: (B', \emptyset, \vec{P}) :: (B + (\widetilde{x_i} \mapsto \widetilde{L}), \emptyset, \vec{P}_i), S \right\rangle$$

En la siguiente regla hay un mensaje $I' \lhd l_i : [\widetilde{K}].\vec{P}$ en la cola de mensajes Msq que intenta comunicarse con el objeto $(I,J) \rhd [l_1 : (\widetilde{x_1})\vec{P}_1, \ldots, l_m : (\widetilde{x_n})\vec{P_m}]$ con delegación ($S \models_\Delta I \neq J$), pero no hay una etiqueta l_i en la lista de métodos del objeto; el mensaje es entonces pasado de la cola de mensajes Msq, al final de la cola de ejecución Rq, pero cambiándole el objeto receptor por la dirección de delegación, es decir por el proceso $J \lhd l_i : [\widetilde{K}].\vec{P}$. El objeto original se pasa a la cola Oq:

$$\text{ObjDel:} \frac{S \models_\Delta I = I' \quad S \models_\Delta I \neq J \quad i \geq m \quad (\widetilde{K} \mapsto \widetilde{L}) \in B'}{\left\langle (I,J) \rhd [l_1 : (\widetilde{x_1})\vec{P}_1, \ldots, l_m : (\widetilde{x_n})\vec{P_m}], B, \emptyset, Oq, Msq :: (B', (\widetilde{x_i} \mapsto \widetilde{L}), I' \lhd l_i : [\widetilde{K}].\vec{P}), Aq, Rq, S \right\rangle \longrightarrow}$$
$$\left\langle nil, B, \emptyset, Oq :: (B, \emptyset, (I,J) \rhd [l_1 : (\widetilde{x_1})\vec{P}_1, \ldots, l_m : (\widetilde{x_n})\vec{P_m}]), Msq, Aq, Rq :: (B', \widetilde{x_i} \mapsto \widetilde{L}, J \lhd l_i : [\widetilde{K}].\vec{P}), S \right\rangle$$

Por ultimo, si hay un mensaje $I' \lhd l_i : [\widetilde{K}].\vec{P}$ en la cola de mensajes Msq para el objeto $(I,J) \rhd [l_1 : (\widetilde{x_1})\vec{P}_1, \ldots, l_m : (\widetilde{x_n})\vec{P_m}]$ sin delegación ($S \models_\Delta I = J$), pero no hay una etiqueta l_i en la lista de métodos de él, el objeto se pasa a la cola Oq:

$$\text{ObjWODel:} \frac{S \models_\Delta I = I' \quad S \models_\Delta I = J \quad i \geq m \quad (\widetilde{K} \mapsto \widetilde{L}) \in B'}{\left\langle (I,J) \rhd [l_1 : (\widetilde{x_1})\vec{P}_1, \ldots, l_m : (\widetilde{x_n})\vec{P_m}], B, \emptyset, Oq, Msq :: (B', (\widetilde{x_i} \mapsto \widetilde{L}), I' \lhd l_i : [\widetilde{K}].\vec{P}), Aq, Rq, S \right\rangle \longrightarrow}$$
$$\left\langle nil, B, \emptyset, Oq :: (B, \emptyset, (I,J) \rhd [l_1 : (\widetilde{x_1})\vec{P}_1, \ldots, l_m : (\widetilde{x_n})\vec{P_m}]), Msq :: (B', (\widetilde{x_i} \mapsto \widetilde{L}), I' \lhd l_i : [\widetilde{K}].\vec{P}), Aq, Rq, S \right\rangle$$

4.10 ESTADO INICIAL Y FINAL Y DIAGRAMA DE TRANSICION DE LA MAQUINA

La máquina abstracta comienza en un estado computacional donde las colas de objetos, mensajes, ask y ejecución están vacías y no hay ligaduras en HB y HA. Además el Store es vacío. El estado inicial es:

$$\langle \vec{P}, \oslash, \oslash, \bullet, \bullet, \bullet, \bullet, T \rangle$$

Donde $\vec{P}$ es el hilo inicial. Véase la sección 4.5.

El estado final de la máquina debe ser:

$$\langle nil, B, H, Oq, Msq, Aq, \bullet, S \rangle$$

Es decir, se llega al estado final sólo cuando no hay nada más que planificar en la cola de ejecución.

La máquina sabe que los procesos en Aq, Msq y Oq están suspendidos y no pueden ser reducidos (ver la sección 4.9).

A medida que se ejecuta un hilo, el proceso cambia de estado. El estado de un proceso se define por su actividad actual. De esta manera, un proceso puede estar en uno de los siguientes estados: ejecución, listo, suspendido o finalizado. Estos estados se ilustran en la figura 4.5. Los procesos suspendidos pueden ser objetos, mensajes y *ask*. Cuando se ejecuta un "*tell*", todos los procesos mensaje y los procesos "*ask*" que estaban suspendidos pasan a la cola de listos para intentar reducirse. Un proceso mensaje también pueden pasar a "listo" cuando un objeto se comunica con él. Un proceso objeto suspendido pasa a "listo" sólo cuando hay un mensaje con el que puedan ser comunicados; a diferencia de los otros dos tipos de procesos, todos los objetos suspendidos no pasan a listos en un solo paso.

4.11 INSTRUCCIONES DE LA MAQUINA Y FORMATO DEL CODIGO DE BYTES

4.11.1 Formato de las Instrucciones de Máquina

Las instrucciones están representadas por un código de operación y de 0 a 3 atributos. Las instrucciones pueden ser clasificadas en 3 grupos: el primero para la manipulación de procesos (tabla 4.2), el segundo para la definición de objetos (tabla 4.3) y el último para la construcción de predicados de primer orden (tabla 4.5).

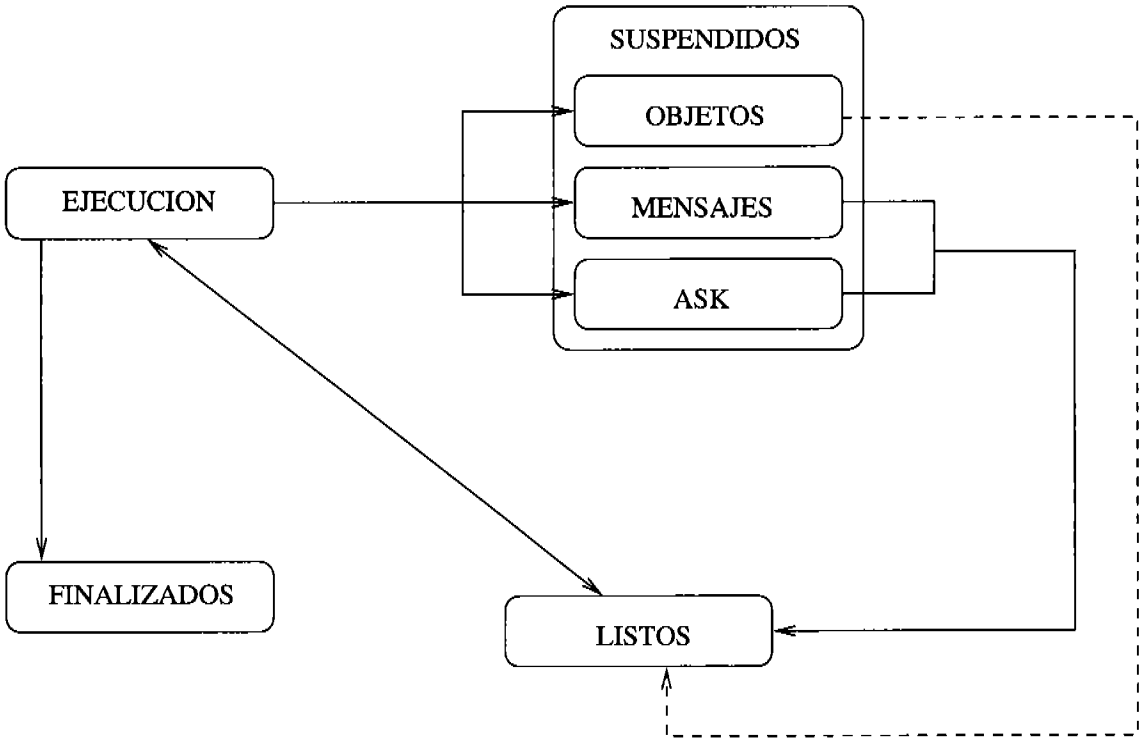

Figura 4.5: Diagrama de Transición de Estados de MAPiCO

En la tabla 4.2 y la tabla 4.3:

- **Opcode** Código de operación de la instrucción, 8 bits.
- **dir** es una dirección valida en la memoria del programa, 32 bits.
- **ind** es un índice dentro de alguna de las listas (apuntadas por PV, PN), 16 bits.
- **num** es un número constante. Para objetos especifica el número de métodos; para métodos y mensajes especifica su nombre o referencia, 16 bits.
- Luego de una instrucción *replicate* siempre debe ir un *call.*
- Antes de hacer un *call* o un *replicate,* el usuario siempre debe especificar el objeto receptor con *pushn* o *pushv.*

Instrucción	Opcode	Src1	Descripción
ret	0		Es el proceso NIL en el cálculo.
par	1	dir	Crea un nuevo proceso apuntado por Src1 para ser ejecutado en paralelo.
newv	2		Definición de una variable.
newn	3		Definición de un nombre.
pushv	4	ind	Pone variables en el PA para ser pasadas como argumentos en comunicación de objetos.
pushn	5	ind	Pone nombres en el PA para ser pasadas como argumentos en comunicación de objetos.
replicate	7		Cuando está antes de una instrucción call, replica este mensaje.
pop	18		Saca el primer elemento de la pila apuntada por PA para que pase a ser una variable o nombre según el caso.
call	20	num	Hace el llamado al método Src1 en un objeto igual (en el *Store*) a la variable o nombre en la cabeza del PA.
tell	24		Adiciona al *store* la restricción apuntada por PA.
ask	25		Realiza la operación de ask al store de una restricción apuntada por PA.

Tabla 4.2: Instrucciones para la Manipulación de Procesos en la Máquina

4.11.2 Definición de Restricciones

En el cálculo PiCO las restricciones se representan con el símbolo ϕ que denota un predicado de primer orden.

Instrucción	Opcode	Src1	Src2	Src3	Descripción
objvv	64	ind	ind	num	Objeto con nombre Src1, delegación Src2 y con src3 métodos. (Src1 y Src2 son índices dentro del PV)
objnn	65	ind	ind	num	Igual a objvv pero Src1 y Src2 son nombres dentro del PN
objvn	66	ind	ind	num	Igual a objvv pero Src1 es una variable dentro del PV y Src2 es un nombre dentro del PN
objnv	67	ind	ind	num	Igual a objvv pero Src1 es un nombre dentro del PA y Src2 es una variable dentro del PN
objvvr	72	ind	ind	num	Igual a objvv pero replicado
objnnr	73	ind	ind	num	Igual a objnn pero replicado
objvnr	74	ind	ind	num	Igual a objvn pero replicado
objnvr	75	ind	ind	num	Igual a objnv pero replicado
meth	80	num	dir		Especifica que el método Src1 está en la dirección de control Src2.

Tabla 4.3: Instrucciones para la Definición de Objetos en la Máquina

Los predicados de primer orden [McA92] están definidos bajo la sintaxis de la tabla 4.4. Esos predicados de primer orden se modelan en la máquina con las instrucciones de la tabla 4.5.

Para lograr que el sistema de restricciones se adicione de una manera ortogonal con respecto a la máquina, las restricciones deben ser predicados de primer orden que siguen las reglas sintácticas de la tabla 4.4. Así, cualquier sistema de restricciones que pueda expresar sus restricciones como fórmulas de primer orden, puede integrarse a la máquina.

Las instrucciones de predicados de primer orden están presentadas en la tabla 4.5, en donde:

<Sentencia>	→	<Sentencia atómica>
	|	<Sentencia> <Conector> <Sentencia>
	|	<Cuantificador> <Variable>,... <Sentencia>
	|	¬ <Sentencia>
	|	(<Sentencia>)
<Sentencia atómica>	→	Predicado(<Término>,...)
	|	<Término> = <Término>
<Término>	→	Función(<Término>,...)
	|	<Constante>
	|	<Variable>
<Conector>	→	$\wedge \mid \vee \mid \Leftrightarrow \mid \Rightarrow$
<Cuantificador>	→	$\forall \mid \exists$
<Constante>	→	A | X1 | Juan | ...
<Variable>	→	a | x | s | ...
<Predicado>	→	Antes | En | ...
<Función>	→	máximo | mínimo | ...

Tabla 4.4: BNF para la Especificación de Predicados de Primer Orden

- **Opcode** Código de operación de la instrucción, 8 bits.
- **ind** Indice dentro de alguna de las listas (apuntadas por PV, PN), 16 bits.
- **num** Número constante entero, 16 bits.
- **fun** Código de la función, 8 bits. Depende del sistema de restricciones.
- **pred** Código del predicado del átomo, 8 bits.
- **ari** Aridad de la función o el átomo, 8 bits.
- **con** Código del conector, las opciones son *and* (0), *or* (1), 8 bits.
- **cuan** Código del cuantificador o negación, 8 bits. $\exists$ (1) $\forall$ (2), *not* (0).

Ejemplo 4.11.1 *Para ilustrar mejor el uso y función de las instrucciones, se mostrará la codificación del factorial de un número en el cálculo y en la máquina.*

fac(x,r)	::=	*r=1*	*si x = 0*
	|	*r = fac(x-1)*x*	*si x > 0*

fac(3,r)

Instrucción	Opcode	Src1	Src2	Descripción
termc	32	num		Término constante dentro de una fórmula de primer orden.
termv	33	ind		Término variable dentro de una fórmula de primer orden. Src1 es la posición dentro de PV.
termn	34	ind		Término nombre dentro de una fórmula de primer orden. Src1 es la posición dentro de PN.
termf	35	fun	ari	Función. Src1 es el código de la función y Src2 es la aridad.
atom	36	pred	ari	Atomo. Src1 es el código del átomo y Src2 es la aridad.
sentenc	40	con		Sentencia con un conector. Src1 es el código del conector de aridad 2.
sentenq	41	cuan		Sentencia con un Cuantificador o negación. Src1 es el código del cuantificador o de la negación.

Tabla 4.5: Instrucciones para Construir Predicados de Primer Orden en la Máquina

En el cálculo PiCO, este programa podría ser codificado como:

$$(\upsilon\ x\ y)(\upsilon\ fac)((*fac \triangleright [input : (n,r)(?(n=0).!(r=1)$$
$$|?(n>0).(\upsilon\ y\ a)((!(y=n-1)$$
$$|(fac \triangleleft input[y,a]))$$
$$|!(r=a*n))))])$$
$$|!(x=3).fac \triangleleft input[x,y])$$

En el lenguaje de máquina, este programa queda codificado de la siguiente manera:

```
0    newv                          ; x
1    newv                          ; y
2    newn                          ; fac
3    par        45                 ; en paralelo con 45
4    objnnr     0      0    1      ; fac, No delega, 1 método
```

```
5    meth      1      6
6    pop                     ; n
7    pop                     ; r
8    par       18
9    atom      eq     2      ; =
10   termv     1              ; n
11   termc     0              ; 0
12   ask
13   atom      eq     2      ; =
14   termv     0              ; r
15   termc     1              ; 1
16   tell
17   ret
18   atom      great  2      ; >
19   termv     1              ; n
20   termc     0              ; 0
21   ask
22   newv                     ; y
23   newv                     ; a
24   par       32
25   atom      eq     2      ; =
26   termv     1              ; y
27   termf     less   2      ; -
28   termv     3              ; n
29   termc     1              ; 1
30   tell
31   ret
32   par       38
33   pushv     0              ; a
34   pushv     1              ; y
35   pushn     0              ; fac
36   call      1              ; input
37   ret
38   atom      eq     2      ; =
39   termv     2              ; r
40   termf     mult   2      ; *
41   termv     0              ; a
42   termv     3              ; n
43   tell
44   ret
45   atom      eq     2      ; =
```

```
46    termv       1               ; x
47    termc       3               ; 3
48    tell
49    pushv       0               ; y
50    pushv       1               ; x
51    pushn       0               ; fac
52    call        1               ; input
53    ret
```

Las operaciones eq, great, less, mult, etc. representan los códigos de las funciones definidas por el sistema de restricciones para hacer igualación, comparación y operaciones matemáticas.

El número de línea y los comentarios (al lado derecho del punto y coma) no son parte de la entrada de la máquina; adicionalmente, en las instrucciones par *y* meth *hay una referencia al número de línea (como una indirección) pero se necesita especificar una dirección del programa y no un número de línea; así como en la instrucción se debe poner el opcode. Se presenta de esta manera por claridad.*

Ver otros ejemplos en los anexos.

4.12 FUNCIONAMIENTO INTERNO DE LA MAQUINA

4.12.1 Instrucciones

A continuación se muestra el comportamiento de cada instrucción a través de su macroalgoritmo. Las siguientes funciones son útiles para la exposición:

new(): Reserva memoria de traducción en una estructura de Traducción.

newarg(): Reserva memoria de traducción en una estructura de argumento.

sig(Pointer, Integer): Devuelve el contenido de la estructura que está en la posición "Integer" de la lista apuntada por "Pointer".

newVar(): Devuelve un identificador de variable libre dentro del *store*.

newName(): Devuelve un identificador de nombre libre dentro del *store*.

ask(Pointer): Función *ask* del sistema de restricciones.

tell(pointer): La función *tell* del sistema de restriciones para adicionar restriciones en él.

- **newv:**
  PAUX = new()
  PAUX.value = newVar()
  PAUX.sig = PV
  PV = PAUX

- **newn:**
  PAUX = new()
  PAUX.value = newName()
  PAUX.sig = PN
  PN = PAUX

- **par:** Src1
  Crea un nuevo proceso con:
  PC = Src1
  PV = PVA
  PN = PNA
  PA = PAA
  y lo adiciona al final de la RunQ.

- **pushv:** Src1
  PAUX = newarg()
  PAUX.sig = PA
  PAUX.value = sig(PV,Src1)
  PAUX.class = VARIABLE
  PA = PAUX

- **pushn:** Src1
  PAUX = newarg()
  PAUX.sig = PA
  PAUX.value = sig(PN,Src1)
  PAUX.class = NAME
  PA = PAUX

- **pop:**
  PAUX = PA
  PA = PA.sig
  SI PAUX.class == VARIABLE
  – – PAUX.sig = PV
  – – PV = PAUX
  SINO
  – – PAUX.sig = PN
  – – PN = PAUX
  FINSI

- **objvv:** Src1 Src2 Src3
  SI hay un mensaje M para la variable Src1
  – – **SI** M está en los métodos de Src1
  – – – – inserta el método en la cola RunQ.
  – – – – inserta la continuación de M en RunQ.
  – – **SINO**
  – – – – inserta al final de RunQ un mesaje sin replicación a la variable Src2.
  – – – – inserta el proceso en ObjQ.
  – – **FINSI**
  – – **SI** M está replicado
  – – – – inserta M en la cola de mensajes MsgQ.
  – – **FINSI**
  SINO
  – – inserta el proceso en ObjQ.
  FINSI

- para las instrucciones **objnn, objvn, objnv, objvvr, objnnr, objvnr** y

objnvr el macroalgoritmo es muy similar al de objvv. En las instrucciones **objvvr, objnnr, objvnr** y **objnvr** el proceso siempre se inserta en la cola Oq.

- **meth:** Src1 Src2
  Estpecifica que el método Src1 está en la dirección de control Src2. Esta instrucción no cambia el estado de la máquina.

- **ret:**
  Saca el siguiente proceso de la pila Rq y lo ejecuta, es decir lo pone en los registros actuales de la máquina.

- **replicate**
  Sirve de marca para replicar los mensajes (instrucciones *call*). Crea un proceso con el *PC* apuntando a la siguiente intrucción para intentar comunicarlo con los objetos en Oq; si puede comunicarlo, crea otro y continua intentando hasta que no puede comunicarse más y el proceso creado es eliminado. Este proceso (*replicate*) es insertado en la cola de mensajes Msq. Se debe tener en cuenta que esta instrucción puede generar procesos innecesarios en las pilas al hacer una delegación de mensajes, razón por la cual por optimización se debería idear un mecanismo para recolección de basura que queda a consideración de la implementación.

- **call:** Src1
  SI está PA.value y PA.class (nombre de objeto destino) en ObjQ
  – – **SI** está Src1 en métodos del objeto destino
  – – – – crea un proceso con:
  – – – – PC = (apuntador del método Src1)
  – – – – PV = (proceso del objeto destino).PV
  – – – – PN = (proceso del objeto destino).PN
  – – – – PAA = PAA.sig
  – – – – PA = PAA
  – – – – PAA = nil
  – – – – y lo pone al final de la cola RunQ
  – – – – **SI** el objeto destino está replicado

– – – – – – lo deja en la cola ObjQ
– – – – **SINO**
– – – – – – lo saca de esta.
– – – – **FINSI**
– – **SINO**
– – – – **SI** el objeto destino tiene delegación
– – – – – – PA.value = (objeto destino).delegación
– – – – – – PA.class = (objeto destino).delegación.class
– – – – – – salta a línea 1
– – – – **SINO**
– – – – – – Inserta el proceso en ejecución en la cola MsgQ.
– – – – **FINSI**
– – **FINSI**
SINO
– – Inserta el proceso en ejecución en la cola MsgQ.
FINSI

- **ask:**
  SI ask(PA)
  – – PC++
  SINO
  – – **SI** not ask(PA)
  – – – –destruye el proceso y continua con el siguiente.
  – – **SINO**
  – – – – inserta el proceso en la cola AskQ
  – – **FINSI**
  FINSI

- **tell:**
  tell(PA)

4.12.2 Manejo Interno del PV, PN y PA en la Memoria de Traducción

Como ya se dijo en la sección 4.5, los apuntadores PV y PN apuntan a una posición en la memoria de traducción donde hay una estructura con la ligadura de las variables y los nombres, respectivamente (véase la sección 4.8), en un determinado ambiente. Similarmente, el puntero PA puede referenciar argumentos o definiciones de restricciones en la misma memoria.

La memoria de traducción puede contener tres tipos de estructuras de datos. La primera sirve para convertir variables y nombres en sus ligaduras y puede ser apuntado por PV y PN; la segunda, para pasar argumentos y la tercera, para pasar restricciones al sistema de restricciones. Los dos ultimos son apuntados por el PA.

El primer tipo de estructura de datos es un árbol ascendente, e.d. que los nodos apuntan a sus padres. Los nodos pueden ser variables (para la estructura apuntada por PV) o nombres (para la estructura apuntada por PN). La estructura de traducción de variables a sus ligaduras está representada por uno o varios de esos árboles; para los nombres existen otros árboles. Cada proceso tiene acceso a su propia estructura de traducción de variables (nombres) por el puntero PV (PN) en la tupla que lo representa.

El PV (PN) apunta a un nodo del árbol de ligaduras de variables (nombres) y cada uno de los nodos tiene una referencia a su padre. Una rama (secuencia de nodos desde el nodo apuntado por PV (PN) en un proceso dado hasta la raiz del árbol) representa las ligaduras de las variables (nombres) de un proceso (ver la figura 4.2).

En la figura 4.6 se puede ver un ejemplo de cómo quedaría la estructura de datos para la ligadura de las variables con 4 procesos. En esta figura el proceso 1 tiene acceso a las variables v2 y v1, los procesos 2 y 3 tienen acceso a las variables v4, v3 y v1 y el proceso 4 tiene acceso a v3 y v1.

Cuando una nueva variable (nombre) es creada en un proceso dado, un nuevo nodo se adiciona al árbol de ligaduras de variables (nombres), especificamente a la rama apuntada por el PV (PN) del proceso; entonces el PV (PN) se actualiza para apuntar al nodo adicionado.

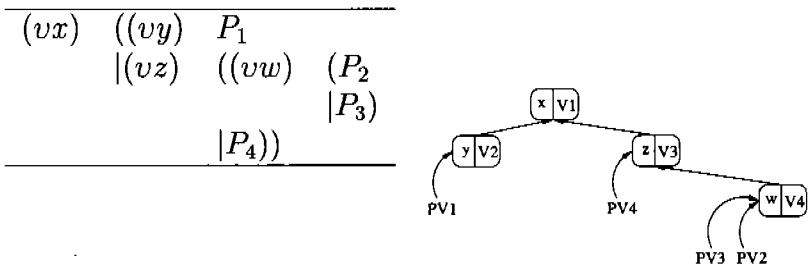

Figura 4.6: Ligadura de Variables de los Procesos de un programa

La rama es ramificada (se abre en dos) cuando una composición paralela es reducida, dado que en su reducción cada uno de los procesos involucrados debe tomar los punteros PV y PN de su padre para así tener una referencia propia.

El segundo tipo de estructura de datos, que puede ser apuntado por el PA, es una lista simplemente encadenada de parámetros, usada por la instrucción *call* o *replicate* (ver la figura 4.3), mientras que la tercera estructura, que tambien puede ser apuntada por el PA, es un árbol n-ario que puede representar una restricción que se envía posteriormente al sistema de restricciones (ver la figura 4.4).

5 IMPLEMENTACION DE LA MAQUINA ABSTRACTA

La estrategia de implementación de MAPiCO se diferencia de la usada en [Lop98] y [PT96], porque la máquina abstracta de TyCO usa una memoria lineal para guardar la pila o heap y los datos volátiles además de un word-code (*w*-code) para codificar las instrucciones básicas. PICT por otra parte, traslada el código fuente a lenguaje C y compila el resultado. Por su parte, MAPiCO ejecuta las instrucciones siguiendo el funcionamiento descrito en el capítulo 4.12.

Para implementar la máquina abstracta, se utilizó el lenguaje de programación JAVA. Se definieron varias clases, usadas para manejar el área de memoria, el manejador de procesos y las estructuras de datos para guardar la información requerida durante la ejecución de la máquina, de acuerdo a las estructuras planteadas en el diseño. Adicionalmente, se ha usado el precompilador de C (cpp) para hacer de esta una implementación más flexible ya que da la posibilidad de utilizar macros y directivas de precompilación.

5.1 ESTRUCTURAS DE DATOS

En MAPiCO están definidas algunas áreas de memoria para guardar referencia a los procesos suspendidos (Cola de objetos, mensajes y ask), una cola de ejecución, un sistema de restricciones y una memoria de programa. Para implementar las colas, se definió la clase `List`.

Esta lista tiene una referencia al primer y ultimo nodo de la lista, y el número de

elementos en la lista.

```
public class List
{
    /** Size of the list */
    private int Elements;

    /** Reference to head and tail of the list */
    private NodeList HeadList,EndList;
    ...
}
```

La clase `List` provee las siguientes funciones:

```
/* Get the length of the list */
  public int length()

/* Insert a data at list head */
  public void insertAtHead( Object data )

/* Insert a data into the end */
  public void insertAtEnd( Object data )

/* Get the head of the list */
  public NodeList FirstNode()

/* Get the last element */
  public NodeList EndNode()

/* Insert a data at given index */
  public void insertAt(int index,Object data)

/* Delete a data at given index */
  public void delAt(int index)

/* Get a data at given index */
  public Object getAt( int index )
```

La clase `NodeList` define un nodo de la lista. Estos nodos pueden ser de cualquier clase porque esta clase tiene una referencia a un elemento de clase Object de java

(La raíz de todo el árbol de jerarquías de clases de Java). La clase `NodeList` tiene una referencia al siguiente nodo y una referencia al elemento de la lista.

```
public class NodeList
{
  /** Next node reference */
  private NodeList NextNode;

  /** Node data */
  private Object Data;
  ...
}
```

Las funciones dadas por la clase `NodeList` son:

```
/* Set the next node */
  public void setNext(NodeList Continuation)

/* Set the data in the node */
  public void setData(Object DataIn)

/* Get the next node */
  public NodeList getNext()

/* Get the data node */
  public Object getData()

/* Get a data node at given index */
  public Object getDataAt(int index)

/* Gets the length of the list */
  public int length()
```

El almacenamiento del programa está en una estructura de datos, la cual está definida por un arreglo de bytes para guardar el bytecode que es cargado de un archivo binario. Este archivo contiene el código de las instrucciones de máquina que especifican un programa. La clase `Loader` es usada para cargar este archivo en la memoria de programa, la cual está también definida en esta clase.

```
public class Loader
{

  byte[] Data;     // Program on machine code storage
  int Size;        // Size of program
  ...
}
```

Para manejar la memoria de programa (en la clase Loader) están las funciones para cargar, sacar bytes, words o double words de la memoria.

```
  Loader(String FileName)      // Load a program into the array
  byte get8(int Pointer)       // Get a byte
  short get16(int Pointer)     // Get a word
  int get32(int Pointer)       // Get a double word
```

La estructura de datos para traducir la ligadura de las variables y de los nombres es un árbol (ver las secciones 4.8 y 4.12.2), la estructura para pasar parámetros en el paso de mensajes es similar a la estructura de ligaduras pero tiene un campo adicional: su tipo (variable o nombre).

```
public class NodeList
{
    private NodeList NextNode;   // Next node reference
    private Object Data;         // Node data
    ...
}


public class NodeArg
{
    private NodeArg NextNode;   // Next node reference
    private Object Data;        // Node data
    private byte type;          // Argument type
    ...
}
```

Adicionalmente, la estructura de datos usada para implementar una restricción es un árbol, donde sus nodos pueden ser términos, átomos o sentencias (ver la sección 4.11.2).

```
public class FrameConst
{
    private FrameConst padre;     // Reference to the father frame
    private int type;             // Type of frame
    private long valor;           // Value of frame
    private byte numhijos;        // Number of children
    private FrameConst hijos[];   // Children references
    ...
}
```

5.2 ESTRUCTURA DE UN PROCESO

El proceso está representado como una cuádrupla donde está la información necesaria para ejecutarlo. Estas variables son el puntero a código, puntero a variables, puntero a nombres y un puntero auxiliar usado para pasar parámetros y en la definición de restricciones. (ver la sección 4.5)

Todas esas variables son atributos de la clase `Process`, donde hay funciones para obtener cada uno de los atributos de dicha clase.

```
public class Process
{
  /** Program counter */
  int  PC;

  /** Variables area pointer */
  NodeList PV;

  /** Names area pointer */
  NodeList PN;

  /** Actual arguments area or constraint definition pointer */
  Object PA;
```

```
  ...
}
```

5.3 BLOQUE PRINCIPAL DE LA MAQUINA

Todos los componentes de la máquina abstracta están agrupados en la clase `Executor`. En esta clase están definidos los registros de la máquina, las colas de procesos suspendidos, la cola de ejecución, el sistema de restricciones la memoria de traducción y la memoria del programa (ver la sección 4.3).

```
public class Executor
{
    Store store;

    /** Execution Queue */
    List RunQ;

    /** Object Queue */
    List ObjQ;

    /** Message Queue */
    List MsgQ;

    /** Ask Queue */
    List AskQ;

    /** Program area */
    Loader Program;

    /** Actual process execute registers */
    int        PCA;
    NodeList   PVA;
    NodeList   PNA;
    Object     PAA;

    /** Other auxiliary pointer */
    Object     PAUX;
```

```
    /** Flag to finish execution */
    boolean finish;
    ...
}
```

Esta clase también define unos métodos útiles para ejecutar la máquina abstracta como son la creación de nuevas variables y nombres, la búsqueda de un objeto en la cola de objetos suspendidos (`ObjQ`), la búsqueda de mensajes en la cola `MsgQ`, la búsqueda de métodos en un objeto determinado, decidir si un objeto tiene o no delegación.

```
/* Create a new variable reference */
  Integer NewVar()

/* Create a new name reference */
  Integer NewName()

/* Search an Object process in ObjQ */
  public Process searchObj(Integer Name, byte type, short Meth)

/* Search a method in a specific object */
  public int searchMethod(Process Obj,short Meth)
```

El método `searchObj`, busca un objeto en la cola de objetos, tratando de encontrar una posible comunicación. Si un objeto es encontrado, el método es buscado dentro del proceso objeto. No es posible implementar una técnica como el "hashing" para hacer la búsqueda de objetos ya que la activación de la comunicación depende del "store".

El método `searchMethod` busca un identificador de método en un proceso determinado tratando de retornar la dirección donde está su cuerpo para crear un nuevo proceso.

```
  public int searchMethod(Process Obj,short Meth)
  {
    int ObjPC,
    MethDir = -1,
```

```
    countM  = 7;        // offset to look for a method Id

    short NumMeth,IdMeth;

    ObjPC = Obj.getPC();
    NumMeth = Program.get16(ObjPC+5);
    while ((NumMeth > 0) && (MethDir < 0))
    {
       IdMeth = Program.get16(ObjPC+1+countM);
       if (Meth == IdMeth)
          MethDir = Program.get32(ObjPC+3+countM);
       countM+=7;
       NumMeth--;
    }
    return MethDir;
  }
```

Otro método en la clase `Executor` es el método `schedule` usado para sacar el primer proceso encolado en `RunQ`.

```
  private void schedule()
  {
    Process Proc;
    if (RunQ.length()>0)
    {
      Proc = (Process)RunQ.getAt(0);
      RunQ.delAt(0);
      PCA = Proc.getPC();
      PVA = Proc.getPV();
      PNA = Proc.getPN();
      PAA = Proc.getPA();

#ifdef DEBUG
      System.out.println("Scheduling");
#endif
    }
    else
      finish = true;
  }
```

Uno de los más importantes métodos en la clase `Executor` es el método `run`. Este método tiene el ciclo de ejecución principal, que sólo termina cuando la cola `RunQ` está vacía, condición que es controlada con una variable booleana "finish", la cual cambia de estado en el método `schedule`. Dentro del ciclo, se realiza una selección de la acción que se debe ejecutar, determinada por el puntero de código actual.

Una función "case" ejecuta la instrucción referenciada por el puntero a código actual.

```
public void run()
{
    finish = false;
    if (Program != null)
    {
        schedule();
        while (!finish)
        {
            switch (Program.get8(PCA))
            {
            case NEWV:
              .
              .
              .
            }
        }
    }
}
```

Las instrucciones dentro del ciclo de ejecución principal están compuestas de una instrucción básica que la máquina considera. Cada caso ejecuta lo que la correspondiente regla dice que tiene que hacer para reducirse. (véase la sección 4.9).

La instrucción `NEWV (NEWN)` crea un nuevo nodo en el árbol de traducción de variables (nombres) y actualiza el `PVA (PNA)`.

```
case NEWV:
   PAUX=new NodeList(NewVar());
   ((NodeList)PAUX).setNext(PVA);
```

```
     PVA=(NodeList)PAUX;

#ifdef DEBUG
     System.out.println(PCA+" NEWV "+((NodeList)PAUX).getData());
#endif

     PAUX=null;
     PCA++;
     break;
```

La instrucción PAR crea un nuevo proceso y lo inserta al final de la cola de ejecución (RunQ).

```
  case PAR:
#ifdef DEBUG
     System.out.println(PCA+" PAR " + Program.get32(PCA+1));
#endif

     RunQ.insertAtEnd(new Process(Program.get32(PCA+1),PVA,PNA,PAA));
     PCA+=5;
     break;
```

Hay tres instrucciones para manejar el paso de parámetros. Dos de ellas adicionan parámetros la pila de parámetros (PUSHV y PUSHN) la ultima saca los parámetros de la cola (POP).

```
  case PUSHV:
#ifdef DEBUG
     System.out.println(PCA+" PUSHV " + Program.get16(PCA+1));
#endif
     if (PVA.length()>=Program.get16(PCA+1))
     {
         PAUX = new NodeArg(PVA.getDataAt(Program.get16(PCA+1)),
                            (byte)VARIABLE);
         ((NodeArg)PAUX).setNext((NodeArg)PAA);
         PAA=(NodeArg)PAUX;
         PAUX = null;
         PCA+=3;
```

```
    }
    else
    {
        System.out.println(PCA+" PUSHV Error");
        finish = true;
    }
    break;


  case POP:
#ifdef DEBUG
    System.out.println(PCA+" POP");
#endif

    if (PAA!=null)
    {
        PAUX = new NodeList(((NodeArg)PAA).getData());
        if (((NodeArg)PAA).getType()==VARIABLE)
        {
            ((NodeList)PAUX).setNext(PVA);
            PVA=(NodeList)PAUX;
        }
        else
        {
            ((NodeList)PAUX).setNext(PNA);
            PNA=(NodeList)PAUX;
        }
        PAA=((NodeArg)PAA).getNext();
        PAUX = null;
        PCA++;
    }
    else
    {
        System.out.println(PCA+" POP Error");
        finish = true;
    }
    break;
```

Los procesos objeto están divididos en muchas alternativas dependiendo de la definición misma del objeto. El objeto podría estar o no replicado, y el identificador o la

dirección de replicación pueden ser nombres o variables. Así, se puede tener combinaciones de direcciones con variables y nombres, con replicación o sin ella; pero básicamente estas instrucciones hacen la misma cosa: detectan si hay un mensaje en MsgQ para comunicar el objeto; si no, insertan el objeto en la cola de objetos (`ObjQ`). Si lo hay, lo intenta comunicarlo si es replicado, lo inserta en la cola de objetos, o lo delega si es posible, insertándolo en la misma cola (`ObjQ`). A continuación mostraremos la codificación de los objetos no replicados.

```
     case OBJVV:
     case OBJNN:
     case OBJVN:
     case OBJNV:
#ifdef DEBUG
     System.out.println(PCA+" OBJ "+Program.get16(PCA+1) +" "+
                        Program.get16(PCA+3)+" "+Program.get16(PCA+5));
#endif
     objProc = new Process(PCA,PVA,PNA,PAA);
     objCode = Program.get8(PCA);
     if ((objCode == OBJVV) ||
         (objCode == OBJVN))
         objType = VARIABLE;
     else
         objType = NAME;
     if (objType == VARIABLE)
         objName = (Integer)objProc.getPV().getDataAt
                    (Program.get16(objProc.getPC()+1));
     else
         objName = (Integer)objProc.getPN().getDataAt
                    (Program.get16(objProc.getPC()+1));
     node = MsgQ.FirstNode();
     count=0;
     exitwhile=false;
     while ((node != null) && (!exitwhile))
     {
         msgProc = (Process)node.getData();
         if (Program.get8(msgProc.getPC())==REPLICATE)
         {
             msgProc.setPC(msgProc.getPC()+1);
             IsReplicated = true;
         }
```

```
else
    IsReplicated = false;
if ((objName == ((NodeArg)msgProc.getPA()).getData()) &&
    (objType == ((NodeArg)msgProc.getPA()).getType()))
{
    MethNumber = Program.get16(msgProc.getPC()+1);
    MethPC = searchMethod(objProc,MethNumber);
    if (MethPC > 0)      /* if exist the method label */
    {
        /* method body */
        Process newp = new Process(MethPC,
                                   PVA,
                                   PNA,
                                   msgProc.getPA());
        RunQ.insertAtEnd(newp);
        /* Call continuation */
        newp = new Process(msgProc.getPC()+3,
                           msgProc.getPV(),
                           msgProc.getPN(),
                           msgProc.getPA());
        RunQ.insertAtEnd(newp);
        if (!IsReplicated)
        {
            MsgQ.delAt(count);
            count--;
        }
        exitwhile = true;
    }
    else if (canDelegate(objProc)>=0)  /* delgate */
    {
        if (!IsReplicated)
        {
            MsgQ.delAt(count);
            count--;
        }
        short Super = canDelegate(objProc);
        Process newp = new Process(msgProc.getPC(),
                                   msgProc.getPV(),
                                   msgProc.getPN(),
                                   null);
        /* delega a nombre */
```

```
                if ((Program.get8(objProc.getPC()) == OBJNN) ||
                    (Program.get8(objProc.getPC()) == OBJVN))
                {
#ifdef DEBUG
                    System.out.println("Del Name: " + Super);
#endif
                    PAUX=new NodeArg((msgProc.getPN().getDataAt(Super)),
                                     (byte) NAME);
                    ((NodeArg)PAUX).setNext(((NodeArg)msgProc.getPA()).
                                     getNext());
                    newp.setPA(PAUX);
                }
                else    /* delega a variable */
                {
#ifdef DEBUG
                    System.out.println("Del Var: " + Super);
#endif
                    PAUX=new NodeArg((msgProc.getPV().getDataAt(Super)),
                                     (byte)VARIABLE);
                    ((NodeArg)PAUX).setNext(((NodeArg)msgProc.getPA()).
                                     getNext());
                    newp.setPA(PAUX);
                }
                PAUX = null;
                /* call delegation */
                RunQ.insertAtEnd(newp);
            }
        }
        if (IsReplicated)
            msgProc.setPC(msgProc.getPC()-1);
        node = node.getNext();
        count++;
    }
    if (!exitwhile)
    {
        ObjQ.insertAtEnd(objProc);
#ifdef DEBUG
        System.out.println("Enqueued");
#endif
    }
    schedule();
```

```
    break;
```

La instrucción CALL toma la referencia del objeto receptor de la pila apuntada por PAA y trata de buscarlo en ObjQ; si lo encuentra, se crea un nuevo proceso. Su PC apunta a la primera instrucción del método requerido, el PV y el PN del objeto receptor y el PA del proceso actual (PAA). Si no hay un objeto para comunicar, el proceso CALL es insertado en la cola de mensajes suspendidos (MsgQ).

Para manejar las restricciones hay varias instrucciones. Algunas de ellas, como TERMV, TERMC, TERMN , TERMF, ATOM, SENTCO y SENTCN son para construir la restricción y otras como TELL y ASK para comunicarle la restricción al sistema.

Las instrucciones TERMV, TERMC, TERMN , TERMF, ATOM, SENTCO y SENTCN tienen el mismo comportamiento pero cambia la información guardada.

```
  case TERMV :
#ifdef DEBUG
    System.out.println(PCA+" TERMV " +
                       Program.get16(PCA+1));
#endif

    if (PAUX ==null)
    {
        System.out.println("Bad Constraint");
        finish = true;
    }
    else
    {
        if (PVA.length()>Program.get16(PCA+1))
        {
            faux=new FrameConst((byte)FO_TERMV,
                                ((Integer)PVA.getDataAt
                                 (Program.get16(PCA+1))).intValue());

            ((FrameConst)PAUX).insertSon(faux);

            PAUX=faux;
            PCA+=3;
        }
```

```
        else
        {
            System.out.println(PCA + " TERMV Error");
            finish = true;
        }
    }
    break;
```

Las instrucciones ASK y TELL, preguntan al sistema de restricciones o adicionan nueva información respectivamente. Ambas usan el PAA como referencia al árbol donde la restricción fue creada por las anteriores instrucciones.

```
  case TELL:
#if defined(DEBUG)
    System.out.println("-----> "+PCA+" TELL " + PAA);
#elif defined(DEBUGSTORE)
    System.out.println("-----> "+PCA+" TELL " + PAA);
#endif

    if (PAA==null)
    {
        System.out.println("Null Constraint");
        finish = true;
    }
    else
    {
        store.tell((FrameConst)PAA);
        RunQ.addAtEnd(AskQ);
        AskQ = new List();
        RunQ.addAtEnd(MsgQ);
        MsgQ = new List();

        PAUX = null;
        PAA = null;
        PCA++;
    }
    break;
```

La instrucción ASK toma la respuesta del sistema de restricciones: si el sistema responde con el entero 2, esto quiere decir que no puede deducir ni verdadero ni

falso. En este caso el proceso es encolado en la cola **AskQ**; si el sistema retorna 0 (falso), continua con la ejecución de otro proceso y el proceso actual es eliminado; si responde 1 (verdadero), continua con la siguiente instrucción en el proceso actual.

```
  case ASK:
#if defined (DEBUG)
     System.out.println("-----> "+PCA+" ASK " + PAA);
#elif defined (DEBUGSTORE)
     System.out.println("-----> "+PCA+" ASK " + PAA);
#endif

     int res;
     if (PAA==null)
     {
         System.out.println("Null Constraint");
         finish = true;
     }
     else
     {
         res = store.ask((FrameConst)PAA);

#if defined(DEBUG) || defined(DEBUGSTORE)
         switch (res)
         {
         case 2:
             System.out.println("A: N/A");
             break;
         case 0:
             System.out.println("A: FALSE");
             break;
         case 1:
             System.out.println("A: TRUE");
             break;
         }
#endif
         switch (res)
         {
         case 2:
             AskQ.insertAtEnd(new Process(PCA,PVA,PNA,PAA));
         case 0:
             schedule();
```

```
            break;
        case 1:
            PCA++;
            PAUX=null;
            PAA=null;
            break;
        }
    }
    break;
```

5.4 RECOLECCION DE BASURA

Según el diseño de la máquina abstracta se pueden presentar 3 situaciones donde sería deseable tener un recolector de basura para mayor eficiencia en la ejecución de un programa.

El primer colector se nombra en la sección 4.9 donde para la regla SCHED se hace necesario liberar la memoria de los enlaces de las variables y lo nombres o usar un colector de basura. El segundo en la sección 4.12 debido a la replicación de mensajes y a la delegación; este es opcional para una mayor eficiencia en espacio. El tercero es debido a que en las colas de objetos y de mensajes es posible que queden procesos con los cuales ningún otro proceso puede comunicarse, siendo procesos que no se podrán reducir; este también es opcional y el más difícil de implementar.

Para esta implementación se han tenido en cuenta sólo el primer colector de basura ya que se puede usar los recursos que provee el lenguaje de programación JAVA en el cual está implementada la máquina. Así, se deja que sea la máquina virtual de JAVA la encargada de esta tarea.

5.5 SISTEMA DE RESTRICCIONES

El sistema de restricciones en la implementación de la máquina abstracta está parametrizado y puede ser cambiado sin necesidad de hacer ningún cambio en la máquina.

Por ahora, la máquina está trabajando con un sistema que permite sólo dos tipos de

restricciones:

- Restricciones de dominio, como x *en* R.
- Restricciones de igualdad sobre variables, como $x = y$.

La BNF de las restricciones reconocidas por el sistema se muestra en la tabla 5.1

<restricción>	::=	<variable> en <rango><continuación> |
		<variable> = <variable>
<variable>	::=	identificador
<rango>	::=	<valor><continuación_rango>
<valor>	::=	número |
		Min(<variable>) |
		Max(<variable>)
<continuación_rango>	::=	: <valor><continuación_rango> |
		<empty>
<continuación>	::=	+ {número}|
		<empty>

Tabla 5.1: BNF para la Especificación de las Restricciones Reconocidas por el Sistema de Restricciones

Un rango es la enumeración explicita de los valores de una variable. Los valores en el rango de las variables son números enteros que pueden ser presentados en dos formas: como el número mismo o como el Min (mínimo) o Max (máximo) de el rango de otras variables. Para mayor información acerca del sistema de restricciones ver [ABH98].

5.6 COMPLEJIDAD DE LA IMPLEMENTACION

Las búsquedas en las colas de objetos y mensajes tienen complejidad de orden lineal. El cuello de botella de la ejecución de la máquina se presenta en la complejidad temporal presentado por la función "Ask" en el sistema de restricciones que, como ya se dijo, está fuera del alcance de este trabajo.

6 PRUEBAS Y RESULTADOS

En este capítulo se mostrarán algunas de las pruebas hechas a la máquina, describiendo el problema por resolver. Se mostrará la codificación del problema en PiCO y el comportamiento esperado, así como el resultado de la ejecución y el estado final de la máquina, enumerando los procesos pendientes en las colas de objetos, mensajes y "ask" suspendidos. El código del programa en Java con el cual se hizo el archivo binario que la máquina toma se muestra únicamente en la primera prueba, ya que para las demás pruebas la codificación es muy similar.

6.1 EMISION

En los ambientes paralelos o concurrentes regularmente es deseable enviar un mismo mensaje a varios destinatarios en un momento determinado.

Para esta prueba se crean tres objetos A, el último con delegación a B y los dos primeros sin delegación. El primer objeto tiene un método llamado *run* el cual no recibe ningún parámetro. En su cuerpo se crea una variable t y se le impone la restricción t *in 2*.

El segundo objeto también tiene un método llamado *run* donde se crea un objeto replicado llamado B, sin delegación. Dicho objeto B tiene un método *run* que no tiene comportamiento.

El último, objeto A con delegación a B tiene un método *set* que crea una variable k y le impone la restricción k *in 5*.

Por último hay dos mensajes replicados al objeto A, uno al método *run* y otro al

método *set*.

La codificación en PiCO puede quedar de la siguiente forma:

$(\upsilon A, B)$
$((A, A) \triangleright [run : ()(\upsilon t).!(t\ in\ \{2\})]$
$| (A, A) \triangleright [run : ()(*(B, B) \triangleright [run : ()0])]$
$| (A, B) \triangleright [set : ()(\upsilon k).!(k\ in\ \{5\})]$
$| *A \triangleleft run : []$
$| *A \triangleleft set : []$
$)$

Para codificar este problema según la especificación de entrada de la máquina, se utilizó el siguiente código en lenguaje Java.

```
/*0*/   MyFile.writeByte(NEWN);          /*  A  */
        MyFile.writeByte(NEWN);          /*  B  */

        MyFile.writeByte(PAR);
/*3*/   MyFile.writeInt(33);             /* paralelo 1 */

/*7*/   MyFile.writeByte(OBJNN);         /* (A,A)>run */
        MyFile.writeShort(1);
/*10*/  MyFile.writeShort(1);
        MyFile.writeShort(1);
/*14*/  MyFile.writeByte(METH);
        MyFile.writeShort(1);
/*17*/  MyFile.writeInt(21);

        MyFile.writeByte(NEWV);          /*  L  */
/*22*/  MyFile.writeByte(ATOM);          /* L in {2} */
        MyFile.writeByte(C_IN);
/*24*/  MyFile.writeByte(2);             /* aridad */
        MyFile.writeByte(TERMV);
/*26*/  MyFile.writeShort(0);            /* L */
        MyFile.writeByte(TERMC);
/*29*/  MyFile.writeShort(2);
```

```
        MyFile.writeByte(TELL);
        MyFile.writeByte(RET);

/*33*/  MyFile.writeByte(PAR);          /* viene de paralelo 1 */
        MyFile.writeInt(67);            /* paralelo 2 */

/*38*/  MyFile.writeByte(OBJNN);        /* (A,A)>run */
        MyFile.writeShort(1);
/*41*/  MyFile.writeShort(1);
        MyFile.writeShort(1);
/*45*/  MyFile.writeByte(METH);
        MyFile.writeShort(1);
/*48*/  MyFile.writeInt(52);

/*52*/  MyFile.writeByte(OBJNNR);       /* *(B,B)>run */
        MyFile.writeShort(0);
/*55*/  MyFile.writeShort(0);
        MyFile.writeShort(1);
/*59*/  MyFile.writeByte(METH);
        MyFile.writeShort(1);
/*62*/  MyFile.writeInt(66);
/*66*/  MyFile.writeByte(RET);          /* begin run */

/*67*/  MyFile.writeByte(PAR);          /* viene de paralelo 2 */
        MyFile.writeInt(98);            /* paralelo 3 */

/*72*/  MyFile.writeByte(OBJNN);        /* (A,B)>set */
        MyFile.writeShort(1);
/*75*/  MyFile.writeShort(0);
        MyFile.writeShort(1);
/*79*/  MyFile.writeByte(METH);

        MyFile.writeShort(2);
/*82*/  MyFile.writeInt(86);

        MyFile.writeByte(NEWV);         /*  K  */
/*87*/  MyFile.writeByte(ATOM);         /* K in {5} */
        MyFile.writeByte(C_IN);
/*89*/  MyFile.writeByte(2);            /* aridad */
        MyFile.writeByte(TERMV);
/*91*/  MyFile.writeShort(0);           /* K */
```

```
        MyFile.writeByte(TERMC);
/*94*/  MyFile.writeShort(5);
        MyFile.writeByte(TELL);
        MyFile.writeByte(RET);

/*98*/  MyFile.writeByte(PAR);          /* viene de paralelo 3 */
        MyFile.writeInt(111);           /* paralelo 4 */

/*103*/ MyFile.writeByte(PUSHN);        /* A < run() */
        MyFile.writeShort(1);
        MyFile.writeByte(REPLICATE);
/*107*/ MyFile.writeByte(CALL);
        MyFile.writeShort(1);
        MyFile.writeByte(RET);

/*111*/ MyFile.writeByte(PUSHN);        /* A < run()  viene de 4*/
        MyFile.writeShort(1);
        MyFile.writeByte(REPLICATE);
/*115*/ MyFile.writeByte(CALL);
        MyFile.writeShort(2);
        MyFile.writeByte(RET);
```

Cuando la máquina ejecuta este problema, debería crear las variables A y B, y continuar con la creación del los objetos (A, A), (A, A) y (A, B).

El mensaje replicado *run* al objeto A debe comunicarse con los dos primeros objetos y delegarse al objeto B,los dos primeros objetos perecen ya que no son replicados pero el último no, por no haberse comunicado sino haberse delegado. Cuando el cuerpo de el método *run* del primer objeto es ejecutado, se debe crear la variable t e instanciarse. De igual forma cuando el cuerpo del método *run* del segundo objeto es ejecutado, se crea el objeto replicado $*(B, B)$ para comunicarlo con la delegación que se realizó anteriormente.

El mensaje replicado *set* al objeto A debe comunicarse con el tercer objeto y ejecutar el cuerpo del método que es la creación de la variable k y su instanciación. En este momento, el objeto A con delegación a B desaparece y sólo debe permanecer el objeto replicado B y los 2 mensajes replicados.

Se usó la máquina en modo de depuración para ejecutar y ver el resultado de este

problema, arrojando el siguiente resultado:

```
Crea las dos variables A y B y encola el primer objeto.
0 NEWN 1
1 NEWN 2
2 PAR 33
7 OBJ 1 1 1
Enqueued
Scheduling


Encola el segundo objeto.
33 PAR 67
38 OBJ 1 1 1
Enqueued
Scheduling


Encola el objeto que delega.
67 PAR 98
72 OBJ 1 0 1
Enqueued
Scheduling


El call replicado se comunica con dos objetos y delega a B
(los cuerpos de los métodos y el call son encolados).
98 PAR 111
103 PUSHN 1
106 *CALL 1
Enqueued
Communicated
Communicated
Del Name:  0
Scheduling


El call replicado se comunica con un objeto.
111 PUSHN 1
114 *CALL 2
Enqueued
Communicated
Scheduling
```

```
Ejecuta el cuerpo del método del primer objeto.
21 NEWV 1
22 ATOM 0 2
25 TERMV 0
28 TERMC 2
-----> 31 TELL v1 in 2
32 RET
Scheduling


Continuación de la primera comunicación.
110 RET
Scheduling


Ejecuta el cuerpo del método del segundo objeto
52 OBJR 0 0 1
Enqueued
Scheduling


Continuación de la segunda comunicación.
110 RET
Scheduling


Mensaje delegado.  Se comunica con el objeto creado en el segundo objeto.
107 CALL 1
Method PC: 66
PAA: 2
Communicated
Scheduling


Se ejecuta el cuerpo del método del tercer objeto.
86 NEWV 2
87 ATOM 0 2
90 TERMV 0
93 TERMC 5
-----> 96 TELL v2 in 5
97 RET
Scheduling
```

```
Continuación del llamado al método set del objeto A
118 RET
Scheduling


Intenta comunicar el primer llamado pero es encolado.
106 *CALL 1
Enqueued
Scheduling


Intenta comunicar el segundo llamado pero es encolado.
114 *CALL 2
Enqueued
Scheduling


Ejecución del cuerpo del método del objeto B.
66 RET
Scheduling


Continuación de la comunicación con el mensaje delegado.
110 RET
```

Al terminar la ejecución del programa, la máquina queda en el siguiente estado final:

En la Cola de objetos queda un solo elemento: el objeto B replicado.

[1](0)PC:52 PV:null PN:2,1 PA:null

En la cola de mensajes suspendidos quedan dos elementos: dos mensajes al objeto A replicados.

[2](0)PC:106 PV:null PN:2,1 PA:1,
(1)PC:114 PV:null PN:2,1 PA:1,

Por último la cola de "ask" suspendidos queda vacía.

El Almacenamiento de las restricciones queda en estado consistente y las variables de la siguiente manera: La variable v1 (t) queda con el valor del entero 2 y la variable v2 (k) con el valor de 5. Además, no queda ninguna restricción pendiente en el sistema.

6.2 SINCRONIZACION POR SEMAFOROS Y POR STORE

Los semáforos son mecanismos que se usan para el manejo de la exclusión mutua. Se usan para reservar y liberar un recurso, o para que desactiven el proceso que las llama cuando el recurso está ocupado.

Un mecanismo de semáforo consta de dos operaciones primitivas: señal y espera. Sólo estas dos operaciones pueden acceder y manipular los semáforos [Mil88].

En este problema, se crea una clase llamada *Sem* donde se encuentra el método *signal*. Cuando el método *signal* es invocado, se crea un objeto que tiene el método *set* que no tiene comportamiento. Así, una región crítica controlada por semáforos puede ser enmarcada por una invocación al método *set* al principio de la región que consume el recurso y otra al método *signal* al final de la región donde el recurso se libera. De manera similar se puede sincronizar procesos dependiendo del valor de las variables en el almacenamiento de las restricciones.

El nombre *Sem* actúa como la clase semáforo mientras que A y B actúan como objetos de esa clase.

$$
\begin{array}{l}
(\nu Sem, A, B) \\
(\ *(Sem, Sem) \triangleright [signal : (x).((x,x) \triangleright [set : ()0]\ ]) \\
\quad \mid (\nu p1, p2, p3, r, m, k, t, on) \\
\quad (!(on\ in\ \{1\}) \\
\quad \mid (Sem \lhd signal : [A]) \\
\quad \mid (Sem \lhd signal : [B]) \\
\quad \mid (!(p1 = on).(\ A \lhd set : [].?(P3 = on).!(R\ in\ \{2\}).(Sem \lhd signal : [A]) \\
\qquad\qquad\qquad \mid B \lhd set : [].!(M\ in\ \{2\}).(Sem \lhd signal : [B])\ ) \\
\quad)
\end{array}
$$

$$
\begin{array}{l}
\quad | \ (!(P2 = on).(A \lhd set : [].!(K \ in \ \{1\}).?(T = on).(Sem \lhd signal : [A])) \\
\quad\quad) \\
\quad | \ (!(P3 = on).(B \lhd set : [].?(K = on).!(T \ in \ \{1\}).(Sem \lhd signal : [B])) \\
\quad\quad) \\
\quad) \\
)
\end{array}
$$

Para mayor simplicidad, en este problema sólo se describirá lo que la máquina ejecuta pero no se mostrarán sus resultados.

Crea los nombres Sem, A y B.
Encola el objeto replicado *(Sem,Sem).
Crea las variables p1,p2,p3,r,m,k,t y on.
Hace *!(on en 1)* en el sistema de restricciones.
Comunica el mensaje signal(A) del objeto Sem.
Comunica el mensaje signal(B) del objeto Sem.
Ejecuta el cuerpo de *signal* de la primera comunicación, creando un objeto A.
Ejecuta la continuación de la primera comunicación.
Hace *!(p1=on)* y se comunica con el método *set* del objeto A.
Ejecuta el cuerpo de *signal* de la segunda comunicación, creando un objeto B.
Ejecuta la continuación de la segunda comunicación.
Hace *!(p2=on)* y se encola el mensaje al método *set* del objeto A.
Se comunica con el mensaje *set* del objeto B.
Ejecuta el cuerpo *set* del objeto A.
Hace *?(p3=on)* al sistema de restricciones y él responde N/A.
Hace *!(p3=on)* y se encola el mensaje al método *set* del objeto B.
Ejecuta el cuerpo *set* del objeto B.
Hace *!(m en 2)* y hace la comunicación de signal(B) en el objeto Sem.
Hace *?(p3=on)*, el sistema respoonde True, luego hace *!(r en 2)* y comunica a signal(A) en el objeto Sem.
Encola el mensaje *set* al objeto A.
Encola el mensaje *set* al objeto B.
Ejecuta el cuerpo de *signal* en Sem, creando un objeto B.

Continuación del llamado de signal(B) en el objeto Sem.
Ejecuta el cuerpo de *signal* en Sem, creando un objeto A.
Continuación del llamado de signal(A) en el objeto Sem.
Ejecuta el cuerpo *set* del objeto B.
Hace *?(k=on)*, el sistema responde N/A.
Ejecuta el cuerpo *set* del objeto A.
Hace *!(k en 1)* y luego *?(t=on)*, el sistema responde N/A.
Hace *?(k=on)* el sistema responde True, *!(t en 1)*, comunica a signal(B) en el objeto Sem.
Hace *?(t=on)* el sistema responde True, comunica a signal(A) en el objeto Sem.
Ejecuta el cuerpo de *signal* con parametro B.
Continuación del llamado signal(B).
Ejecuta el cuerpo de *signal* con parametro A.
Continuación del llamado signal(A).

Al terminar la ejecución del programa, la máquina queda en el siguiente estado final:

En la Cola de objetos quedan tres elementos, el objeto *Sem* replicado, un objeto A y un objeto B producto de los dos últimos llamados al método *signal* del objeto *Sem*.

[3](0)PC:8 PV:null PN:3,2,1 PA:null,
(1)PC:23 PV:null PN:3,3,2,1 PA:null,
(2)PC:23 PV:null PN:2,3,2,1 PA:null

La cola de Mensajes y Ask suspendidos quedan vacías.

El Almacenamiento de las restricciones queda en estado consistente y las variables de la siguiente manera: Las variables *p1, p2, p3, k, t* y *on* quedan con el valor 1 y las variables r y m con el valor de 2. Además, no queda ninguna restricción pendiente en el sistema.

7 CONCLUSIONES

Los cálculos computacionales modelan, matemática, sólida y formalmente los lenguajes de programación. PiCO es un cálculo computacional desarrollado por el grupo AVISPA el cual integra objetos concurrentes y restricciones como elementos básicos.

El resultado principal del presente trabajo es el diseño e implementación de una máquina abstracta basada en PiCO para ejecutar programas por restricciones, concurrentes y orientados a objetos que cumplan con las reglas de dicho cálculo. Para su desarrollo, se estudiaron diferentes modelos computacionales formales incluyendo el diseño e implementación de otros cálculos y máquinas abstractas teniendo especial cuidado en sus fortalezas y desventajas.

Para garantizar que la máquina abstracta diseñada satisface las especificaciones del cálculo PiCO, se presentó la misma también como un cálculo adaptando las reglas de transición de PiCO para hacer más sencilla la implementación.

La adecuación más importante consistió en que según la sintaxis de PiCO cualquier proceso puede ser replicado; en la máquina abstracta sólo se pueden replicar los procesos normales. Esta simplificación no le resta poder de expresividad al cálculo pero hace más sencilla y eficiente la implementación de la máquina ya que replicar todo tipo de procesos puede generar otros procesos que no alteran el resultado final pero sí consumen recursos de espacio en memoria y tiempo de ejecución.

Para definir los alcances de variables y nombres dentro de un proceso, se utilizó un árbol ascendente para así, tener espacios locales de computación en cada proceso, a diferencia de como se hace en otros cálculos como TyCO.

En el cálculo PiCO el sistema de restricciones es parametrizable. En la máquina ab-

stracta esta misma característica fue considerada permitiendo la integración de diferentes sistemas de restricciones de una manera sencilla, sin afectar el funcionamiento de la máquina.

Dado que el diseño es independiente de un lenguaje de programación en particular, Java por su portabilidad y los requerimientos exigidos en la justificación del presente trabajo, fue utilizado para la implementación, teniendo en cuenta que por ser este un lenguaje interpretado, el rendimiento en la ejecución de los programas se ve afectado, pero existen algunos métodos para solucionar dicho problema. Además, la falta de manejo de datos sin signo en Java, requirió un manejo especial. La ventaja de Java es obviamente su portabilidad.

CORDIAL un lenguaje de programación visual orientado a objetos y con restricciones, creado por el grupo AVISPA, tiene su base formal en el cálculo PiCO. Este lenguaje usa la máquina abstracta para ejecutar sus programas a pesar de que él no explota toda la capacidad computacional del cálculo. Sin embargo, la máquina puede ser usada por otros lenguajes de programación que tenga características de los paradigmas orientado a objetos, concurrente y por restricciones.

8 RECOMENDACIONES

- Implementar un sistema de restricciones poderoso y más completo para la máquina pues es allí donde está el poder y el cuello de botella de la eficiencia en la programación por restricciones.
- Idear un mecanismo para hacer enlace dinámico de programas.
- Implementar un mecanismo de recolección de basura tanto en la pila de objetos como la pila de mensajes suspendidos.
- Idear un mecanismo para manejo interno de cadenas.
- Idear un mecanismo de entrada por teclado, salida por consola y manejo de archivos.
- Portar la máquina virtual a un ambiente paralelo para comparar su desempeño y explotar la característica paralela del cálculo.
- Tener una manera más amigable de ver el conjunto de soluciones almacenadas en la memoria de restricciones.
- Implementar un compilador de tal manera que su código fuente sea PiCO y su código destino sea la máquina abstracta.
- Debido a que el cálculo PiCO ha estado cambiando, se debería tener en cuenta los últimos cambios para ser modelados en la máquina.

Bibliografía

[ABH98] Gloria Alvarez, Antal A. Buss, and Mauricio Heredia. Mapico. an abstract machine for the pico calculus. Technical report, Pontificia Universidad Javeriana - Cali, September 1998.

[AC96] Martín Abadi and Luca Cardelli. *A Theory of Objects.* Springer, 1996.

[ADQ+98] Gloria Alvarez, Juan Francisco Diaz, Luis O. Quesada, Camilo Rueda, and Frank D. Valencia. Pico: A calculus of concurrent constraint object for musical applications. ECAI98, Workshop on Constraint & the Arts, Brighton, England, 1998.

[ASU90] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compiladores: Principios, tecnicas y herramientas.* Addison-Wesley, 1990.

[BFHW94] Cristoph Boeschen, Chiristian Fecht, Andreas Hense, and Reinhard Wilhelm. An abstract machine for an object-oriented language with top-level classes, March 1994.

[BOP+96] Tomás Bautista, Tobias Oetiker, Humbert Partl, Irene Hyna, and Elisabeth Schlegl. Una descripción de latex2e, February 1996.

[Eco94] Umberto Eco. *Como se Hace una Tesis: Técnicas y procedimientos de investigación, estudio y escritura.* Gedisa Editorial, 1994.

[HM94] M. Henz and M. Muller. Programming in oz. *DFKI Documentation series*, 1994.

[Lam94] Leslie Lamport. *Latex: a document preparation system.* Addison-Wesley, 1994.

[Lop98] Luís Lopes. The tyco abstract machine implementation. Technical report, Universidade do Porto, January 1998.

[LSV97] Luís Lopes, Fernando Silva, and Vasco T. Vasconcelos. A framework for compiling object calculi. Technical report, Universidade do Porto, November 1997. Technical Report Series: DCC-97-12.

[LV97] Luís Lopes and Vasco T. Vasconcelos. Tyco abstract machine: The definition. Technical report, Universidade do Porto, April 1997. Technical Report Series: DCC-97-1.

[LY96] Tim Lindholm and Frank Yellin. *The Java^{TM} Virtual Machine Specification.* Addison-Wesley, September 1996.

[McA92] David McAllester. First order logic, 1992.

[Mil88] Milan Milenković. *Sistemas Operativos. Conceptos y Diseño.* McGraw-Hill, 1988.

[Mil91] Robin Milner. The polyadic π-calculus: a tutorial. Technical Report ECS-LFCS-91-180, Laboratory for Foundations of Computer Science, Department of Computer Science, University of Edinburgh, UK, October 1991. Also in *Logic and Algebra of Specification*, ed. F. L. Bauer, W. Brauer and H. Schwichtenberg, Springer-Verlag, 1993.

[MPW90] Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, part i. September 1990.

[MPW92] Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, Parts I and II. *Journal of Information and Computation*, 100:1–77, September 1992.

[MS92] Robin Milner and Davide Sangiorgi. Barbed bisimulation. In W. Kuich, editor, *Proc. of 19th International Colloquium on Automata, Languages and Programming (ICALP '92)*, volume 623 of *Lecture Notes in Computer Science*, pages 685–695. Springer-Verlag, 1992.

[MSS95] Michael Mehl, Ralf Scheidhauser, and Christian Schulte. An abstract machine for oz, June 1995.

[Nie96] Joachim Niehren. Functional computation as concurrent computation. In *Proceedings of the ACM Symposium on Principles of Programming Languages.* ACM Press, January 1996. to appear.

[NM95] Joachim Niehren and Martin Müller. Constraints for Free in Concurrent Computation. In Kanchana Kanchanasut and Jean-Jacques Lévy, editors, *Asian Computing Science Conference*, Lecture Notes in Computer Science, vol. 1023, pages 171–186, Pathumthani, Thailand, December 11–13 1995. Springer-Verlag.

[Pey93] Simon Peyton. Implementig lazy functional languages on stock hardware: the spineless tagless g-machine. March 1993.

[Pie95] Benjamin C. Pierce. Foundational calculi for programming languages. 1995.

[Pie96] Benjamin C. Pierce. Programming in the pi-calculus: An experiment in programming language design. Tutorial notes on the PICT language. Available electronically, 1996.

[Pra84] Terrence W. Pratt. *Programming Languages: Design and Implementation.* Prentice Hall, 1984.

[PT94] Benjamin C. Pierce and David Turner. PICT user manual. Forthcoming report. Available electronically, 1994.

[PT96] Benjamin C. Pierce and David Turner. Pict: A programming language based on the pi-calculus. Technical report in preparation; available electronically, 1996.

[QRT97a] Luis O. Quesada, Camilo Rueda, and Gabriel Tamura. Programación visual de *Cordial.* Technical Report No.3, AVISPA Research Team, 1997.

[QRT97b] Luis O. Quesada, Camilo Rueda, and Gabriel Tamura. The visual model of cordial. In *Proceedings of the CLEI97*, Valparaiso, Chile, 1997.

[QRT97c] Luis O. Quesada, Camilo Rueda, and Gabriel Tamura. The visual model of cordial, 1997. XXIII Conferencia Latinoamericana de Informática.

[QRT98] Luis O. Quesada, Camilo Rueda, and Gabriel Tamura. Rule specification of the translation of cordial into pico. Technical report, AVISPA Research Team, nov 1998.

[RAQ+98] Camilo Rueda, Gloria Alvarez, Luis O. Quesada, Gabriel Tamura, Frank D. Valencia, Juan F. Díaz, and Gerard Assayag. Integrating constraints and concurrent objects in musical applications: A calculus and its visual language. Technical report, AVISPA Research Team, 1998.

[Sar93] Vijay A. Saraswat. *Concurrent Constraint Programming.* The MIT Press, Cambridge, MA, 1993.

[Shi96] Olin Shivers. Supporting dynamic languages on the java virtual machine. Technical report, Artificial Intelligence Laboratory, Massachusetts Institute of Technology, may 1996.

[Smo94] Gert Smolka. A calculus for higher-order concurrent constraint programming with deep guards. Research Report RR-94-03, Deutsches Forschungszentrum für Künstliche Intelligenz, Stuhlsatzenhausweg 3, D-66123 Saarbrücken, Germany, February 1994.

[Tur95] David Turner. *The Polymorphic Pi-Calculus: Theory and Implementation.* PhD thesis, Laboratory for Foundations of Computer Science, Department of Computer Science, University of Edinburgh, UK, 1995.

[Vas94a] Vasco T. Vasconcelos. Typed concurrent objects. In M. Tokoro and R. Pareschi, editors, *Proc. of 8th European Conference on Object-Oriented Programming (ECOOP'94)*, volume 821 of *Lecture Notes in Computer Science*, pages 100–117. Springer-Verlag, July 1994.

[Vas94b] Vasco T. Vasconcelos. *Typed Concurrent Objects.* PhD thesis, Keio University, November 1994.

[VDR97a] F. Valencia, J. F. Diaz, and C. Rueda. The π^+-calculus. In *Proceedings of the CLEI97*, Valparaiso, Chile, 1997.

[VDR97b] Frank D. Valencia, Juan Francisco Diaz, and Camilo Rueda. The π^+-calculus: Uses and behavioral equivalence. Technical report, AVISPA Research Team, 1997.

[Wal91] David Walker. π-calculus semantics of object-oriented programming languages. In Takayasu Ito and Albert Meyer, editors, *Theoretical Aspects of Computer Software*, volume 526 of *Lecture Notes in Computer Science*, pages 532–547. Springer-Verlag, 1991. Available as Report ECS-LFCS-90-122, University of Edinburgh.

[Wal95] David Walker. Objects in the π-calculus. *Journal of Information and Computation*, 116(2):253–271, 1995.

Anexo A. Ejemplo de Codificación en MAPiCO

Anexo A. Ejemplo de Codificación en MAPiCO

Este ejemplo devuelve en un parámetro r el valor de una variable global x elevado a una potencia n que entra como primer parámetro.

$$fa(n,r) = \begin{cases} r = x & \text{si } n = 1 \\ r = fa(n-1) * x & \text{en otro caso} \end{cases}$$

Para este caso donde el primer parámetro es 3 y la variable global x es 8, la variable r quedaría instanciada en 512.

En el cálculo PiCO, este programa podría ser codificado como:

$$\begin{aligned}(\upsilon\ fa)(\upsilon\ x)((*fa \triangleright [input : (n,r)(?(n=1).!(r=x) \\ &\quad |?(n>1).(\upsilon\ t\ p)((!(t=n-1).!(r=p*x))). \\ &\quad\quad (fa \triangleleft input[t,p]))]) \\ &\quad |(\upsilon\ k\ s)(!(x=8).!(s=3).fa \triangleleft input[s,k]))\end{aligned}$$

En el lenguaje de máquina, pueden haber muchos formas de codificar este programa ya que el orden de los procesos no modifica la semantica del programa. Una de las posibles codificaciones puede ser de la siguiente manera, teniendo en cuenta que el sistema de restricciones soporta las funciones aqui utilizadas:

```
0    newn                              ; fa
1    newv                              ; x
2    par        40
3    objnr      0       0     1        ; *fa, no delega, 1 método
4    meth       0       5              ; input
5    pop                               ; n
6    pop                               ; r
7    par        20
8    atom       eq      2              ; =
9    termv      1                      ; n
10   termc      1                      ; 1
11   ask
```

```
12   atom        eq     2          ; =
13   termv       0                 ; r
14   termv       2                 ; x
15   tell
16   ret
17   atom        >      2          ; >
18   termv       1                 ; n
19   termc       1                 ; 1
20   ask
21   newv                          ; t
22   newv                          ; p
23   atom        eq     2          ; =
24   termv       1                 ; t
25   termf       less   2          ; -
26   termv       3                 ; n
27   termc       1                 ; 1
28   tell
29   atom        eq     2          ; =
30   termv       2                 ; r
31   termf       mult   2          ; *
32   termv       0                 ; p
33   termv       4                 ; x
34   tell
35   pushv       0                 ; p
36   pushv       1                 ; t
37   pushn       0                 ; fa
38   call        0                 ; input
39   ret
40   newv                          ; k
41   newv                          ; s
42   atom        eq     2          ; =
43   termv       2                 ; x
44   termc       8                 ; 8
45   tell
46   atom        eq     2          ; =
47   termv       0                 ; s
48   termc       3                 ; 3
49   tell
50   pushv       1                 ; k
51   pushv       0                 ; s
52   pushn       0                 ; fa
```

```
53   call        0                ; input
54   ret
```

En la implementación actual de la máquina abstracta, no es posible ejecutar el ejemplo anterior ya que el sistema de restricciones no considera algunas de las funciones aquí usadas.

Anexo B. Uso de la Máquina Abstracta

Anexo B. Uso de la Máquina Abstracta

La máquina abstracta implementada es de línea de comando, razón por la cual el archivo de entrada del programa se especifica en los argumentos de la misma, en el momento de la invocación.

Debido a que la implementación está hecha en Java, se debe correr su máquina virtual. La sintaxis de la invocación es la siguiente:

```
java Mapico program.pico
```

donde program.pico denota un archivo binario con el programa a ejecutar. Este programa tiene la estructura mostrada en el capítulo de diseño de la máquina.

En la variable de ambiente CLASSPATH debe estar el directorio donde recide la máquina, el sistema de restricciones (si no es el mismo de la máquina) y la librería generica jgl ya que el sistema de restricciones (que en este momento usa la máquina) hace uso de esta librería.

Anexo C. Guia de Implantación

Anexo C. Guia de Implantación

Para construir la máquina se utilizó la utilidad make de GNU, el preprocesador cpp de GNU (The GNU C-Compatible Compiler Preprocessor), el compilador y la máquina virtual de java JDK1.1.6. de Sun Microsystems y la herramienta javadoc de Sun Microsystems para generar documentación en formato HTML acerca de la implementación de la máquina. El archivo de dependencias se denomina Makefile.

Antes de compilar la máquina abstracta hay que asegurarse de que el sistema de restricciones está compilado y que el path del sistema está en la variable CLSPATH del Makefile.

Los archivos fuentes están denominados con la extención "`.c.java`" ya que con el uso del preprocesador cpp se han usado opciones de precompilación del lenguaje C. Estos archivos se preprocesan y generan archivos "`.java`", que son compilados con `javac` del JDK1.1.6.

El uso del preprocesador de C (cpp) nos permite usar constantes en el código fuente de una manera más eficiente (en cuanto a tiempo de ejecución y uso de memoria) de lo que lo hace java; además, nos permite usar directivas de precompilación como `#define, #ifdef, #if, #elif, #endif, etc.`

Para compilar la máquina abstracta de PiCO se deben invocar las siguientes instrucciones.

```
make preproceso
make
```

La invocación de `make preproceso` genera los archivos "`.java`" a partir de los archivos "`.c.java`" con el precompilador cpp, quedando listos para ser compilados con javac.

La invocación de `make` genera los archivos "`.class`" compilados.

Para generar la documentación en formato HTML acerca de la implementación de la máquina, debe hacer la siguiente invocación.

```
make docs
```

la cual genera las páginas en el directorio "`./docs/`".

Como se dijo a lo largo de este documento, el sistema de restricciones es totalmente independiente de la máquina abstracta. Razón por la cual el usuario puede utilizar otro sistema de restricciones sin modificar la máquina, ni recompilarla.

Para usar un sistema de restricciones diferente al proveido por esta implementación de MAPiCO, basta con hacer una clase Store.class que contenga los siguientes métodos:

```
public int tell(FrameConst rest)
public int ask(FrameConst rest)
public void ver()
```

Donde el método *ver()* sirve para ver el store resultante de la computación. *FrameConst* es la clase donde recide la restricción, la cual tiene los siguientes atributos y prototipo de métodos:

```
/* Reference to the father frame */
private FrameConst padre;

/* Type of frame */
private int type;

/* Value of frame */
private long valor;

/* Number of children */
private byte numhijos;

/* Children references */
private FrameConst hijos[];
```

```
public FrameConst(byte type, long value, byte sons)
public FrameConst(byte type, long value)
public void insertSon(FrameConst son)
public void setPadre(FrameConst father)
public int getType()
public long getValue()
public byte getNumHijos()
public FrameConst getHijo(int i)
```

Para más información acerca de los métodos de la clase *FrameConst*, ver la documentación generada por javadoc.

Indice de Materias