Reimplementación de la Máquina Abstracta MAPiCO

ALBA LILIANA SARASTI CAMPO
CARLOS ALBERTO LLANO RODRIGUEZ

PONTIFICIA UNIVERSIDAD JAVERIANA
FACULTAD DE INGENIERIA
INGENIERIA DE SISTEMAS Y COMPUTACION
SANTIAGO DE CALI
2007

Reimplementación de la Máquina Abstracta MAPiCO

ALBA LILIANA SARASTI CAMPO
CARLOS ALBERTO LLANO RODRIGUEZ

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

Director
ANTAL ALEXANDER BUSS MOLINA
Ingeniero de Sistemas y Computación

PONTIFICIA UNIVERSIDAD JAVERIANA
FACULTAD DE INGENIERIA
INGENIERIA DE SISTEMAS Y COMPUTACION
SANTIAGO DE CALI
2007

Santiago de Cali, Enero 10 de 2007

Doctor
JORGE FRANCISCO ESTELA
Decano Académico de la Facultad de Ingeniería
Pontificia Universidad Javeriana
Ciudad

Certifico que el presente proyecto de grado, titulado "Reimplementación de la Máquina Abstracta MAPiCO" realizado por ALBA LILIANA SARASTI CAMPO y CARLOS ALBERTO LLANO RODRIGUEZ, estudiantes de Ingeniería de Sistemas y Computación, se encuentra terminado y puede ser presentado para sustentación.

Atentamente,

Ing. ANTAL ALEXANDER BUSS MOLINA
Director del Proyecto

Santiago de Cali, Enero 10 de 2007

Doctor
JORGE FRANCISCO ESTELA
Decano Académico de la Facultad de Ingeniería
Pontificia Universidad Javeriana
Ciudad

Por medio de ésta, presentamos a usted el proyecto de grado titulado "Reimplementación de la Máquina Abstracta MAPiCO" 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,

ALBA LILIANA SARASTI CAMPO CARLOS ALBERTO LLANO RODRIGUEZ

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.

JORGE FRANCISCO ESTELA
Decano Académico de la Facultad de Ingeniería

MARIA CONSTANZA PABÓN
Director de la Carrera de Ingeniería de Sistemas y Computación

ANTAL ALEXANDER BUSS MOLINA
Director de Tesis

PERSONA1
Jurado

PERSONA2
Jurado

Alba Liliana Sarasti Campo

A Dios, mis padres, hermanos y sobrinos.

Carlos Alberto Llano Rodriguez

A mi Mamá, mi Papá, Teresa, Julio, Elena, Yayita y a Dios,
por su paciencia y comprensión, los quiero muchísimo a todos.

Agradecimientos

Los autores expresan sus agradecimientos:

- A Antal Alexander Buss, profesor de la Facultad de Ingeniería de la Pontificia Universidad Javeriana, por su inmensa paciencia, dedicación y apoyo a lo largo de todo el proyecto.
- Al grupo AVISPA.
- A amigos colaboradores de la Pontificia Universidad Javeriana, por todo el apoyo y por creer siempre en este gran esfuerzo.
- A todos nuestros compañeros y amigos por sus palabras de aliento y constante apoyo.
- A nuestras familias que sin ellas, no hubieramos podido alcanzar esta meta.
- A todas aquellas personas que de una u otra forma colaboraron en la realización del presente trabajo.

Índice general

Índice de Tablas

Índice de figuras

Índice de Anexos

Resumen

La Máquina Abstracta *MAPiCO* es la implementación de una máquina abstracta para el cálculo computacional *PiCO* [1][GF98], esta máquina sigue las especificaciones y las reglas de reducción de dicho cálculo, que es la base del modelamiento matemático, sólido y formal del lenguaje de programación *Cordial* [2] [LG97].

La actual implementación de *MAPiCO* [BH99] es correcta, formal y cumple con los requisitos y propósitos para los cuales fue creada, sin embargo, el lenguaje de programación en la que fue implementada, *JAVA*, hace que el tiempo de ejecución se vea afectado ya que por ser un lenguaje interpretado y no compilado compromete la eficiencia [Mar01]; adicionalmente esta implementación no soporta las modificaciones del Cálculo *PiCO* [GG99].

Este trabajo muestra una reimplementación de la Máquina Abstracta *MAPiCO*, considerando las modificaciones realizadas al cálculo [GG99] y la extensión de instrucciones de la máquina a través de módulos dinámicos, que garantiza en un futuro no rediseñar la Máquina Abstracta. Así mismo se ofrece una alternativa de eliminación de variables locales de procesos que ya se han ejecutado por completo y que pueden no necesitarse más en el transcurso de la ejecución.

[1] *PiCO*: Cálculo de objetos y restricciones
[2] *CORDIAL*: Constraints Objects Relations Defining an Iconic Applicative Languaje

INTRODUCCIÓN

Una Máquina Abstracta define la arquitectura para la ejecución de programas de lenguajes que estan soportados por un cálculo computacional. El lenguaje de programación visual *Cordial*, integra los paradigmas de programación orientada a objetos, concurrente y por restricciones, toma como base formal el cálculo computacional *PiCO* [GF98] y utiliza la Máquina Abstracta *MAPiCO* para la ejecución de programas.

En pruebas realizadas a *Cordial*, se determinó que la etapa de ejecución conformada por la Máquina Abstracta *MAPiCO* y el Sistema de Restricciones, no arrojaba los resultados esperados en rendimiento, generando tiempos no deseables [GV01].

En el proyecto [GV01], se evaluó los dos componentes de la etapa de ejecución para determinar las posibles causas de los problemas de eficiencia. Se identificó que el Sistema de Restricciones Aritmético era ineficiente y presentaba problemas de no correctitud; las causas de ineficiencia se sustentaron en que la implementación de éste sistema se realizó en el lenguaje de programación *JAVA*, que por ser un lenguaje interpretado es mas lento que un lenguaje completamente compilado como *C*; Las causas de no correctitud, se sustentaron en que existían errores en la implementación del modelo de sistema de restricciones aritmético de *Björn Carlson*, en los procesos que involucran la condición 'diferente'y en la actualización de dominios de las variables cuando éstos son inconsistentes.

Inicialmente se implementó un Sistema de Restricciones Aritmético en lenguaje *C*, basado en restricciones de dominio finito; que garantiza la eficiencia y la correctitud. Posteriormente se determinó, que el tiempo de ejecución podría mejorarse implementando el Sistema de Restricciones Ecuacional y la Máquina Abstracta *MAPiCO* en un lenguaje no interpretado como *C*.

La Reimplementación de la Máquina Abstracta *MAPiCO* se hace necesaria, no solo para mejorar la eficiencia del tiempo de ejecución de *Cordial*, sino para adaptar su diseño con respecto a las modificaciones realizadas al cálculo *PiCO* y brindar extensibilidad del conjunto de instrucciones.

Este trabajo plantea una reimplementación de la máquina con las anteriores características, además de evaluar una alternativa para determinar la posibilidad de eliminación de variables locales de procesos ejecutados, las cuales no se necesitarán más a lo largo de la ejecución, mejorando así el tiempo de ejecución del Sistema de Restricciones con el cual interactua constantemente.

En el capítulo 1 se da una breve descripción de máquinas abstractas, sus ventajas y desventajas. En el capítulo 2 se describen algunos cálculos computacionales, haciendo enfásis en el cálculo *PiCO*, su definición anterior y actual. En el capítulo 3 se describe y analiza la primera implementación de la Máquina Abstracta *MAPiCO*, siguiendo con la presentación de alternativas de diseño para su reimplementación y terminando con la descripción de caracteristicas fundamentales de la nueva máquina. En el capítulo 4 se presentan detalles de la implementación, considerando estructuras de datos, alternativas evaluadas para la implementación de módulos dinámicos y complejidades. Por último en el capítulo 5 se describen algunas pruebas computacionales, el resultado de la ejecución y se realiza un análisis comparativo del desempeño de la nueva *MAPiCO* contra la versión anterior.

1 MÁQUINAS ABSTRACTAS Y VIRTUALES

Entre Máquinas Abstractas y Máquinas Virtuales existen diferencias que se evidencian en su definición. A continuación se presentan sus definiciones y varios ejemplos de máquinas representativas.

1.1. Máquinas Abstractas

El concepto de Máquina Abstracta hace referencia a portabilidad entre plataformas que permite elevar el poder de abstracción y desarrollar todos los elementos de un sistema integral. Una Máquina Abstracta define la arquitectura para la ejecución de programas de lenguajes que están soportados por un cálculo computacional [Mar01].

Las ventajas de utilizar una Máquina Abstracta son basicamente [ASS99]:

- Soporte formal.
- Es un excelente medio para alcanzar portabilidad.
- Permite separar claramente la compilación de la ejecución, es decir, se puede modificar el compilador o la Máquina Abstracta en forma independiente. De esta forma se puede mejorar la eficiencia definiendo nuevas arquitecturas para la Máquina Abstracta y extender el lenguaje fuente realizando modificaciones sobre el compilador.

La utilización de Máquinas Abstractas tiene inconvenientes con respecto al rendimiento, en un orden mas bajo que los intérpretes y más alto que los de código compilado. En definitiva esta pérdida de rendimento se ve compensada por los múltiples beneficios que la máquina ofrece [Mar01].

La Máquina Abstracta puede optimizarse y de esta forma minimizar la pérdida de rendimiento, una técnica es la compilación dinámica o justo a tiempo (JIT, Just In Time), que consiste en optimizar la interpretación del juego de instrucciones de la máquina en lugar de interpretarlas una a una, se realiza una compilación a las instrucciones nativas (del procesador convencional) del código de los métodos en el momento del acceso inicial de los mismos (justo a tiempo). Los siguientes accesos a ese método no son interpretados de manera lenta, si no que acceden directamente al código previamente compilado, sin pérdida de velocidad [1] [Mar01].

A continuación se presentan ejemplos de Máquinas Abstractas:

- **AMOz: Máquina Abstracta para el cálculo *Oz***

  Oz es un lenguaje que soporta programación declarativa, programación orientada a objetos, programación por restricciones y concurrencia como parte de un todo coherente [Moz06]; tiene un simple pero poderoso modelo computacional, el cual extiende el modelo de restricciones concurrentes por medio de procedimientos de primera clase, guardas profundas, estados concurrentes y búsqueda encapsulada [MC95].

  El concepto central del modelo es el de *espacio computacional* que consiste en una serie de agentes conectados a un espacio de trabajo, cada agente lee del espacio y una vez que tiene la información que esperaba se reduce y adiciona información al espacio de trabajo y crea mas procesos. Los agentes pueden tener uno o mas espacios computacionales [BH99].

  La semántica operacional del lenguaje *Oz* esta definida por un modelo matemático llamado el cálculo *Oz*. La noción básica de *Oz* es un espacio computacional.

[1] Este código nativo compilado puede ir siendo optimizado aún mas en cada llamada adicional, esta forma de optimización es llamada generación incremental de código

AMOz es la Máquina Abstracta para el Lenguaje *Oz*, es una máquina secuencial diseñada con registros y un conjunto de instrucciones tradicionales basada en un solo trabajador, pero con múltiples hilos de ejecución con schedule preemptively, solo un hilo puede ejecutarse a la vez. *AMOz* implementa árboles de relaciones para las restricciones, procedimientos de primera clase, guardas profundas e hilos [MC95].

- **PAMOZ: Máquina Abstracta Paralela para *Oz***

  PAMOz es la Máquina Abstracta Paralela para *Oz*, modela la ejecución de un sublenguaje para *Oz* sin la facilidad de solución de restricciones. *PAMOz* ha sido implementada en el sistema paralelo de *Oz*, el cual es derivado del sistema secuencial de *Oz* y de sus optimizaciones [Pop97].

  PAMOz esta destinada a memoria compartida y multiprocesos, ejecuta hilos de *Oz* en paralelo y es derivada de *AMOz*, la máquina secuencial para *Oz*. Existen dos principales diferencias entre *PAMOz* y *AMOz* básicamente en cuanto a la estructura y a la implementación de operaciones sobre datos *stateful*. *PAMOz* puede ser extendida completamente para *Oz* realizando una interfaz entre *PAMOz* y su extensión de solución de restricciones [Pop97].

- **Máquina Abstracta para el Cálculo *TyCO***

  El cálculo *TyCO*, Typed Concurrent Objects, plantea formalmente la interacción de objetos en una comunicación asíncrona utilizando mensajes, que junto a los objetos, son entidades fundamentales. La semántica de *TyCO* es una variación y extensión del cálculo π [BH99].

  La Máquina Abstracta de *TyCO* esta compuesta por dos áreas de memoria:

  - Estática: Memoria de almacenamiento de instrucciones de programas, canales locales, tablas de métodos y arreglos estáticos no inicializados.
  - Dinámica: Heap usado para estructuras como colas de ejecución y comuni-

cación.

La Máquina Abstracta de *TyCO* opera con dos tipos básicos internos: palabra (word) y marco (frame).

- **Máquina Abstracta para *Curry* en JAVA**.

  Curry es un lenguaje que combina dos paradigmas de programación declarativa: La programación lógica y la programación funcional, que constituyen la llamada programación lógica funcional.

  Este lenguaje utiliza una técnica denominada *narrowing* que generaliza la unificación de *Prolog* y contiene un sistema de tipos similar al del lenguaje *Haskell* [Mai00].

  La Máquina Abstracta para *Curry* en *JAVA* es un híbrido entre una Máquina Abstracta y un sistema de ejecución (o runtime). La compilación de *Curry* puede producir código *JAVA* que extiende el core de la Máquina Abstracta.

1.2. Máquinas Virtuales

Para una mejor comprensión de la definición de una Máquina Virtual se establecerá el significado de intérprete.

Un intérprete es un programa que ejecuta la secuencia de operaciones especificadas por un programador, en función de la descripción semántica de las instrucciones del lenguaje utilizado [JF04].

Una Máquina Virtual es un intérprete definido sobre una Máquina Abstracta, de esta forma la Máquina Abstracta es utilizada en la descripción semántica de las instrucciones [JF04]. En este sentido puede decirse que una Máquina Abstracta confiere formalidad a una Máquina Virtual en cuanto a la especificación semántica de las instrucciones del lenguaje utilizado.

Algunos ejemplos de Máquinas Virtuales son:

- **SECD**

  Máquina Virtual destinada a la programación funcional, fue desarrollada por *P. Landín* en 1966. Cada una de las letras de su denominación hace referencia a los registros internos de la máquina, S = Stack, E= Environment , C= Code y D= Dump, que apuntan a listas encadenadas en memoria.

  Posteriormente la implementación de lenguajes funcionales ha recurrido a la utilización de máquinas basadas en transformación de grafos como la máquina G [JF04].

- **P-CODE**

  Máquina Virtual basada en pila que interpreta un lenguaje intermedio para el lenguaje Pascal. El compilador de Pascal generaba código intermedio para la Máquina Virtual *P-CODE* y la traducción de *P-CODE* a código máquina se realizaba fácilmente, lo que le significó popularidad al lenguaje Pascal.

- ***JVM*: JAVA Virtual Machine**

  Máquina Virtual desarrollada para el lenguaje *JAVA*. A continuación se presentan algunas características de esta máquina [JF04]:

  - Pila de Ejecución: La *JVM* se basa en la utilización de una pila de ejecución y un repertorio de instrucciones que manipulan dicha pila.
  - Código Multi-hilo: La Máquina Virtual puede dar soporte a varios hilos (threads) de ejecución concurrente. Estos hilos ejecutan código de forma independiente sobre un conjunto de valores y objetos que residen en una memoria principal compartida.
  - Compilación JIT (Just In Time): Un programa compilado se representa mediante un conjunto de archivos de código de bytes (archivos class) que se cargan de forman dinámica y que contienen una especificación sobre el comportamiento de una clase.

- Verificación estática de código de bytes: Los archivos class contienen información sobre el comportamiento del módulo que puede verificarse antes de su ejecución. Es posible verificar estáticamente de que el programa no va a producir determinados errores al ejecutarse.
- Gestión de memoria dinámica: La máquina integra un recolector de basura.
- Dependencia del Lenguaje *JAVA*: La máquina ha sido diseñada para ejecutar programas *JAVA*, adaptándose fuertemente al modelo de objetos de *JAVA*. Incluye instrucciones de gestión de clases, interfaces, etc. Esta característica perjudica la compilación a *JVM* de lenguajes diferentes a *JAVA*.

- **CLR: Common Languaje Runtime o lenguaje de ejecución común**

  Entorno computacional desarrollado por Microsoft para la plataforma .NET. Utiliza el lenguaje intermedio denominado CIL (Comon Intermediate Languaje) y ofrece algunas características similares a la Máquina Virtual de *JAVA* como la utilización de una pila de ejecución, código multi-hilo, compilación JIT, verificación estática y gestión dinámica de memoria. Además de las anteriores, pueden destacarse [JF04]:

  - Independencia del lenguaje: *CLR* es utilizado como plataforma de ejecución de numerosos lenguajes de programación e incluye facilidades que permite desarrollar otros mecanismos de paso de parámetros e incluir tipos de datos primitivos en la pila de ejecución (boxing/unboxing). La plataforma ha sido adaptada como destino de lenguajes de diversos paradigmas como *Basic*, *Haskell*, *Mercory*, etc.
  - Sistema de Componentes: Un programa .NET se forma a partir de un conjunto de archivos assembly que se cargan de forma dinámica, éstos contienen la especificación de un módulo que puede estar formado por varias clases e incluyen especificación de control de versiones, firmas criptográficas, etc.

- **Máquina Virtual para el Cálculo NTCC (Non Deterministic Temporal Concurrent Constraint)**

  NTCC es un cálculo de procesos concurrentes para modelar no determinismo y asincronía en sistemas discretos temporales [AM04].

La Máquina Virtual para *NTCC* se realizó bajo la especificación formal de la Máquina Abstracta LMAN. Ésta última se determina operacionalmente mediante un conjunto de reglas de reducción relacionadas de manera directa con la semántica operacional de *NTCC*.

La Máquina Virtual se compone de instrucciones, registros y un modelo de memoria que interactúan con un Sistema de Restricciones para aritmética modular y un protocolo de comunicaciones *LEGO* [AM04].

A continuación en la tabla 1.1 se presenta un resumen de las definiciones de Máquina Abstracta y Máquina Virtual [JF04].

MÁQUINA	
ABSTRACTA	**VIRTUAL**
procedimiento para ejecutar un conjunto de instrucciones de un lenguaje formal. Estructura sobre la cual es posible representar la semántica computacional de un lenguaje	Intérprete definido sobre una máquina abstracta, de esta forma la máquina abstracta es utilizada en la descripción semántica de las instrucciones

Tabla 1.1: Definición de Máquinas Abstractas y Virtuales

2 CÁLCULOS COMPUTACIONALES

La aplicación de métodos y lenguajes formales son de gran utilidad en la especificación y validación de propiedades en etapas de diseño y desarrollo de sistemas. Los formalismos como el cálculo π, se utilizan en el desarrollo de lenguajes de alto nivel y herramientas que sirven de soporte al desarrollo de sistemas software que acortan la distancia entre los fundamentos formales y su aplicación práctica.

A continuación se presentan dos cálculos computacionales que guardan relación en su definición: *TyCO* y *PiCO*, éste último es la base formal de la Máquina Abstracta *MAPiCO*.

2.1. TyCO

TyCO [Vas94] cuenta con procesos básicos definidos en el π-Cálculo, pero su comunicación no se establece a través de canales, sino mediante el mecanismo de paso de mensajes entre objetos. Los objetos se localizan en nombres y los mensajes especifican con el nombre al cual se envian.

Este tipo de comunicación cuenta con los procesos básicos de replicación, composición y restricción de un nombre a un contexto y dos procesos adicionales para crear objetos y enviar mensajes [DR01].

2.1.1. Sintáxis

Los elementos sintácticos de *TyCO* se presentan en la Tabla 2.1:

nombres:	$a, b, ...$
	$v, x, y, ...$
etiquetas:	$l_1, ..., l_n$
procesos:	P, Q
agentes:	$A, B, ...$
variables agentes:	X,Y,...
colecciones de métodos:	M, N

Tabla 2.1: Simbolos Sintácticos de *TyCO* - [GIAD97]

La sintaxís de *TyCO* esta representada por:

- $x_1, ..., x_n$: Hace referencia a una secuencia de nombres $\widetilde{x}$.
- $P[a/x]$: Representa sustitución, reemplazar todas las ocurrencias de x en P por a.
- En *TyCO* pueden existir procesos y agentes. Un proceso es una expresión que esta definida en los siguientes términos:
  - $a \triangleleft l : \widetilde{v}$: Expresión que representa el envio del mensaje l al objeto a con párametros actuales $\widetilde{v}$.
  - $a \triangleright [l_1 : A_1, ..., l_n : A_n]$: Representa la definición de un objeto, el cual consta de un conjunto de métodos llamados l_i $(0 \leq i \leq n)$ y cuyo cuerpo es A_i respectivamente.
  - $P \mid Q$: Representa la ejecución concurrente de procesos.
  - $(v\ x)P$: Expresión que hace referencia al alcance de las variables, en este caso, el alcance de la variable x está limitado al proceso P.
  - $A(\widetilde{v})$: Representa la generación de una instancia del agente A y los argumentos formales serán reemplazados por los actuales $A[\widetilde{v}/\widetilde{x}]$.
  - $X(\widetilde{v})$: Si una variable X está instanciada por algún agente A estará definida como $A(\widetilde{v})$, si no, la expresión no se reduce.
  - $let\ x = A\ in\ P$: Esta expresión representa una variable X instanciada por el agente A en P y permite el uso repetido de A en P a través de X.
- A: Representa un agente, proceso abstraido de una serie de nombres.

A continuación se muestra en la Tabla 2.2 un resumen de la sintáxis de procesos y agentes en *TyCO*:

P	$::= a \triangleleft l : \widetilde{v}$	Mensaje
	$\mid a \triangleright [l_1 : A_1, ..., l_n : A_n]$	Objeto
	$\mid P \mid Q$	Concurrencia
	$\mid (v\ x)P$	Variable local
	$\mid X(\widetilde{v})$	reescritura de agente
	$\mid A(\widetilde{v})$	reescritura de agente
	$\mid let\ x = A\ in\ P$	intanciación local
A	$::= (\widetilde{v})P$	
	$\mid rec\ X.A$	

Tabla 2.2: Sintáxis de Procesos y Agentes en *TyCO* - [GIAD97]

2.1.2. Reglas de Congruencia *TyCO*

La congruencia estructural $\equiv$ sirve para simplificar la reducción y se define como la relación de congruencia más pequeña sobre los procesos generados con las siguientes reglas (ver Tabla 2.3):

$P \equiv Q$	Si P es $\alpha - convertible\ en\ Q$
$P \mid Q \equiv Q \mid P$	
$P \mid (Q \mid R) \equiv (P \mid Q) \mid R$	
$a \triangleright M \equiv a \triangleright N$	Si M es permutación de N
$(vx)P \mid Q \equiv (vx)(P \mid Q)$	Si x no pertenece a $fn(Q)$
$(\widetilde{x})P(\widetilde{v}) \equiv P[\widetilde{v}/\widetilde{x}]$	Si $\mid \widetilde{x} \mid = \mid \widetilde{v} \mid$
$recX.A \equiv A[recX.A/X]$	
$let\ X = A\ in\ P \equiv P[A/X]$	
$*P \equiv P \mid *P$	

Tabla 2.3: Reglas de Congruencia *TyCO* [GIAD97]

$P \equiv Q$ significa que Q puede obtenerse haciendo unicamente renombramiento de variables a P; $fn(Q)$ es el conjunto de los nombres libres en Q; $|x| = |v|$ determina si la longitud de la secuencia x es igual a la longitud de la secuencia v.

2.1.3. Reglas de Reducción

La reducción de un paso $P \rightarrow Q$ es la mínima relación generada por las siguientes reglas [GIAD97]:

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

$$COMM\text{: } (v\ \widetilde{x})(a \triangleleft l_i : \widetilde{v}_i \mid a \triangleright [l_1 : A_1, ..., l_n : A_n] \;\mid \widetilde{P}) \;\Rightarrow (v\ \widetilde{x})(A_i[\widetilde{v}_i/\widetilde{x}_i] \;\mid \widetilde{P})$$

El sistema de reglas de reducción de *TyCO* cuenta con dos reglas: La primera hace reducción estructural y la segunda reducción cuando un objeto recibe un mensaje que va dirigido a él. Algunas reglas que podrían considerarse de reducción aparecen como reglas de congruencia.

La regla $COMM$ expresa el hecho que, dos procesos uno objeto y otro mensaje sobre el mismo, reducen a la aplicación del método solicitado por el mensaje, adicionalmente otros procesos que pudieran existir concurrentemente (representados por $\widetilde{P}$), continuan existiendo una vez que la reducción se realiza [GIAD97].

2.2. Cálculo PiCO: Cálculo de Objetos y Restricciones

El Cálculo *PiCO* [GF98] introduce la noción de restricción combinada ortogonalmente con la noción de objeto y paso de de mensajes como mecanismo de comunicación. Es un cálculo concurrente de Objetos y Restricciones, diseñado como cálculo base para ayudar a construir composiciones musicales.

Éste cálculo pretende mantener lo más posible la independencia entre los modelos de objetos y restricciones al nivel del cálculo [BH99].

La interacción con objetos puede modelarse a través de la noción de objetos concurrentes donde la ejecución sincronizada es simulada por cambios en los objetos reales y por medio del uso de restricciones para "cambiar" el estado del objeto redefiniendo el

conocimiento parcial que se tiene de cada atributo de los objetos [BH99].

El Cálculo *PiCO* desde su primera definición ha tenido una serie de cambios para alcanzar una mayor expresividad y flexibilidad dentro de su especificación [GG99].

Inicialmente se expondrá su primera definición considerando la sintásis y la semántica, posteriormente se hará una descripción de sus modificaciones que hacen parte del cálculo *PiCO* actual.

2.2.1. Cálculo PiCO - Definición Anterior

La primera definición del Cálculo *PiCO*, su sintáxis y semántica estan contemplada en [GF98].

2.2.1.1. Sintáxis

Como se observa en la Tabla 2.4 existen tres tipos básicos de procesos: Mensajes, Objetos y Restricciones, en donde:
El proceso O representa el proceso nulo.

El proceso $I \triangleleft m.P$ representa un objeto direccionado por I, localizado en I o idenficado por I y m identifica el mensaje que se invoca.

El proceso $(I, J) \triangleright M$ representa el proceso de reenvío, en el cual M es una colección de métodos $l_i : (\widetilde{x_1})P_1 \& ... \& l_m : (\widetilde{x_m})P_m$, identificados por un conjunto de nombres distintos $Labels(M) = l_1...l_m$ y el identificador J es la dirección donde el mensaje debe ser reenviado si no existe el método apropiado al realizar un llamado. Cuando $I = J$ no existe condición de reenvío.

En el método $l : (\widetilde{x})P$, $\widetilde{x}$ representa los parámetros formales y P el cuerpo del método. Los nombres, valores y variables se pueden usar como identificadores de objetos.

El proceso de comunicación tiene lugar cuando un mensaje es dirigido al objeto I con argumentos $\widetilde{K}$ $(I \triangleleft : [\widetilde{K}]P)$. Para definir la continuación de la ejecución del proceso P,

se debe seleccionar el método correspondiente en el objeto al cual se dirige el mensaje, reemplazar los parámetros formales por sus correspondientes argumentos actuales contenidos en el mensaje, siempre y cuando el método seleccionado exista en la definición del objeto; de lo contrario, el mensaje es reenviado a la dirección de reenvío (si esta declarada tal dirección en el objeto).

El proceso $(va)P$ restringe el uso del nombre a al proceso P o declara un único nuevo nombre a distinto de todos los nombres externos que son usados en P. Igualmente $(vx)P$ 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 replicación $*P$ expresa la composición $P|P...$ tantas copias como sean necesarias como en el cálculo π [DR01]. $*P = P \mid *P$.

Los procesos de restricciones son tipos de procesos cuyo comportamiento depende de un sistema de restricciones, conformado por un store que contiene información propocionada por las restricciones y un par de operadores usados para agregar información y preguntar por ella; de esa forma el store controla todas las comunicaciones posibles [GV01].

El proceso tell $!\phi.P$, impone la restricción ϕ al sistema de restricciones y luego continua con P. El proceso ask $?\phi.P$, ejecuta el proceso P si la restricción ϕ es una consecuencia lógica de la información almacenada en el sistema de restricciones o elimina a P si $\neg\phi$ lo es. En caso contrario, suspende el proceso $?\phi.P$ hasta que el store contenga suficiente información para reducirlo.

2.2.1.2. Semántica

PiCO en su semántica operacional esta parametrizado en un sistema de restricciones, el cual para la especificación de sus sentencias, utiliza los predicados de primer orden. Los elementos que hacen parte de la semántica son la congruencia estructural y la equivalencia entre configuraciones.

Procesos Normales: N	$::= O$	Proceso nulo
	$\mid I \triangleleft m.P$	Mensaje enviado a I.
	$\mid (I, J) \triangleright M$	Objeto I con condición de reenvío
Procesos Restricciones:R	$::= !\phi.P$	Proceso Tell
	$\mid ?\phi.P$	Proceso Ask
Procesos P, Q	$::= (vx)P$	Variable nueva x en P
	$\mid (va)P$	Nombre nuevo a en P
	$\mid N$	Proceso Normal
	$\mid P \mid Q$	Composición
	$\mid *P$	Proceso replicado
	$\mid R$	Proceso Restricción
Objeto identificador: I,J,K	$::= a$	Nombre
	$\mid v$	valor
	$\mid x$	variable
Colección de Métodos M	$::= [l_i : (\widetilde{x_1})P_1 \& ... \& l_m : (\widetilde{x_m})P_m]$	
Mensajes m	$::= l : [\widetilde{I}]$	

Tabla 2.4: Sintáxis Cálculo PiCO - Definición Anterior [GF98]

La semántica operacional esta 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 $\rightarrow$, describiendo las transiciones de estas configuraciones. Una configuración es la tupla $\langle P; S\rangle$, considerando el proceso P y el store S.

El comportamiento de un proceso P se define por transiciones sobre la configuración inicial $\langle P; \top\rangle$; una transición $\langle P; S\rangle$ puede ser transformada a una $\langle P'; S'\rangle$ en un solo paso y se denota así: $\langle P; S\rangle \rightarrow \langle P'; S'\rangle$.

La relación de reducción esta definida sobre configuraciones y se aplica si al menos satisface una de las siguientes reglas:

$COMM$: Describe el proceso de comunicación entre un mensaje $I' \triangleleft l : [\widetilde{K}].Q$ y un objeto $(I, J) \triangleright [l : (\widetilde{x})P...]$ utilizando al store como elemento de decisión para tal fin. Para que la regla se aplique $I = I'$ se debe deducir del store y el número de parámetros debe cumplir que $|\widetilde{K}| = |\widetilde{x}|$. El proceso empieza con el reemplazo de los parámetros formales $\widetilde{x}$ por los argumentos actuales $\widetilde{K}$ ($P\{\widetilde{K}/\widetilde{x}\}$); continuando con la activación del mensaje Q cuando este sea recibido.

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

$DELEG$: Establece el proceso de delegación-reenvío. Cuando un mensaje $I \triangleleft l : [\widetilde{K}].Q$ es enviado a un objeto $(I, J) \triangleright M$ y el método requerido l no hace parte de los métodos del Objeto, el mensaje es enviado por reenvío al objeto direccionado por J (esta dirección de reenvío esta especificada por el objeto).

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

ASK y $TELL$: Describen la forma de interactuar con el sistema de restricciones, Store S. Ask es el método para obtener información del store y dependiendo de esa información activar/eliminar el proceso P o suspender el proceso hasta que haya información suficiente para reducir la restricción ϕ.

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

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

PAR : Describen la reducción dentro de una composición paralela.

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

$DEC-N$ y $DEC-V$: Son la manera de introducir nuevos nombres y nuevas variables.

$fv(S)$: Define el conjunto de variables libres en S $fn(S)$: Define el conjunto de nombres libres en S

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

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

$EQUIV$: Expresa la equivalencia de reducción entre configuraciones P-equivalentes.

$$EQUIV\text{: } \frac{\langle P_1;S_1\rangle\equiv P\left\langle P_1';S_1'\right\rangle \qquad \langle P_2;S_2\rangle\equiv\left\langle P_2';S_2'\right\rangle \qquad \langle P_1;S_1\rangle\longrightarrow\langle P_2;S_2\rangle}{\left\langle P_1';S_1'\right\rangle\longrightarrow\left\langle P_2';S_2'\right\rangle}$$

2.2.2. Cálculo PiCO - Definición Actual

Al Cálculo $PiCO$ se le hicieron una serie de modificaciones con respecto a la noción de localización de objetos; en la definición anterior los objetos se localizaban en identificadores, en la actual, los objetos se localizan en restricciones [GG99].

La localización de objetos en restricciones hace referencia a que cada objeto receptor define explicitamente su tipo de comunicación localizandose en una restricción $\phi(sender)$ parametrizada en una variable $sender$, así mismo cada objeto emisor se localiza en una variable o nombre. De esta forma los posibles valores que toma la variable que cumplen

con la restricción establecen el tipo de comunicación del emisor del mensaje y el conjunto de posibles emisores definen el tipo de comunicación del receptor.

Debido a lo anterior, se puede aumentar el número de posibles receptores de un mensaje adicionando restriciones sobre su localización, al igual que aumentar el conjunto de potenciales emisores añadiendo restricciones sobre las variables, distintas de *sender*, de su restricción de localización.

La comunicación entre un emisor y receptor solo puede establecerse si de la información global almacenada en forma de restriciones, se puede inferir que el tipo de comunicación del emisor de un mensaje es un subtipo del tipo de comunicación del objeto receptor.

La definición actual también comtempla el reenvío de mensajes, que consiste en el reenvío de mensajes por parte del objeto receptor cuando el objeto emisor cumple uno de los requisitos para la comunicación: el tipo de comunicación del emisor es igual al tipo de comunicación del receptor; pero el emisor solicita un servicio no provisto por el receptor.

Este tipo de comunicación para reenvío también se define por medio de una restricción. Reenviar consiste entonces en "mover" el tipo del emisor del mensaje al tipo reenviado [DR01].

2.2.2.1. Sintáxis

Existen tres tipos básicos de procesos: Mensajes, Objetos y Restricciones (ver Tabla 2.5).

Como se puede notar, en el Cálculo *PiCO* actual se conservan bajo la misma especificación el proceso nulo, los objetos identificadores, las colecciones de métodos y los mensajes. Los procesos de creación de variables/nombres, de composición y de replicación guardan similitud con su representación, sin embargo sufrieron algunos cambios en su denotación.

El cambio mas representativo, como se mecionó anteriormente, es en la definición de los procesos objetos, localizados en restricciones: $(\phi_{sender}, \delta_{forward}) \triangleright M)$ es un objeto receptor de mensajes que responde mensajes enviados por otros procesos, reemplazan-

Procesos Normal: N	$::= O$	Proceso nulo
	$\mid I \triangleleft m\ then\ P$	Mensaje enviado al objeto I
	$\mid (\phi_{sender}, \delta_{forward}) \triangleright M$	Objeto con condición de reenvío $\delta_{forward}$ guardada por la restricción ϕ
Procesos Restricción: R	::= tell ϕ then P	Proceso Tell
	| ask ϕ the P	Proceso Ask
Procesos P, Q	::= Local x in P	Nueva variable x en P
	| Local a in P	Nuevo nombre a en P
	| N	Proceso Normal
	| $P \mid Q$	Composición
	| clone P	Proceso replicado
	| R	Proceso Restricción
Identificadores de Objetos: I,J,K	::= a, l	Nombres
	| v	valor
	| x	variable
Colección de Métodos M	::= $[l_i : (\widetilde{x_1})P_1 \& ... \& l_m : (\widetilde{x_m})P_m]$	
Mensajes m	::= $l : [\widetilde{I}]$	

Tabla 2.5: Sintáxis Cálculo PiCO - Definición Actual [GG99]

dose por otro proceso o método, que es elegido por el mensaje que tiene la información necesaria para seleccionarlo dentro de la colección de métodos del objeto.

Los procesos mensaje siguen la misma especificación: $I \triangleleft m\ then\ P$ es un proceso emisor de mensajes localizado en un identificador, nombre, variable o valor. Cada proceso receptor cuenta con la restricción ϕ_{sender} que determina si puede comunicarse con el proceso emisor. Por ejemplo, el objeto $(sender \in \{a, b\},\ forward \in \{b, c\}) \triangleleft M$ acepta mensajes provenientes del proceso $a \triangleright m[\widetilde{x}]\ then\ Q$ porque se cumple $a \in \{a, b\}$.

El reenvío de mensajes tiene las siguientes variaciones que se determinan por el resultado de evaluar la restricción ϕ_{sender} y la correspondencia del servicio solicitado como requisito de la comunicación

Reenvío de mensajes: Cuando en el proceso emisor $a \triangleleft m[\widetilde{x}]\ then\ Q$, el mensaje m no hace parte de los servicios ofrecidos por el receptor ($m \notin M$), este proceso es reemplazado por un nuevo proceso emisor $J \triangleleft m[\widetilde{x}]\ then\ Q$, localizado en una variable J; la cual es restringida imponiendo la restricción $\delta_{forward}$. Posteriormente un proceso receptor cuya restricción de localización sea satisfecha por J puede responder el mensaje m.

Al igual que en la definición anterior del Cálculo PiCO, del *store* depende el comportamiento de los procesos de restricciones *tell* y *ask* y el control de todas las comunicaciones posibles.

2.2.2.2. Semántica

La semántica de la definición actual del Cálculo *PiCO* sigue el mismo esquema de la definición anterior, las relaciones de congruencia y reducción se definen de la misma forma, solo que se considera la localización de objetos en restriciones ϕ_{sender} y $\delta_{forward}$ y el reenvío de mensajes mediante la imposición de la restricción $\delta_{forward}$ sobre una nueva variable.

La relación de congruencia se define de la siguiente forma:

$(\phi_{sender}, \delta_{forward}) \triangleright M \ \equiv\ (\phi_{sender}, \delta_{forward}) \triangleright M'$, si M' es una permutación de M.

$tell\ \phi\ then\ P\ \equiv\ tell\ \psi\ then\ Q$
$ask\ \phi\ then\ P\ \equiv\ ask\ \psi\ then\ Q$ Si $\phi\ \models \triangle\ \psi\ y\ P \equiv\ Q$ (la notación $\phi\ \models \triangle\ \psi$ denota la equivalencia lógica de las restriciones ϕ y ψ).
Las reglas de reducción que representan la semántica de $PiCO$ se definen con base en configuraciones de la forma $\langle P; S\rangle$, donde P es un proceso y S es el almacén o *Store*.

En $PiCO$ el comportamiento de un proceso P se define por transiciones sobre la configuración inicial $\langle P; \top\rangle$; una transición $\langle P; S\rangle$ puede ser transformada a una $\langle P'; S'\rangle$ en un solo paso y se denota así: $\langle P; S\rangle \rightarrow \langle P'; S'\rangle$

La relación de reducción varía con respecto en la definición de objetos a través de las restricciones ϕ_{sender} y $\delta_{forward}$. Se realizan reemplazos de las variables *sender* y $forward$ en cada definición y comunicación de objetos:

La regla $COMM$ define la comunicación entre un proceso emisor $(I' \triangleleft l : [\widetilde{K}]\ then\ Q)$ y un proceso receptor $((\phi_{sender}, \delta_{forward}) \triangleright [l : (\widetilde{x})P\&...])$ utilizando como elemento de decisión la información almacenada en el store. De deducirse ϕ_{sender} se activa el proceso asociado al objeto emisor P, en paralelo con el proceso definido como continuación del proceso emisor Q.

$$COMM\colon \frac{S \vdash \Delta\phi[I'/Sender] \qquad |\widetilde{K}|=|\widetilde{x}|}{\left\langle I' \triangleleft l:[\widetilde{K}]\ then\ Q\ \mid\ (\phi sender, \delta forward) \triangleright [l:(\widetilde{x})P\&...]; S\right\rangle \longrightarrow \left\langle Q\ \mid P\{\widetilde{K}/\widetilde{x}, I'/sender\}; S\right\rangle}$$

La regla REE describe formalmente como se delega/reenvia el procesamiento de los mensajes; las condiciones impuestas para el reenvío evitan que el procesamiento del mensaje lo haga un proceso idéntico al original. Así se evitan ejecuciones infinitas por reenvío.

$$REE\colon$$
$$\frac{S \vdash \Delta\phi[I'/Sender] \qquad S \sqcup \delta[I'/forward] \vdash_{\Delta} \perp \qquad l \notin Labels(M)}{\left\langle (I' \triangleleft l:[\widetilde{K}]\ then\ Q \mid\ (\phi sender, \delta forward) \triangleright M); S\right\rangle \longrightarrow \left\langle (local\ J\ in\ tell\ \delta[J/forward]\ then\ (J \triangleleft l:[\widetilde{K}]\ then\ Q) \mid (\phi sender, \delta forward) \triangleright M); S\right\rangle}$$

Las reglas PAR, $DEC-V$, $DEC-N$ y $EQUIV$ se definen de la misma forma que en la definición anterior.

$$TELL\text{: } \langle tell\ \phi\ then\ P; S\rangle \longrightarrow \langle P; S \wedge \phi\rangle$$

$$ASK\text{: } \frac{S \vdash \Delta\phi}{\langle ask\ \phi\ then\ P; S\rangle \longrightarrow \langle P; S\rangle} \quad , \quad \frac{S \vdash \Delta\neg\phi}{\langle ask\ \phi\ then\ P; S\rangle \longrightarrow \langle O; S\rangle}$$

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

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

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

$$EQUIV\text{: } \frac{\langle P_1; S_1\rangle \equiv P\left\langle P'_1; S'_1\right\rangle \qquad \langle P_2; S_2\rangle \equiv \left\langle P'_2; S'_2\right\rangle \qquad \langle P_1; S_1\rangle \longrightarrow \langle P_2; S_2\rangle}{\left\langle P'_1; S'_1\right\rangle \longrightarrow \left\langle P'_2; S'_2\right\rangle}$$

A continuación se mencionan en forma de resumen, las diferencias más representativas entre la definición anterior y actual del Cálculo $PiCO$ (ver tabla 2.6).

Diferencia en	*CÁLCULO PiCO*	
	Definición Anterior [GF98]	**Definición Actual [GG99]**
Sintáxis	Los objetos estan localizados en indentificadores: $(I, J) \triangleright M$ Objeto localizado en I con condición de reenvío J. J es el identificador de la dirección de reenvío. I/J pueden ser nombres o variables.	Los objetos estan localizados sobre restricciones: $(\phi_{sender}, \delta_{forward}) \triangleright M$ Objeto con reenvío, la restricción ϕ_{sender} debe ser satisfecha por una localización (variable *sender*) en el momento que se envian mensajes al objeto. La restricción $\delta_{forward}$, condición de reenvío, respresenta la guarda de los procesos que podrían ser reenviados cuando no existe un método apropiado para comunicarse.
Semántica	La definición de objetos se realiza a través de identificadores	La relación de reducción varia con respeto al cálculo anterior en la definición de objetos a través de restricciones ϕ_{sender} y $\delta_{forward}$. Se realizan reemplazos de las variables *sender* y $forward$ en cada definición y comunicación de objetos.
	$(I, I) \triangleright M$: Mensaje sin reenvío. En casos de mensajes sin condición de reenvío, se reenvia al mismo identificador que lo envia I	$clone\ (sender > 0, forward = forward)$: En el cálculo actual todas las comunicaciones deben tener condición de reenvío, en este caso la restricción que debe evaluarse es $\delta_{forward} \equiv (forward = forward)$.

Tabla 2.6: Comparación Cálculo *PiCO* Anterior y Cálculo *PiCO* Actual

3 MAPiCO: Máquina Abstracta para el Cálculo PiCO

3.1. Diseño e Implementación Inicial de MAPiCO

La primera implementación de la Máquina Abstracta *MAPiCO* [BH99] se realizó con base en la especificación y reglas de reducción de la primera definición del Cálculo *PiCO* [GF98].

Cuenta con las siguientes características:

1. Dos areas de Memoria:
   - Memoria Estática: Almacena las instrucciones
   - Memoria Dinámica: Utilizada para tener las traducciones o ligaduras de variables y nombres al *Store*. Almacena los parámetros para el llamado de métodos y es donde se crea el árbol sintáctico que describe a las restricciones.
2. Cuatro Colas:
   - RunQ: Cola de procesos listos para ejecución (**Rq**).
   - ObjQ: Cola de objetos suspendidos (**Oq**).
   - MsgQ: Cola de mensajes suspendidos (**Msq**).
   - AskQ: Cola de procesos ask suspendidos (**Aq**).
3. Un *store* (S): Almacenamiento de la información global (restricciones). Este *store* no hace parte de *MAPiCO*, pero la máquina interactua activamente con él.

4. Procesos: Estructuras conformadas por:

   - **PC**: Puntero al código de instrucciones
   - **PV**: Puntero a variables. Apunta al primer elemento en la memoria de traducción de la lista de ligaduras de variables dentro del *Store*.
   - **PN**: Puntero a nombres. Apunta al primer elemento en la memoria de traducción de la lista de ligaduras de nombres dentro del *Store*.
   - **PA**: Puntero auxiliar. Tiene dos usos:
     a). Apunta a una lista con los argumentos y al objeto receptor al darse una comunicación.
     b). Guarda referencia a nodos en la creación de un árbol sintáctico que define una restricción.

5. Registros Internos: Son registros de uso específico que guardan la información de la memoria y procesos. La máquina modifica esta información para actualizar procesos y colas.

   - **PCA**: Puntero a código actual. Apunta a la instrucción actual que se este ejecutando en la máquina.
   - **PVA**: Puntero a variable actual. Apunta al PV del proceso en ejecución.
   - **PNA**: Puntero a nombre actual. Apunta al PN del proceso en ejecución.
   - **PAA**: Puntero auxiliar actual. Apunta al PA del proceso actual del proceso que se esta ejecutando.
   - **PAUX**: Puntero auxiliar. Utilizado para manipulación y ejecución de procesos.

3.1.1. Especificación Formal de la Máquina

Se realizó una simplificación en la primera definición del Cálculo *PiCO* para mejorar la eficiencia de la máquina. Se restringió el operador de replicacion $*P$, haciendo que P solo pueda ser un proceso normal (Ver definición en [BH99]).

3.1.2. Reglas de Reducción de la Máquina

Estas reglas siguen la semántica operacional del Cálculo *PiCO* (ver sección 2.2.1.2). Se especifican reglas de transición entre configuraciones y estados. A continuación se hará una descripción conceptual de cada una de las reglas definidas en la primera implementación; la descripción formal completa se muestra en [BH99].

Sched: Permite la continuación de los procesos en ejecución, removiendo el proceso actual y adicionando el siguiente proceso en la cola de ejecución para reducirlo.

$Newref$: Permite la creación de nuevas referencias de nombres/variables ejecutadas por el sistema.

Parallel: Describe la composición paralela $P|Q$.

Tell: Describe la reducción del proceso tell $!\phi P$

Tomando en consideración la restricción $\phi(\widetilde{L})$ en la reducción del proceso ask $?\phi P$, se establecen tres posibles casos:

- $Ask1$: Hace referencia al caso en el que del *Store* se deduce $\phi(\widetilde{L})$ continuando con la ejecución del proceso P.
- $Ask2$: Caso en el que del *Store* se deduce $\neg\phi(\widetilde{L})$, eliminando el proceso actual.
- $Ask3$: Si del *Store* no se puede deducir $\phi(\widetilde{L})$ ni $\neg\phi(\widetilde{L})$, el proceso es suspendido y pasado a la cola $AskQ$.

$MsgEnq$: Hace referencia a la suspensión de mensajes debido a la reducción del paso de mensajes a un objeto que no se encuentra en la cola $ObjQ$.

$MsgComm$: Describe la comunicación de los procesos mensaje cuando hay un proceso objeto en la cola $ObjQ$ que cumple con los requisitos para la comunicación.

$MsgDel$: Consiste en la delegación/reenvío de procesos mensaje, cuando existe el proceso objeto para comunicarse, pero no existe el método requerido.

3.1.3. Conjunto de Instrucciones de MAPiCO

Se definió un conjunto de instrucciones clasificados en tres grupos: el primero para la manipulación de procesos (Tabla 3.1), el segundo para la definición de objetos (Tabla 3.2) y el último para la construcción de predicados de primer orden (Tabla 3.4).

Estas instrucciones estaban representadas por un código de operación y de 0 a 3 parámetros. Las tablas 3.1 y 3.2 guardan la siguiente representación:

- opcode: Código de operación de la instrucción, 8 bits.
- dir: Dirección valida en la memoria del programa, 32 bits.
- ind: Índice dentro de alguna de las listas apuntadas por PV, PN, 16 bits.
- num: Número constante. Para objetos especifica el número de métodos; para métodos y mensajes especifica su nombre o referencia, 16 bits.

Forma de realizar un llamado (envío de mensajes): Luego de una instrucción *replicate* siempre debe ir un *call* y antes de hacer un *call* o un *replicate*, el usuario siempre debe especificar el objeto receptor con *pushn* o *pushv*.

Los predicados de primer orden son los que especifican las restriciones en *PiCO*. Éstos estan definidos según la sintáxis de la Tabla 3.3 y son modelados en la máquina con las instrucciones de la Tabla 3.4. Según esto:

- opcode: Código de operación de la instrucción, 8 bits.
- ind: Índice 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 átomo, 8 bits.
- ari: Aridad de la función o el átomo, 8 bits.

Instrucción	**Opcode**	**Src1**	**Descripción**
ret	0		Es el proceso NIL en el cálculo
par	1	dir	Crea un nuevo porceso apuntado por Src1 para er ejecutado en paralelo
newv	2		Definición de una nueva variable.
newn	3		Definición de un nuevo 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 Scr1 en un objeto igual (en el *Store*) a la variable o nombre enla cabeza del PA.
tell	24		Adiciona al *store* la restricción apuntada por PA.
ask	25		Realiza la operación ask al *store* de una restricción apuntada por PA.

Tabla 3.1: Instrucciones para la Manipulación de Procesos en la Máquina - Versión Anterior

Instrucción	Opcode	Src1	Src2	Src3	Descripción
objvv	64	ind	ind	num	Objeto con nombre Scr1, delegación Src2 y con Src3 métodos. (Sc1 Src2 son indices dentro del PV)
objnn	65	ind	ind	num	Igual a objvv pero Sc1 y Src2 son nombres dentro del PN
objvn	66	ind	ind	num	Igual a objvv pero Sc1 es una variable dentro del PV y Src2 un nombre dentro del PN
objnv	67	ind	ind	num	Igual a objvv pero Sc1 es un nombre dentro del PN y Src2 una variable dentro del PV
objvvr	72	ind	ind	num	Igual a objvv pero clonado
objnnr	73	ind	ind	num	Igual a objnn pero clonado
objvnr	74	ind	ind	num	Igual a objvn pero clonado
objnvr	75	ind	ind	num	Igual a objnv pero clonado
meth	80	num	dir		Especifica que el métdo Scr1 esta en la dirección Src2

Tabla 3.2: Instrucciones para la Definición de Objetos de la Máquina - Versión Anterior

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 | Jhon | ...
Variable	→	a | x | s | ...
Predicado	→	Antes | En | ...
Función	→	Madre | mínimo | ...

Tabla 3.3: BNF para la especificación de Predicados de Primer Orden - Versión Anterior

- con: Código del conector, las opciones son *and* (0), *or* (1), 8 bits.
- cuan: Código del cuantificador o negación, 8 bits.

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 del PV
termn	34	ind		Término nombre dentro de una fórmula de primer orden. Src1 es la posición dentro del PN
termf	35	fun	ari	Función. Src1 es el código del átomo y src2 es la aridad.
atom	36	pred	ari	Atomo. Scr1 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.
setenq	41	cuan		Sentencia con un cuantificador o negación. Src1 es el código del cuantificador o de la negación.

Tabla 3.4: Instrucciones para construir Predicados de Primer Orden en la Máquina - Versión Anterior

3.2. Diseño Actual

En la sección anterior se realizó una breve descripción de cada uno de los componentes y características del diseño de la primera implementación de la máquina abstracta *MAPiCO*.

En esta sección se describirá las características de la nueva implementación, presentando cada una de las alternativas de diseño, los componentes de la arquitectura, las reglas de reducción con base en las modificaciones del cálculo *PiCO* y la definición del conjunto de instrucciones. El diseño de la nueva *MAPiCO* guarda estrecha relación con la definición anterior, tomando elementos esenciales que hacen parte de la correcta y formal funcionalidad de una máquina abstracta para el cálculo computacional *PiCO*.

3.2.1. Alternativas de Diseño

Para establecer el nuevo diseño de *MAPiCO* se evaluaron varias alternativas que pretendían seguir las reglas de reducción de la máquina con base en las modificaciones del cálculo *PiCO* y de alguna manera permitir una implementación genérica y flexible.

Anteriormente en el Cálculo *PiCO*, los objetos estaban definidos con identificadores, en ese caso durante la ejecución los objetos eran seleccionados de las pilas de variables y nombres y de esta forma era posible la comunicación (Ver sección 2.2.1.1).

$(I, J) \triangleright M$; Objeto I con condición de reenvío a J. I, J Nombres o Variables

En el cálculo actual los objetos se localizan en restricciones ϕ_{sender} y $\delta_{forward}$, parametrizadas en variables *sender* y $forward$, lo que hace necesario su evaluación para determinar una posible comunicación.

$(\phi_{sender}, \delta_{forward}) \triangleright M$; Objeto con condición de reenvío

En el diseño de la nueva máquina se tuvo en consideración la actual definición de los objetos, por esta razón las restricciones se evalúan como si fueran funciones, pero no funciones que hacen parte de la sintáxis del cálculo, sino, como funciones de evaluación de las variables *sender* y $forward$ respectivamente y la comunicación entre objetos depende del resultado de dicha evaluación.

Por ejemplo si se tiene la definición del siguiente programa en cálculo *PiCO*, que describe la función de potencia al cuadrado, siendo el primer parámetro el argumento y el segundo parámetro el resultado:

$$
\begin{aligned}
(\mathit{local}\ X,\ Y \mathit{in}\ & (\mathit{clone}\ (\mathit{sender} > 0,\ \mathit{forward} \leq 0)\ \triangleright\ [Sqr(n, r) :\ (\mathit{tell}\ r =\ n * n)]) \\
& \mid X \triangleleft Sqr(X, Y) \\
& \mid \mathit{tell}\ X\ =\ 3)
\end{aligned}
$$

En donde el objeto esta definido por las restricciones ϕ_{sender} $(sender > 0)$, $\delta_{forward}$ $(forward \leq 0)$ y tiene dentro de su colección de métodos al método Sqr y el mensaje

es enviado a través de la variable X.

La restricción ϕ_{sender} podría expresarse en términos de una función R_s para evaluar la posible comunicación así:

$R_s(X) \equiv \phi_{sender}[sender/x]$

Si al reemplazar la variable *sender* por el parámetro de llamada X, satisface la restricción ϕ_{sender} se verificaría la existencia del método Sqr dentro de la colección de métodos del objeto:

- Si existe el método Sqr se establece la comunicación con el objeto.
- Si no existe, la restricción $\delta_{forward}$ podría expresarse en términos de una función R_f de la siguiente forma:
  $R_f(Y) \equiv \delta_{forward}[forward/Y]$

A continuación se presentan las alternativas tomadas en cuenta a lo largo del diseño, en algunas de esas alternativas se plantea un *bytecode* o *código de bytes* basado en instrucciones similares a la de la primera implementación de *MAPiCO* [BH99].

3.2.1.1. Primera Alternativa

En esta alternativa se determinan las restricciones ϕ_{sender} y $\delta_{forward}$ como funciones *sender* y $forward$, con etiquetas especificas que enmarcan cada restricción y en el momento de realizar la definición del objeto se toma en consideración la referencia de dichas etiquetas.
Dadas las siguientes funciones *sender* y $forward$:

$f_s(s) \equiv (s > 0)$
$f_f(f) \equiv (f = 0)$

y el siguiente objeto:

Objcc XX YY: definición de un objeto que tiene asociado dos funciones XX y YY en

donde se realizan los reemplazos correspondientes de las variables *sender* y $forward$. La consideración es la siguiente:

$XX\ def\ f_s(s)$	Definición de XX como función en términos de la variable *sender* XX es el nombre de la función *sender* f_s es la función *sender* s es la variable *sender*
$YY\ def\ f_f(f)$	Definición de YY como función en términos de la variable $forward$ YY es el nombre de la función $forward$ f_f es la función $forward$ f es la variable $forward$

En bytecode sería algo así:

```
DEFS:              ; Etiqueta de la función sender
POP                ; Pasar el PA al PV
ATOM GREAT 2       ; Función >de dos operandos
TERMV 0            ; Variable 0 para evaluar sender
TERMC 0            ; Término constante 0
DEFF:              ; Etiqueta de la función sender
POP                ; Pasar el Pa al PV
ATOM EQ 2          ; Función = de dos operandos
TERMV 0            ; Variable 0 para evaluar forward
TERMC 0            ; Término constante 0
OBJCC DEFS DEFF    ; Definición de objeto OBJCC
                   ; DEFS dirección etiqueta DEFS,
                   ; DEFF direción de etiqueta DEFF
```

Las etiquetas $DEFS$ y $DEFF$ definen las funciones *sender* y $forward$ respectivamente.

Esta alternativa fue descartada porque presenta dificultad en la identificación de funciones *sender* y $forward$ para cada objeto, aunque es una alternativa sencilla en la representación de restricciones no brinda la solución completa en la identificación y comunicación de procesos.

3.2.1.2. Segunda Alternativa

La segunda alternativa contempla la definición de las funciones *sender* y $forward$ como condicionales, en donde la primera guarda es la restricción ϕ_{sender} y la guarda de dele-

gación es la restricción $\delta_{forward}$. De esta manera al realizar los reemplazos de la variable *sender* y satisfacer la restricción ϕ_{sender} pero no encontrar el método respectivo, se procede a ejecutar $\delta_{forward}$. El resultado de la evaluación de las restricciones $\phi_{sender}/\delta_{forward}$ se guarda en un registro de la máquina.

if $s(\phi)$ then P	if $\phi(sender)$ then P
else $f(\delta)$	else $\delta(forward)$

Los procesos quedarían definidos por dos nuevos elementos, un apuntador al registro *sender* y otro a $forward$ para obtener en cada caso el resultado de la evaluación de sus respectivas funciones: $< PC, PA, PV, PN, s, f >$

La restricción ($sender > 0$) quedaría determinada así:

```
>             ; Función >
0             ; Constante 0
SENDER        ; Variable sender
REG SENDER    ; Registro del resultado de evaluación del sender
```

Y la restricción $forward < 0$ quedaría:

```
<             ; Función <
0             ; Constante 0
FORWARD       ; Variable forward
REG FORWARD   ; Registro del resultado de evaluación de forward
```

Un ejemplo del código de bytes sería el siguiente:

```
POP
>
TERMC 0
TERMV 0
REG SENDER
POP
<
TERMC 0
TERMV 0
REG FORWARD
OBJCC S F
```

Esta alternativa se descartó porque presenta dificultad en el mantenimiento de los resultados de evaluación en los registros *sender* y $forward$ y en la determinación de

éstos para cada objeto. El solo hecho de tener estos registros no garantiza realizar una comunicación apropiada y utilizar el concepto de condicional para la comunicación podría no ser clara en cuanto a que hay dos condiciones para la comunicación:

- Que al reemplazar la variable *sender* por los respectivos parámetros, debe satisfacer la restricción ϕ_{sender}.
- Que exista en el proceso receptor el método requerido solicitado por el proceso emisor.

La primera condición prima sobre la segunda, en ese caso la restricción $\delta_{forward}$ no sería una guarda o consecuencia de la condición inicial.

3.2.1.3. Tercera Alternativa

Considera la definición de funciones *sender* y $forward$, tomando como referencia una dirección de retorno. En esta alternativa cada función esta determinada por una dirección de referencia, la cual es necesaria para realizar su respectivo llamado. Así como se realiza un llamado a los métodos, también se realizan llamados a las funciones *sender* y $forward$ y cada una va a tener su correspondiente dirección en una tabla de funciones, de esta forma serán identificadas en el momento del llamado.

def func1 XXX
...
XXX:ret
def func2 YYY
...
YYY:ret

Considerando la siguiente lista de objetos y sus respectivas funciones de evaluación, como se muestra en la figura 3.1.

Obji: Objeto i
si: Función *sender* del Objeto i
fi: Función $forward$ del Objeto i

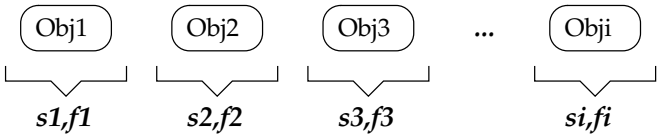

Figura 3.1: Lista de Objetos

Y el siguiente mensaje $a \lhd m(X)$

```
PUSHV X    ; Push parámetros(X)
PUSHM m    ; Push métodos
PUSHV a    ; Push objeto a evaluar
CALLM      ; Llamado al método
CALLF      ; Llamado a función
```

Para el caso de esta alternativa, cuando existe un mensaje queda difícil determinar que objeto puede recibirlo o con cual objeto puede comunicarse, ya que solo después de evaluar la restricción y determinar que existe el método apropiado de comunicación se puede establecer cual es ese objeto, no como en el cálculo anterior que los objetos ya venian determinados por un identificador. Igualmente para esta alternativa, la lógica de la comunicación estaría controlada desde la instrucción $CALLF$ y no habría claridad en el proceso de selección del objeto y mucho menos de la secuencia de ejecución, pues esta mas del lado de la definición de las restriciones ϕ_{sender} y $\delta_{forward}$ que desde el mismo envio del mensaje; por estas razones se descartó esta alternativa.

3.2.1.4. Cuarta Alternativa

En esta alternativa la definición de las funciones *sender* y $forward$ esta contemplada dentro del cuerpo del objeto; en ese caso el objeto se determina por su etiqueta y la dirección del comienzo del primer método. Cuando se trata de establecer comunicación con el objeto se pasa a la línea de evaluación de las restricciones ϕ_{sender} y $\delta_{forward}$, de acuerdo a esto se sigue con la ejecución del cuerpo del método desde la dirección respectiva.

OBJCC METH(DIR)
DEFS:

...
DEFF:
...
METH 1

Esta alternativa proporciona una solución parcial, pero finalmente no hay claridad en la evaluación de las restricciones ϕ_{sender} y $\delta_{forward}$ y en el retorno al método apropiado para la comunicación después de la declaración del objeto, ya que durante la ejecución de la evaluación de las restricciones podría perderse la referencia al método.

3.2.1.5. Quinta Alternativa

Como el bytecode debe soportar la característica de los objetos en el nuevo *PiCO*, que no se referencian por un nombre o una variable, sino por la evaluación de una restricción; para esta alternativa se definió una instrucción que enmarca la definición de las restricciones ϕ_{sender} y $\delta_{forward}$: $FUNC$ (comienzo de la definción de función). El bytecodes de las restricciones se define con respecto a las instruccions de predicados de primer orden.

Para el ejemplo la sección 3.2.1, la codificación del lenguaje máquina estaría definida de la siguiente forma:

```
0          NEWV              ;X
1          NEWV              ;Y
2          PAR ETQ1          ;En paralelo con ETQ1
3 FUNC1:   FUNC RET1         ;Función sender, RET1:dir fin
4          POP               ;Variable sender evaluar del PA
5          ATOM GREAT 2      ;>
6          TERMV 0           ;Variable sender
7          TERMC 0           ;Constante 0
8          ASKF              ;Ask al Store
9          PUSHV 0           ;Variable 0
10         CALL              ;Llamada
11         POP               ;Sacar el método
12         COMM              ;Establecer Comunicación
```

```
13          PUSHM 0                 ;Recuperar el método
14          REE                     ;Reenvío
15 RET1:    FUNC RET2               ;Función forward, RET2:dir fin
16          NEWV                    ;Variable forward a imponer la restricción
17          ATOM EQ 2               ;=
18          TERMV 0                 ;Variable forward
19          TERMC 0                 ;Constante 0
20          TELL                    ;Tell al Store
21          PUSHV 0                 ;Variable 0
22          CALL                    ;Llamada
23 RET2:    COBJCC FUN1 FUN2 1      ;Definición de Objeto clonado, FUN1:R. sender,
                                    ;FUN2:R. forward, 1: Nro de métodos
24          METH 1 25               ;Método Sqr
26          POP                     ;n
27          POP                     ;r
28          ATOM EQ 2               ;=
29          TERMV 0                 ;r
30          TERMF MULT 2            ;*
31          TERMV 1                 ;n
32          TERMV 1                 ;n
33          TELL
34          RET
35 ETQ1:    PAR ETQ2                ;En paralelo con ETQ1
36          PUSHV 1                 ;X
37          PUSHV 0                 ;Y
38          PUSHM 1                 ;Método 1, Sqr
39          PUSHV 1                 ;X
40          CALL                    ;Llamada
41          RET
42 ETQ2:    ATOM EQ 2               ;=
43          TERMV 1                 ;X
44          TERMC 3                 ;Constante 3
45          TELL
```

Cada objeto debe tener inicialmente la definición de las funciones *sender* y $forward$ a través de los predicados de primer orden. Primero se debe definir la función *sender*, determinando la dirección de terminación de la misma (comienzo de la función $forward$), luego la función $forward$; por último la clase del objeto y sus respectivos métodos.

En la anterior codificación, la definición de la restricción ($sender > 0$) esta dada desde el número de línea 3 a la línea 7 y la restricción ($forward \leq 0$) desde la línea 17 a la línea 21. El código intermedio entre estos rangos de líneas definen la comunicación o reenvío de mensajes.

La evaluación de la restricción ϕ_{sender} se realiza a través de la instrucción $ASKF$ (línea 8), dependiendo del resultado de esta evaluación se establece la comunicación o verificación del resto de objetos que puedan atender el llamado. De ser verdadera se procede a establecer la comunicación ($COMM$) y se verifica si el método del llamado se encuentra dentro de la colección de métodos del objeto, de ser así se ejecuta el cuerpo del método, si no, se realiza el reenvío (REE) y se ejecuta la definición de la función $forward$; dando paso a la creación de una nueva variable, a la imposición de la condición de reenvío, y al almacenamiento de esta variable por medio de la instrucción TELL (línea 20).

Una vez se realiza esta evaluación, el comportamiento de la máquina es igual al de la máquina anterior, conservando la mayoría de instrucciones, con algunas variaciones en la implementación.

Esta fue la alternativa escogida porque presenta una solución completa que considera tanto la definición de las restricciones ϕ_{sender} y $\delta_{forward}$, como su evaluación, comunicación y reenvío de procesos.

3.2.2. Características del Diseño Actual

En *MAPiCO*, se definieron las estructuras de soporte, el formato de instrucciones, la codificación y el comportamiento interno, siguiendo la especificación y las reglas de reducción del cálculo *PiCO*.

Un primer elemento a exponer es la arquitectura de la máquina actual, muy similar a la arquitectura de la máquina anterior. Para ello, se describen las áreas constituyentes de la arquitectura que estan representadas en las estructuras usadas en tiempo de ejecución. El diagrama de bloques de la máquina se muestra en la figura 3.2.

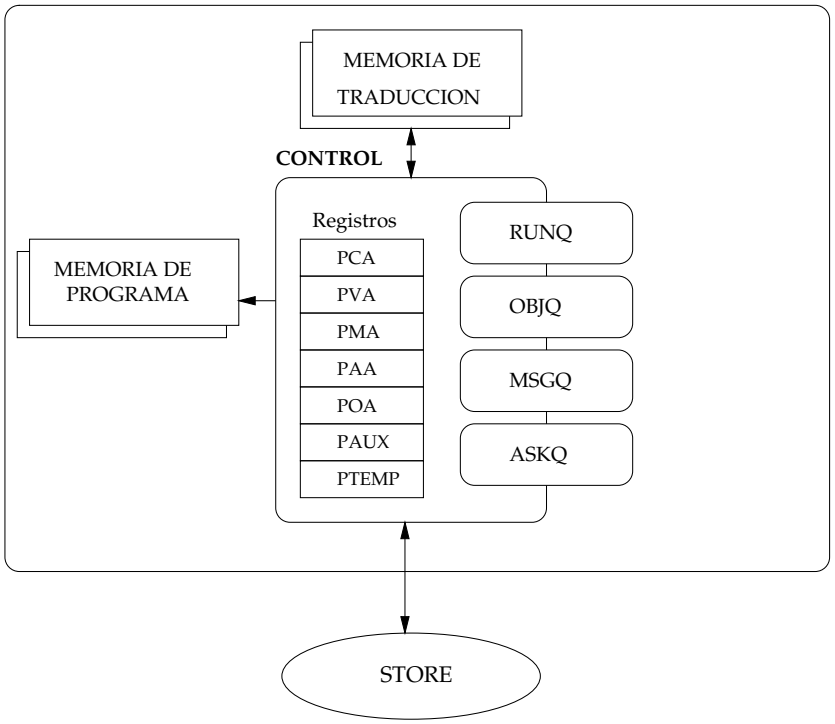

Figura 3.2: Diagrama de Bloques de *MAPiCO*

3.2.2.1. Memoría Estática y Dinámica

La máquina esta compuesta por dos áreas de memoría: Memoría de Programa y Memoría de Traducción.

La memoría de programa hace referencia a la memoría estática que almacena los programas a ser ejecutados.
La memoría de traducción o memoría dinámica, es en donde se almacena las traducciones de ligaduras de variables al *store*, parámetros para el llamado de métodos y es en donde se crea el árbol sintáctico que describe las restricciones.

3.2.2.2. Colas

La máquina cuenta con cuatro colas para los diferentes estados de ejecución de un proceso:

- RunQ: Cola de Ejecución **Rq**.
- ObjQ: Cola de Objetos suspendidos **Oq**.
- MsgQ: Cola de Mensajes suspendidos **Msq**.
- AskQ: Cola de Ask suspendidos **Aq**.

3.2.2.3. Registros Internos

Al igual que en la definición anterior, la máquina tiene un control donde se guardan los registros internos, que son de uso específico para guardar la información de la memoría y de los procesos y a través de éstos la máquina modifica la información para actualizar procesos y colas; estos registros son:

- Puntero a Código Actual **PCA**: Apunta a la instrucción que esta siendo ejecutada.
- Puntero a Variables Actual **PVA**: Apunta al PV del proceso en ejecución. (ver sección 3.2.2.4).
- Puntero a Argumentos de Funciones **PAA**: Apunta al PA del proceso que esta siendo ejecutado.
- Puntero a Métodos Actual **PMA**: Apunta al PM del proceso en ejecución.
- Puntero a Objetos Actual **POA**: Apunta al PO del proceso actual o en ejecución.
- Puntero Auxiliar **PAUX**: Usado para almacenar información temporal de tipo argumentos.
- Puntero Temporal **PTEMP**: Puntero adicional de uso genérico para necesidades futuras.

3.2.2.4. Procesos

Un proceso en *MAPiCO*, tal como se definió en el diseño inicial de la máquina, es él conjunto de información requerida para la ejecución de procesos en *PiCO'*[BH99]. La estructura de procesos en la *MAPiCO* actual cuenta con los siguientes elementos:

- Puntero a Código **PC**: Registro que apunta al próximo código de instrucción a ser ejecutado, en la memoría del programa.
- Puntero a Variables **PV**: Registro que apunta al primer elemento de la pila de variables en la memoría de traducción, al igual que en la definición anterior se modifica con las instrucciones *NEWV* o *POP*, adicionando variables y obteniendo las mismas (Ver sección 3.2.6).
- Puntero a Argumentos de Funciones **PA**: Registro que apunta a una lista con los argumentos y el objeto receptor. Para ser manipulado se utilizan las instrucciones *PUSHV* y *PUSHM* (Ver sección 3.2.6).
- Puntero a Métodos **PM**: Registro que apunta al primer elemento de la pila de métodos, en la memoría de traducción. La pila de métodos puede ser manipulada a través de las instrucciones *PUSHM* para adicionar referencias de métodos y *POP* (para obtener el método actual).Ver sección 3.2.6.
- Puntero a Objetos **PO**: Apunta al pimer elemento de la lista de objetos evaluados en la comunicación actual; Esta lista es manipulada a través de las instrucciones *COMM* y *REE* (Ver sección 3.2.6).
- Puntero Auxiliar **PAUX**: Apunta a estructuras auxiliares para manipulación de procesos.
- Puntero Temporal **PTEMP**: Puntero adicional de uso genérico para necesidades futuras.

En el diseño actual se descartó el siguiente elemento planteado en la máquina anterior: ***Puntero a Nombres PN***, por no considerar necesario referenciar o manipular nombres dentro de la definición actual del cálculo *PiCO*.

3.2.2.5. Interacción con el Sistema de Restricciones

La máquina interactúa constantemente con el sistema de restricciones *store* S. Éste último provee el almacenamiento de restricciones y las operaciones ASK y $TELL$ (Ver sección 3.2.6).

3.2.3. Estados de la Máquina Actual

El diseño de la máquina actual considera la definición y denotación formal de estados de la máquina anterior, con el fin de establecer similitudes y diferencias entre ambas máquinas.

El estado de *MAPiCO* esta representado por:

- **Un Proceso en ejecución** compuesto por:
  $Hilo$: Hilo del proceso.

  $HBind$ ó B: Es el conjunto de variables ligadas que se referencian en el *Store*, estas son de la forma $I_1 \mapsto L_1, ..., I_n \mapsto L_n$, $L \in Storef$ (conjunto de variables que ocurren dentro del *Store*)

  $HAux$ ó H: Es el conjunto de variables y restricciones ligadas en términos de L. $HAux ::= HBind \mid \phi(\widetilde{L})$.

  Cuando no haya ligaduras en $HBind$ y $HAux$ se denotará como: $\oslash$.

- **Colas para los diferentes estados de la máquina**:

  $ObjQ$: Estructura de ligaduras ($HBind$), parámetros ($HAux$) y procesos objeto ($ObjQ := (\phi_{sender}, \delta_{forward}) \triangleright M \mid clone(\phi_{sender}, \delta_{forward}) \triangleright M$).

  $MsgQ$: Estructura de Ligaduras ($HBind$), párametros ($HAux$) y procesos tipo mensaje ($MsgQ := X \lhd Sqr(X, Y) \mid clone(X \lhd Sqr(X, Y))$).

AskQ: Estructura de Ligaduras ($HBind$), restricciones ($HAux$) y procesos ask ($AskQ := ask\ \phi\ then\ P$)

RunQ: Estructura de Ligaduras ($HBind$), restricciones ($HAux$) e hilos de ejecución ($Hilo$).

La denotación de las colas esta dada por:
$Cola(X) ::= X :: ... :: X|\bullet$

En donde:
"::" Representa la concatenación de dos colas o de un elemento y una cola según el caso.
"$\bullet$" Denotación para colas vacias.

- **Store ó S**: Conjunto de conjunciones de restricciones en términos de $L : Store = \phi_1(\widetilde{L}) \wedge ... \wedge \phi_m(\widetilde{L})$. Un *store* vacio se denotará por "$\bullet$".

Según lo anterior, un estado estaría representado por:

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

ó en forma resumida

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

El estado inicial de la máquina es el estado computacional con el cual la máquina comienza su ejecución, en este caso la cola de objetos, mensajes y ask estan vacías. Este estado incial estará representado por:

$$\left\langle \vec{P}, \oslash, \oslash, \bullet, \bullet, \bullet, \bullet, \oslash \right\rangle$$

El estado final de la máquina es el estado en el cual no hay nada mas que planificar en la cola de ejecución y la máquina sabe que los procesos Aq, Mq y Oq estan suspendidos y no pueden ser reducidos. El estado final estará representado por:

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

3.2.4. Diagrama de Transiciones de la Máquina Actual

El diagrama de transiciones de la máquina se presenta en la siguiente figura 3.3:

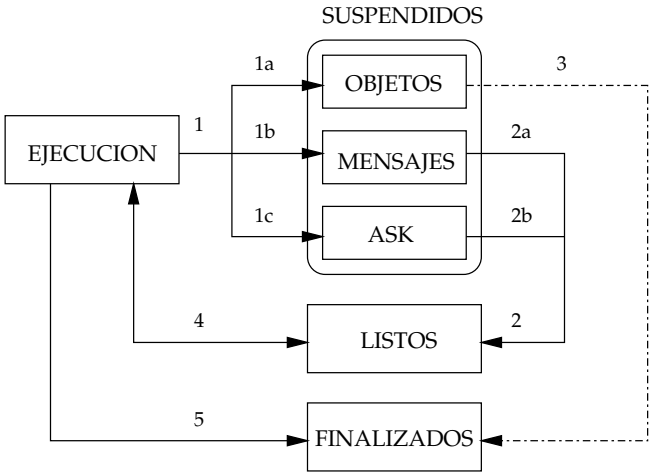

Figura 3.3: Diagrama de Transiciones entre Estados de *MAPiCO*

Donde:

1. Cuando hay procesos emisores (Mensajes), receptores (Objetos) y ask que no cumplen con las reglas de comunicación, reenvío y ask con respecto al *Store*; éstos procesos pasan al estado de suspendidos.

2. Cuando ocurre un *tell*, todos los mensajes y los ask se pasan al estado de listos en la cola de ejecución para resolverse

3. Los procesos objeto pasan a listos cuando hay un mensaje con el que puedan ser comunicados; a diferencia de los otros tipos de proceso, los objetos no pasan a listos en un solo paso.

4. Ejecución de procesos en estado de listos.

5. Finalización de ejecución de procesos.

3.2.5. Reglas de Reducción de la Máquina

Las reglas de reducción se definieron con base en la semántica operacional del Cálculo *PiCO* y en la especificación de la máquina anterior con respecto a la reducción de procesos a través de configuraciones o estados.

Los procesos se reducen transformando una configuración inicial en una final, evaluando y aplicando cada proceso conformado por el hilo de ejecución y el ambiente actual.

El estado de la máquina esta representado por:

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

3.2.5.1. Definición de Reglas de Reducción

Las reglas son de la forma:

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

En donde:

- $Hilo,\ HBind\ y\ HAux$: Representan el ambiente en ejecución.
- B o $HBind$: Es el conjunto de variables ligadas que se referencian en el *Store*.
- H o $HAux$: Es el conjunto de variables y restricciones ligadas.
- S: Es el *Store*, conjunto de conjunciones de restricciones.
- $ObjQ$ (Oq): Estructura de Ligaduras, parámetros y procesos objeto.
- $MsgQ$ (Msq): Estructura de Ligaduras, párametros y procesos tipo mensaje.
- $AskQ$ (Aq): Estructura de Ligaduras, restricciones y procesos ask.
- $RunQ$ (Rq): Estructura de Ligaduras, párametros, restricciones e hilos de ejecución.

Las reglas son las siguientes:

El proceso $SCHED$ remueve el poceso actual en ejecución, permitiendo que el siguiente proceso en la cola de ejecución sea considerado para reducir:

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

El proceso *local x in P* o *Local a in P* crea un nuevo enlace de x a L donde L es realmente una variable o nombre.

$$NEW - REF{:}\frac{L\ Nueva}{\langle Local\ x\ in\ P,B,H,Oq,Msq,Aq,Rq,S\rangle \longrightarrow \langle P,B+(x\rightarrow L),H,Oq,Msq,Aq,Rq,S\rangle}$$

En el caso de una composición paralela $(P|Q)$, se deja el proceso Q al final de la cola de ejecución siguiendo con la ejecución del proceso P:

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

Para reducir un proceso *tell* ϕ *then* $\vec{P}$, la restricción $\phi(\vec{L})$ 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 en ejecución:

$$TELL{:}\frac{\forall L_i \in \widetilde{L}\ \ L_i \in Dom(B)}{\left\langle tell\ \phi\ in\ \vec{P},B,\phi(\widetilde{L}),Oq,Msq,Aq,Rq,S\right\rangle \longrightarrow \left\langle \vec{P},B,\oslash,Oq,\bullet,\bullet,Rq::Msq::Aq,S\wedge\phi(\widetilde{L})\right\rangle}$$

En donde $Dom(B)$ es el dominio del conjunto de ligaduras de variables a referenciar dentro del *store*.

Para reducir un proceso *ask* ϕ *then* $\vec{P}$, se necesita del Sistema de Restricciones, hay tres posibles casos:
1. Si del *Store* se deduce $\phi(\widetilde{L})$ entonces la máquina continua con la ejecución del hilo $\vec{P}$

$$ASK_1{:}\frac{S\models\Delta\phi(\widetilde{L}) \qquad \forall L_i \in \widetilde{L}\ \ L_i \in Dom(B)}{\left\langle ask\ \phi\ then\ \vec{P},B,\phi(\widetilde{L}),Oq,Msq,Aq,Rq,S\right\rangle \longrightarrow \left\langle \vec{P},B,\oslash,Oq,Msq,Aq,Rq,S\right\rangle}$$

2. Si del *Store* se deduce $\neg\phi(\widetilde{L})$ entonces la máquina elimina la continuación del proceso actual

$$ASK_2: \frac{S\models\Delta\neg\phi(\widetilde{L}) \qquad \forall L_i\in\widetilde{L} \quad L_i\in Dom(B)}{\langle ask\ \phi\ then\ \vec{P},B,\phi(\widetilde{L}),Oq,Msq,Aq,Rq,S\rangle \longrightarrow \langle nil,B,\phi(\widetilde{L}),Oq,Msq,Aq,Rq,S\rangle}$$

3. Si del *Store* no se puede deducir $\phi(\widetilde{L})$ ó $\neg\phi(\widetilde{L})$, entonces el proceso es suspendido pasándolo a la cola de Aq:

$$ASK_3: \frac{S\not\models\Delta\phi(\widetilde{L}) \qquad S\not\models\Delta\neg\phi(\widetilde{L}) \qquad \forall L_i\in\widetilde{L} \quad L_i\in Dom(B)}{\langle ask\ \phi\ then\ \vec{P},B,\phi(\widetilde{L}),Oq,Msq,Aq,Rq,S\rangle \longrightarrow \langle nil,B,\phi(\widetilde{L}),Oq,Msq,Aq::(B,\phi(\widetilde{L}),ask\ \phi\ then\ \vec{P}),Rq,S\rangle}$$

Si se intenta reducir una comunicación de un objeto que no se encuentra en la cola de Objetos Oq, este mensaje se suspende en la cola de Mensajes Msq. Objetos y mensajes replicados son llevados a las colas Oq y Msq respectivamente.

$$MsgEnq: \frac{S\models\Delta I' \qquad S\not\models\Delta\phi[I'/sender] \qquad (\phi_{sender},\delta_{forward})\rhd M\notin Oq \qquad (\widetilde{K}\to\widetilde{L})\in B}{\langle I'\lhd l_i:[\widetilde{K}]\ then\ \vec{P},B,\widetilde{x_i}\to\widetilde{L},Oq,Msq,Aq,Rq,S\rangle \longrightarrow \langle nil,B,\widetilde{x_i}\to\widetilde{L},Oq,Msq::(B,\widetilde{x_i}\to\widetilde{L},I'\lhd l_i:[\widetilde{K}]\ then\ \vec{P}),Aq,Rq,S\rangle}$$

Si el proceso es un mensaje $I'\lhd l_i:[\widetilde{K}]\ then\ \vec{P}$ 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 Oq 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 $\vec{P}$

$$MsgComm$$
$$\frac{S\models\Delta I' \qquad S\models\Delta\phi[I'/sender] \qquad |\widetilde{K}|=|\widetilde{x}|}{\left\langle \begin{array}{l} I'\lhd l_i:[\widetilde{K}]\ then\ \vec{Q},B,\widetilde{x_i}\to\widetilde{L},\\ Oq::(B',\emptyset,(\phi_{sender},\delta_{forward})\rhd[l_1:(\widetilde{x_1})\vec{P_1},...,l_m:(\widetilde{x_m})\vec{P_m}]),\\ Msq,Aq,Rq,S \end{array} \right\rangle \longrightarrow \left\langle \begin{array}{l} \vec{Q},B,\emptyset,Oq,Msq,Aq,\\ Rq::(B'+(\widetilde{x_i}\to\widetilde{L}),\emptyset,\vec{P_i}\{\widetilde{K_i}/\widetilde{x_i},I'/sender\}),S \end{array} \right\rangle}$$

Si el objeto $(\phi_{sender},\delta_{forward})\rhd[l_1:(\widetilde{x_1})\vec{P_1},...,l_m:(\widetilde{x_m})\vec{P_m})]$ existe en la colaOq pero no hay una etiqueta l_i para comunicarse con $I'\lhd l_i:[\widetilde{K}]\ then\ \vec{Q}$ y el objeto tiene una restricción de reenvío $\delta_{forward}$, se crea una nueva variable J y se le impone la

restricción $\delta_{forward}$ $(local\ J\ in\ tell\ \delta[J/forward])$ y se deja en la cola Oq el proceso $(J \lhd l_i : [\widetilde{K}]\ then\ Q) \mid (\phi_{sender}, \delta_{forward}) \rhd M)$

Inicialmente se quita el proceso actual y se envia a ejecución el proceso:
$(local\ J\ in\ (tell\ \delta[J/forward])\ then\ (J \lhd l_i : [\widetilde{K}]\ then\ Q))$:

$$MsgRee: \quad \frac{S \models \Delta\phi[I'/Sender] \qquad S \sqcup \delta[I'/forward] \models \Delta\perp \qquad l \notin Labels(M)}{\left\langle \begin{array}{l} I' \lhd l_i : [\widetilde{K}]\ then\ \vec{Q}, B, \widetilde{x_i} \to \widetilde{L}, \\ Oq :: (B', \emptyset, (\phi_{sender}, \delta_{forward}) \rhd [l_1 : (\widetilde{x_1})\vec{P_1}, ..., l_m : (\widetilde{x_m})\vec{P_m}]), \\ Msq, Aq, Rq, S \end{array} \right\rangle \longrightarrow \left\langle \begin{array}{l} nil, B, \widetilde{x_i} \to \widetilde{L}, \\ Oq :: (B', \emptyset, (\phi_{sender}, \delta_{forward}) \rhd [l_1 : (\widetilde{x_1})\vec{P_1}, ..., l_m : (\widetilde{x_m})\vec{P_m}]), \\ Msq, Aq, Rq :: (B, (\widetilde{x_i} \to \widetilde{L}), (local\ J\ in\ (tell\ \delta[J/forward]), \\ then\ (J \lhd l_i : [\widetilde{K}]\ then\ Q))), S \end{array} \right\rangle}$$

Posteriormente la ejecución de este tipo de Proceso quedaría de la siguiente forma:

$$MsgRee: \quad \frac{S \models \Delta\phi[I'/Sender] \qquad S \sqcup \delta[I'/forward] \models \Delta\perp \qquad l \notin Labels(M)}{\left\langle \begin{array}{l} I' \lhd l_i : [\widetilde{K}]\ then\ \vec{Q}, B, \widetilde{x_i} \to \widetilde{L}, \\ Oq :: (B', \emptyset, (\phi_{sender}, \delta_{forward}) \rhd [l_1 : (\widetilde{x_1})\vec{P_1}, ..., l_m : (\widetilde{x_m})\vec{P_m}]), \\ Msq, Aq, Rq, S \end{array} \right\rangle \longrightarrow \left\langle \begin{array}{l} nil, B, (\widetilde{x_i} \to \widetilde{L}), \\ Oq :: (B', \emptyset, (\phi_{sender}, \delta_{forward}) \rhd [l_1 : (\widetilde{x_1})\vec{P_1}, ..., l_m : (\widetilde{x_m})\vec{P_m}]), \\ \bullet, \bullet, Rq :: Msq :: Aq :: (B, (\widetilde{x_i} \to \widetilde{L}) + (J \to \widetilde{L}), (J \lhd l_i : [\widetilde{K}]\ then\ Q)), \\ S \wedge \delta[J/forward] \end{array} \right\rangle}$$

En la siguiente regla no hay ningún mensaje en la cola Msq para comunicarse con $(\phi_{sender}, \delta_{forward}) \rhd [l_1 : (\widetilde{x_1})\vec{P_1}, ..., l_m : (\widetilde{x_m})\vec{P_m})]$, por tanto el objeto es pasado al final de la cola de objetos Oq para su posterior reducción:

$$ObjEnq: \quad \frac{S \models \Delta I' \qquad S \not\models \Delta\phi[I'/sender] \qquad I' \lhd m\ then\ \vec{P} \notin Msq}{\left\langle \begin{array}{l} (\phi_{sender}, \delta_{forward}) \rhd [l_1 : (\widetilde{x_1})\vec{P_1}, ..., l_m : (\widetilde{x_m})\vec{P_m}], \\ B, \emptyset, Oq, Msq, Aq, Rq, S \end{array} \right\rangle \longrightarrow \left\langle \begin{array}{l} nil, B, \emptyset, Oq :: (B, \emptyset, (\phi_{sender}, \delta_{forward}) \rhd [l_1 : (\widetilde{x_1})\vec{P_1}, ..., l_m : (\widetilde{x_m})\vec{P_m}]), \\ Msq, Aq, Rq, S \end{array} \right\rangle}$$

Cuando existe un mensaje $I' \lhd l_i : [\widetilde{K}]\ then\ \vec{P}$ en la cola de mensajes Msq para comunicarse con el objeto $(\phi_{sender}, \delta_{forward}) \rhd [l_1 : (\widetilde{x_1})\vec{P_1}, ..., l_m : (\widetilde{x_m})\vec{P_m})]$ y adicionalmente hay una etiqueta l_i en la lista de métodos del objeto; el mensaje es eliminado de la cola Msq (si no es replicado),la continuación del mensaje $\vec{P}$ es colocado al final de la cola de ejecución Rq, seguido por el cuerpo del método $\vec{P_i}$ con el nuevo enlace para $\widetilde{k}$

$$ObjComm:\quad \frac{S\models\Delta I' \qquad S\models\Delta\phi[I'/sender] \qquad |\widetilde{K}|=|\widetilde{x}|}{\left\langle \begin{array}{l}(\phi_{sender},\delta_{forward})\triangleright[l_1{:}(\widetilde{x_1})\vec{P}_1,...,l_m{:}(\widetilde{x_m})\vec{P}_m],B,\emptyset,Oq,\\ Msq{::}(B',(\widetilde{x_i}\to\widetilde{L}),I'\triangleleft l_i{:}[\widetilde{K}]\ then\ \vec{P}),Aq,Rq,S\end{array}\right\rangle \longrightarrow \left\langle \begin{array}{l}nil,B,\emptyset,Oq,Msq,Aq,\\ Rq{::}(B+(\widetilde{x_i}\to\widetilde{L}),\emptyset,\vec{P}_i\{\widetilde{K_i}/\widetilde{x_i},I'/sender\}),S\end{array}\right\rangle}$$

Si existe un mensaje $I' \triangleleft l_i : [\widetilde{K}]\ then\ \vec{P}$ en la cola Msq para intentar comunicarse con el objeto $(\phi_{sender}, \delta_{forward}) \triangleright [l_1 : (\widetilde{x_1})\vec{P}_1, ..., l_m : (\widetilde{x_m})\vec{P}_m)]$ con una restricción de reenvío $\delta_{forward}$, pero no hay una etiqueta l_i en la lista de métodos del objeto; el mensaje es pasado de la cola de mensajes Msq al final de la cola de ejecución Rq, pero creando una nueva variable J, imponiendole la restricción $\delta_{forward}$, cambiando el proceso por $(J \triangleleft l_i : [\widetilde{K}]\ then\ Q) \mid (\phi_{sender}, \delta_{forward}) \triangleright M)$ y pasando el objeto a la cola Oq:

$$ObjRee:\quad \frac{S\models\Delta\phi[I'/_{Sender}] \qquad S\sqcup\delta[I'/forward]\models\Delta\bot \qquad l\notin Labels(M)}{\left\langle \begin{array}{l}(\phi_{sender},\delta_{forward})\triangleright[l_1{:}(\widetilde{x_1})\vec{P}_1,...,l_m{:}(\widetilde{x_m})\vec{P}_m],B,\emptyset,Oq,\\ Msq{::}(B',\emptyset,(\widetilde{x_i}\to\widetilde{L}),I'\triangleleft l_i{:}[\widetilde{K}]\ then\ \vec{P}),Aq,Rq,S\end{array}\right\rangle \longrightarrow \left\langle \begin{array}{l}nil,B,\emptyset,\\ Oq{::}(B,\emptyset,(\phi_{sender},\delta_{forward})\triangleright[l_1{:}(\widetilde{x_1})\vec{P}_1,...,l_m{:}(\widetilde{x_m})\vec{P}_m]),Msq,\\ Aq,Rq{::}(B',(\widetilde{x_i}\to\widetilde{L})+(J\to\widetilde{L}),J\triangleleft l_i{:}[\widetilde{K}]\ then\ \vec{Q}),S\wedge\delta[J/forward]\end{array}\right\rangle}$$

3.2.6. Conjunto de Instrucciones de MAPiCO

Con respecto al conjunto de instrucciones de la nueva versión de *MAPiCO*, se mantienen la BNF para la especificación de predicados de primer orden (Tabla 3.3) y las instrucciones para construir estos predicados (Tabla 3.4); pero se contemplan cambios en las instrucciones para la manipulación de procesos y para la definición de objetos de la máquina (ver Tabla 3.5 y Tabla 3.6).

Se puede observar en la Tabla 3.5 que se adicionaron instrucciones como $PUSHM$, $ASKF$ y $FINISH$. La instrucción $PUSHM$ se adicionó para pasar la referencia del método, $ASKF$ para evaluar las restricciones ϕ_{sender} y $\delta_{forward}$ a través del sistema de restricciones y $FINISH$ para Finalizar o realizar un brake en la ejecución de la Máquina Abstracta.

En el caso de las instrucciones para la definición de objetos de la máquina (Tabla 3.6), se cuenta con las instrucciones $OBJCC$ y $COBJCC$ para la definición de objetos por restricciones y no por las combinaciones de nombres y variables como en el diseño de la máquina anterior (Tabla 3.2); $COMM$ y REE para establecer la comunicación y el

Instrucción	**Opcode**	**Src1**	**Descripción**
INIT	1		inicializa procesos
RET	12		Es el proceso NIL en el cálculo
PAR	14	dir	Crea un nuevo porceso apuntado por Src1 para ser ejecutado en paralelo
NEWV	13		Definición de una nueva variable.
PUSHV	8	ind	Pone variables en el PA para ser pasadas como argumentos en comunicación de objetos.
PUSHM	9	ind	Pone métodos en el PA para ser pasados como argumentos en comunicación de objetos.
POP	7		Saca el primer elemento de la pila apuntada por PA para que pase a ser una variable o nombre según el caso.
ASK	24	num	Realiza la operación ask al *Store* de una restricción apuntada por PA
TELL	23		Adiciona al *Store* la restricción apuntada por PA.
CALL	17		Llamado de funciones y parámetros
ASKF	25		Realiza la operación de *Ask* para restricciones ϕ_{sender} y $\delta_{forward}$.
FINISH	15		Finaliza o realiza un brake en la ejecución de la Máquina Abstracta.

Tabla 3.5: Instrucciones para la Manipulación de Procesos en la Máquina - Versión Actual

Instrucción	**Opcode**	**Src1**	**Src2**	**Src3**	**Descripción**
OBJCC	10	ind	ind	num	Objeto definido por medio de restricciones ϕ_{sender} y $\delta_{forward}$, estas funciones estan determinados por indices de función Src1 y Src2 y número de métodos igual a num
COBJCC	11	ind	ind	num	Objeto objcc pero clonado
METH	6	num	dir		Especifica que el método Scr1 esta en la dirección Src2
COMM	19				Establece la comunicación entre los objetos
REE	18				Establece el reenvío entre objetos.
FUNC	16	dir			Definición de Funciones *sender* y $forward$, sr1 es la dirección fin de la función

Tabla 3.6: Instrucciones para la Definición de Objetos de la Máquina - Versión Actual

reenvío entre objetos y $FUNC$ para enmarcar la definición de las funciones *sender* y $forward$.

3.3. Análisis de Cambios entre la Versión Anterior y la Versión Actual de MAPiCO

A continuación se plantearan algunas diferencias entre la Versión Anterior de *MAPiCO* y la Versión Actual de acuerdo a las modificaciones realizadas al cálculo *PiCO*, al diseño y a la implementación de la Máquina Abstracta. Los detalles se presentan en la Tabla 3.7:

Diferencia en	*MÁQUINA ABSTRACTA MAPiCO*	
	MAPiCO Anterior	**MAPiCO Actual**
Especificación formal	$(I, J) \triangleright M$: Las reglas de reducción se realizaron con base en el establecimiento de comunicación a través de identificadores I, J	$(\phi_{sender}, \delta_{forward}) \triangleright M$:Las reglas de reducción se realizaron con base en el establecimiento de comunicación a través de restricciones ϕ_{sender} y $\delta_{forward}$.
Diseño	Un proceso estaba representado por: $< PC, PV, PN, PA >$. Existía el apuntador a nombres para identificar los objetos a los cuales se les enviaba mensajes	Un proceso esta representado por $< PC, PV, PA, PO, PTEMP >$. No existe el apuntador a nombres, a pesar de que en la definición del Cálculo se declara la utilización de nombres; no se expresa ni en la semántica, ni en las reglas de reducción de la máquina.
Implementación	Conjunto de Instrucciones no extensibles. Para adicionar instrucciones se debe volver a compilar toda la máquina.	Conjunto de instrucciones extensibles, gracias a la implementación de módulos dinámicos. No se debe volver a compilar toda la máquina (ver sección 4.6).
	Conjunto de Instrucciones simples que representan la comunicación de objetos y reenvío de mensajes: $CALL$,$ASKF$,$REPLICATE$.	Conjunto de Instrucciones mas elaboradas para la comunicación de objetos y reenvío de mensajes: $FUNC$, $COMM$ y REE. Estas instrucciones representan la definición y evaluación de restricciones, comunicación de procesos y reenvío de mensajes.

Tabla 3.7: Diferencias Implementación y Reimplementación de *MAPICO*

3.4. Sistema de Restricciones y Eliminación de Variables

Durante el proceso de ejecución de la Máquina Abstracta *MAPiCO*, ésta interactúa constantemente con el *Store*, ya sea creando variables, adicionando información (imponiendo restricciones mediante la instrucción $TELL$) y evaluando la información contenida en éste (mediante la instrucción ASK). Algunos procesos finalizan en determinado punto de la ejecución dejando variables que no pueden ser alcanzadas al no existir una referencia a ellas, por tal razón éstas podrían ser eliminardas del *store* luego de propagar sus valores (*arco-consistencia parcial* [1]) y continuar la ejecución asegurando mayor rendimiento en el momento de evaluar la consistencia del *store* ante instrucciones como $TELL$ y ASK.

En el presente trabajo presentaremos una propuesta que realiza el reconocimiento de estas variables. La propuesta consiste en que durante la ejecución, *MAPiCO* sea la encargada de suministrar la información de las variables que podrían eliminarse al *Store* y este último se encargue de la propagación de sus valores y de su eliminación dentro del Sistema de Restricciones.

Para llevar a cabo el reconocimiento de las variables locales que pueden eliminarse se debe determinar la finalización de los procesos que las crearon, esta labor puede cumplirse evaluando cada instrucción PAR que define los ambientes a los cuales están asociadas las diferentes variables (ver figura 3.4).

Según lo anterior, la finalización de un proceso se determina a través de los diferentes ambientes de las variables enmarcados por la instrucción PAR; analizando este árbol, cada vez que no exista un PAR interno en el transcurso de la ejecución se puede decir que ese proceso finalizó y que las variables locales asociadas a éste pueden eliminarse.

En *MAPiCO* las variables tienen asociado un identificador de proceso de creación, el cuál como su nombre lo indica contiene la información del proceso durante el cual se creo esa variable. En ese caso, si tenemos un proceso X que finalizó en el transcurso de la ejecución, se realiza un recorrido de la pila de variables para determinar cuales

[1] Un algoritmo parcialmente arco-consistente solamente elimina un subconjunto de todos los valores que satisfacen la restricción para los cuales esta no es arco-consistente.

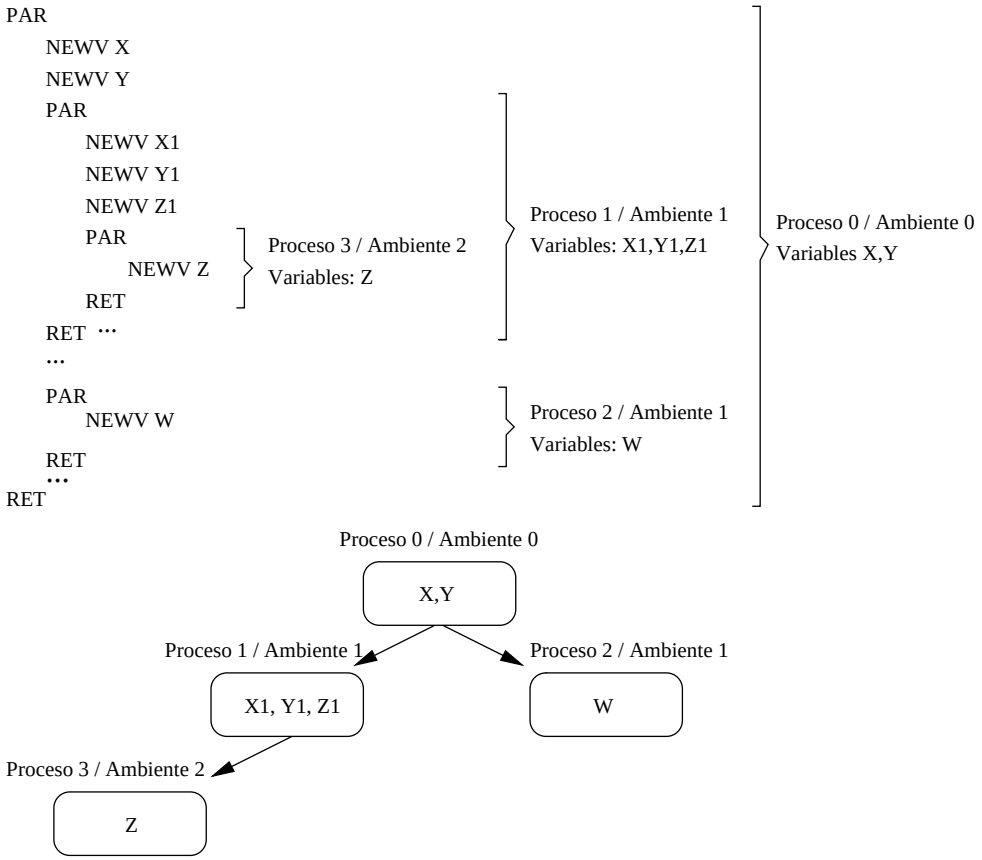

Figura 3.4: Eliminación de Variables

variables tienen asociado como identificador de proceso a X; y éstas serán reportadas al *Store* como candidatas de eliminación. Tomando en consideración lo anterior, es posible utilizar la identificación de variables que no se usan más en la realización de un garbage collector dentro de *MAPiCO*.

Una vez que *MAPiCO* cuente con la información de las variables a eliminar debe suministrarla al *store* y éste propagar sus valores a través de un algoritmo de *arco-consistencia parcial* y posteriormente eliminar estas variables. La definición formal describe al *Store* como monotonicamente creciente en información, pero la eliminación de variables no esta en contra de esta definición siempre y cuando antes se haya propagado la información (*arco-consistencia*).

4 IMPLEMENTACION DE LA MAQUINA ABSTRACTA

Para implementar la máquina abstracta *MAPiCO*, se utilizó el lenguaje de programación *C*, se definieron varias estructuras de datos para manejar el área de memoria, el manejador de procesos, el paso de parámetros a las instrucciones y las estructuras requeridas para guardar información necesaria durante la ejecución de la máquina, todo esto siguiendo las estructuras dadas en el diseño. Adicionalmente se usó el precompilador de C para hacer de ésta una implementación mas flexible utilizando macros y directivas de precompilación.

4.1. Estructuras de Datos

En *MAPiCO* se definieron áreas de memoria para guardar referencia de los procesos suspendidos (colas de objetos, mensajes y ask), procesos en ejecución (cola de ejecución), mantener el Sistema de Restricciones y la memoria del programa. Para implementar las colas, se definió la librería *List.h* la cual fue implementada utilizando macros tipo función de *C*, que aumentan la velocidad de ejecucion del código porque no se pierde tiempo en llamados de funciones y permite manejar el TAD parecido a los template de $C++$.

Esta lista tiene una referencia al primer y último nodo de la lista, y el número de elementos en la lista.

```
#ifndef LIST_H
#define LIST_H
```

```
#define DEFINE_LIST( listTypeName , NodeList )
\
struct listTypeName
\
{
\
  int Elements;                    // TamaNo de la lista
\
  struct NodeList *head, *tail;    // Referencia el inicio y final de la lista
\
};
...
#endif // LIST_H
```

La librería List provee las siguientes macros:

```
#ifndef LIST_H
#define LIST_H

// Define una lista con cabecera y cola
#define DEFINE_LIST( listTypeName , nodeTypeName )                \

// Crea un nueva lista
#define LIST(NewList)                                             \

// Retorna la longitud de la lista. Complejidad: O(1)
#define LENGTH_LIST(List) ( (List)->Elements )

// Inserta un dato al inicio de la lista. Complejidad: O(1)
#define INSERT_AT_HEAD( List , Object )                           \

// Inserta un nuevo dato al final de la lista. Complejidad: O(1)
#define INSERT_AT_END( List , Object )                            \

// Devuelve la referencia del inicio de la lista. Complejidad: O(1)
#define FIRST_NODE(List) ( (List)->head )

// Devuelve la referencia del final de la lista. Complejidad: O(1)
#define END_NODE(List) ( (List)->tail )

// Agrega una lista al final de otra lista. Complejidad: O(1)
#define ADD_AT_END(List,ListAppend)                               \

// Inserta un nuevo dato en la lista en un indice especifico. Complejidad: O(N)
#define INSERT_AT(List,index,Object)                              \

// Elimina un dato de la lista de un indice especifico. Complejidad: O(N)
#define DEL_AT(List,index)                                        \

// Devuelve un dato de la lista de un indice especifico. Complejidad: O(N)
```

```
#define GET_AT(List,index,Object)                                    \

// Asigna un dato a la lista en un indice especifico. Complejidad: O(N)
#define SET_AT(List,index,Object)                                    \

// Libera una Lista. Complejidad: O(N)
#define FREE_MEMORY_LIST(List)                                       \

// Imprime una Lista. Complejidad: O(N)
#define PRINT_LIST(List)                                             \

#endif // LIST_H
```

La librería *NodeList.h* define un nodo de la lista. Estos nodos pueden almacenar cualquier tipo de dato porque tienen una referencia a un elemento de tipo void * (puntero genérico), el cual permite recibir cualquier tipo de argumento puntero sin que se produzca un error de discordancia de tipo. La librería *NodeList.h* tiene una referencia al siguiente nodo, al anterior y al elemento de la lista.

```
#ifndef NODELIST_H
#define NODELIST_H

//Define los apuntadores siguiente y atras de una lista
#define DEFINE_LINK( listTypeName, nodeTypeName )                    \
   struct nodeTypeName * prev##listTypeName, * next##listTypeName    \
   ...
#endif // NODELIST_H

struct NodeList
{
   void *Data;                      // Dato del nodo
   DEFINE_LINK(NodeList,NodeList); // Siguiente nodo y anterior nodo
};
```

Las macros dadas por la librería son:

```
#ifndef NODELIST_H
#define NODELIST_H

// Crea un nuevo nodo con un dato especifico
#define NODELIST(Node,Object)

// Crea un nuevo nodo con un dato especifico y un siguiente nodo
#define NODELIST_CONTINUATION(Node,Object,Continuation)
```

```
// Asigna un siguiente nodo
#define SETNEXT(Node, Continuation)

// Asigna un dato al nodo
#define SETDATA(Node, Object)

// Devuelve el siguiente nodo
#define GETNEXT(Node)

// Devuelve el dato del nodo
#define GETDATA(Node)

// Devuelve un dato de un indice especifico
#define GETDATAAT(Node, index, Respuesta)

// Devuelve la longitud de la lista
#define LENGTH(Node, Respuesta)

// Libera la lista
#define FREE_MEMORY_NODELIST(Node)

// Imprime la lista
#define PRINT_NODELIST(Node)

#endif // NODELIST_H
```

El almacenamiento del programa esta en una estructura de datos, la cual esta definida por un arreglo de caracteres que guarda el *bytecode* cargado desde un archivo binario. Éste archivo contiene el código de las instrucciones de máquina que especifican un programa. La librería *Loader* es usada para cargar este archivo en la memoría del programa, la cual esta definida así:

```
// Estructura que almacena el archivo en bytes y su tamaNo
struct strProgram
{
   int Size;    // Longitud programa en memoria
   char *File; // Programa en memoria
   ...
};

typedef struct strProgram *Program; // Tipo para definir el programa en bytes
```

Para manejar la memoria de programa (en la libreria Loader) estan las funciones para cargar, obtener bytes, words o double words de la memoria.

```
// Cargar un programa en bytes en un arreglo en memoria
Program Loader(char *FileName);

// Retorna un byte
char get8(Program Data, int Pointer);

// Retorna un word
short get16(Program Data, int Pointer);

// Retorna un double word
int get32(Program Data, int Pointer);
```

Para el manejo del Sistema de Restricciones se definió la librería *StoreMA.h*, la cual realiza la interfaz con el Sistema de Restricciones de Dominios Finitos hecho en C [GV01].

La estructura esta definida de la siguiente manera:

```
struct str_StoreMA
{
   Store s;          // Store de Restricciones
   Indexicals i;     // Indexicals
   Stamp *t;         // Stamp
   int *indextemp;   // IndexTemp
   int Response;     // Codigo de respuesta de la accion
   char *Error;      // String con el mensaje de error
   int inicializado; // Indica si el store ha sido inicializado
};
typedef struct str_StoreMA *StoreMA;

//Se define el tipo FrameConstMA para la interfaz con el Store
typedef FrameConst FrameConstMA;
```

Para interactuar con el *Store* estan las siguientes funciones:

```
// Inicializa el Store
StoreMA inicStoreMA(void);

// Realiza un Tell al Store
StoreMA tellMA(StoreMA sm, FrameConstMA fc);

// Realiza un Ask al Store
StoreMA askMA(StoreMA sm, FrameConstMA fc);

// Inicializa una RestricciOn
```

```
FrameConstMA InicFrameConstMA(int, int);

// Inicializa una RestricciOn a partir de otra
FrameConstMA InicpFrameConstMA(FrameConstMA fc);

// Devuelve una restriciOn hija por la izquierda
FrameConstMA getLefSonFrameConstMA(FrameConstMA);

// Devuelve una restriciOn hija por la derecha
FrameConstMA getRigSonFrameConstMA(FrameConstMA);

// Asigna una restriciOn hija por la izquierda
void setLefSonFrameConstMA(FrameConstMA, FrameConstMA);

// Asigna una restriciOn hija por la derecha
void setRigSonFrameConstMA(FrameConstMA, FrameConstMA);

// Libera la una restricciOn
void DestruirFrameConstMA(FrameConstMA);

// Imprime una restricciOn
void ImprimirFrameConstMA(FrameConstMA);

// Indica si el Store se encuentra Inicializado
int getInicializadoMA(StoreMA sm);
```

La librería *NodeArg.h* implementa un árbol que permite traducir la ligadura de variables y de métodos, la estructura es similar a la estructura de ligaduras de la implementación anterior [BH99], pero tiene un campo adicional, su tipo: (variable o método).

```
struct NodeList
{
   void *Data; // Dato del nodo
   DEFINE_LINK(NodeList,NodeList);   // Siguiente nodo y anterior nodo
};

struct NodeArg
{
   void *Data;                          // Dato del nodo
   int Type;                            // tipo de nodo (Variable o Metodo)
   DEFINE_LINK_ARG(NodeArg, NodeArg);   // Siguiente nodo y anterior nodo
};
```

Adicionalmente para pasar parámetros en el envio de mensajes se implementó la librería *Arguments.h* que permite almacenar las variables o métodos por medio de la librería *NodeArg.h* y las restricciones por medio de la librería *StoreMA.h*

```
struct Arguments
{
   NodeArg nArg;          // Variables o Metodos
   FrameConstMA Fconst;   // RestriccIOn
   short Type;            // Type = { -1[Not Defined], 0[NodeArg], 1[FrameConst] }
};
```

La estructura para registrar los objetos evaluados para un mensaje, se implementó en la librería *ObjectEval.h* la cual permite identificar en memoria los objetos a los cuales les ha llegado el mensaje pero no lograron satisfacer la fución *sender*.

```
struct ObjEval
{
   int *PCS;         // Arreglo de direcciones
   int NumObjects;   // Numero de Objetos que contiene el arreglo
   int Longitud;     // Longitud inicial del arreglo
};

//Tipo de ObjectEval
typedef struct ObjEval *ObjectEval;
```

La anterior estructura implementa un arreglo con las direcciones de los objetos en el *bytecode*, que no lograron satisfacer la función *sender*, este es un arreglo que se encuentra siempre ordenado por medio del algoritmo de ordenación *HeapSort* (implementado en la librería *HeapSort.h*) para una rápida validación de objetos.

4.2. Estructura de un Proceso

El proceso esta representando por una septupla donde esta la información necesaria para ejecutarlo. Estas variables son el puntero a código, puntero a variables, puntero a métodos, puntero para paso de parámetros y de definición de restricciones, puntero a objetos, puntero auxiliar y un puntero temporal reservado para uso futuro. A excepción del puntero a código **PC**, los apuntadores fueron definidos como void * para guardar cualquier tipo de información.

Todas estas variables son atributos de la librería *Process.h*, en donde hay funciones para obtener cada uno de los atributos de dicha librería.

```
struct strProcess
```

```
{
   int PC;        // Contador de Programa
   void *PV;      // Apuntador a Variables
   void *PM;      // Apuntador a Metodos
   void *PA;      // Apuntador a Argumentos
   void *PO;      // Apuntador a Objetos
   void *PAUX;    // Apuntador auxiliar
   void *PTEMP;   // Apuntador temporal
};

// Tipo para definir un proceso
typedef struct strProcess *Process;
```

4.3. Estructura de los PlugIn's

Para cargar e identificar los *PlugIn's* se definió una estructura que apunta al manejador y a la ruta donde se encuentra la librería dinámica, la estructura es la siguiente:

```
struct str_PlugIn
{
   void *Handle;        // Apuntador al Manejador del PlugIn
   char *PathLibrary;   // Path donde se encuentra la libreria dinamica
};
typedef struct str_PlugIn  PlugIn;
```

Cada PlugIn puede contener una o más instrucciones que ejecuta la Máquina Abstracta *MAPiCO*, que son almacenadas en memoria mediante una estructura que contiene un identificador único y el nombre de la instrucción.

```
struct str_Instruction
{
   int KeyPlugIn;       //  Llave para identificar la instrucciOn
   char *NameFunction;  // Nombre de la funciOn
};

//Tipo para definir una InstrucciOn
typedef struct str_Instruction *Instruction;
```

Cada instrucción es guardada en un árbol binario ordenado que facilita la inserción y la búsqueda de las mismas.

```
struct NodeTree
```

```
{
   int key;                  // Identificador del nodo
   void *Value;              // Valor del nodo
   struct NodeTree *Left;    // Hijo izquierdo del nodo
   struct NodeTree *Right;   // Hijo derecho del nodo
};

//Tipo para definir un 'arbol binario
typedef struct NodeTree *NodeTree;
```

Para pasar información a los *PlugIn's* se definió una estructura donde se almacena principalmente las variables que necesitan las instrucciones, los *PlugIn's* y el árbol de instrucciones cargados en la inicialización; además se almacena el *Store* utilizado por la Máquina Abstracta *MAPiCO*. La Estructura para el paso de parámetros es la siguiente:

```
struct str_Parameters
{
   List VblesMAPiCO;         //Lista de variables
   NodeTree Instructions;    //Arbol de Instrucciones
   PlugIn *PlugIns;          //Array con los apuntadores a los Plugin's cargados
   int NumPlugInsLoad;       //Numero de Plugin's que se cargaron en la ejecucion
   int Response;             //Codigo de respuesta de la accion
   char *Error;              //String con el mensaje de error
   int OpCodeSchedule;       //Codigo OpCode de la funcion que realiza el Schedule
   StoreMA sMA;              //Store MAPiCO
};

// Tipo para definir los parametros
typedef struct str_Parameters Parameters;
```

Se realizaron pruebas exitosas con respecto a la carga de las diferentes estructuras de *MAPiCO*: Carga de archivos de programa de gran tamaño en la memoria estática y de *PlugIn's* en tiempo de ejecución; estas pruebas determinaron una gran capacidad de almacenamiento y eficiencia en la extensibilidad de las instrucciones de *MAPiCO*. En la Tabla 4.1 se presentan los tiempos de carga de las diferentes estructuras para los programas Factorial, SEND + MORE = MONEY (SMM) y NReinas (ver capitulo 5):

4.4. Bloque Principal de la Máquina

Todos los *PlugIn's* de la Máquina Abstracta *MAPiCO* son invocados desde el ejecutable *MAPiCO.c*, ahí se encuentran definidos los registros de la máquina, las colas de pro-

Programa	Información de Cargue			
	Cargue ProgramBytes	Ejecución ProgramBytes	Cargue PlugIn's	Tiempo Total
Factorial				
0	0	0,0008	0,0001	0.0009
1	0	0,003	0,0001	0,0031
2	0	0,005	0,0001	0,0051
3	0	0,008	0,0001	0.0081
4	0	0,011	0,0001	0.0111
5	0	0,014	0,0001	0.0141
6	0	0,020	0,0001	0.0201
7	0	0,023	0,0001	0.0231
SMM	0,001	0,068	0,0001	0.0691
NReinas	0,0001	0,0227	0,0001	0,0229

Tabla 4.1: Tiempos de Carga en milisegundos de Estructuras de *MAPiCO*

cesos suspendidos, las colas de ejecución, la memoria de traducción y la memoria del programa. Los registros de la máquina son declarados como tipo void * (puntero genérico) permitiendo que estos registros puedan ser configurados por fuera de la máquina para almacenar cualquier tipo de información.

```
int Execute(char *fileName)
{
    ////////////////////////////
    // DECLARACION DE VARIABLES //
    ////////////////////////////

    //Vbles para manejo de PlugIn's
    short OpCode=0;                          // OpCode de la instruccion
    Configuration VarConfig;                 // Variable de configuracion inicial
    Parameters (*Instruction)(Parameters);   // Funcion de la instruccion
    Parameters argvPlugIn;                   // Parametros para los PlugIn's

    //Declaracion de colas
    List RunQ;                               // Cola de Ejecucion
    List ObjQ;                               // Cola de Objetos
    List MsgQ;                               // Cola de Mensajes
    List AskQ;                               // Cola de Ask

    //Registros de MAPiCO
    int PCA;                                 // Puntero a Codigo Actual
    void *PVA;                               // Puntero a Variables Actual
    void *PMA;                               // Puntero a Metodos Actual
```

```
    void *PAA;                                  // Puntero a Argumentos Actual
    void *POA;                                  // Puntero a Objetos Actual
    void *PAUX;                                 // Puntero Auxiliar
    void *PTEMP;                                // Puntero Temporal

    //Vble para definir procesos
    Proceso Proc;

    //Vble que contiene el ProgramByte
    Program Data;

    ////////////////////////////////
    // CARGAR CONFIGURACION INICIAL //
    ////////////////////////////////
    VarConfig = loadConfiguration();

    //Inicializador de Parametros
    argvPlugIn = newParameters(VarConfig.MaxOpenPlugIns);

    //Lectura de todo el archivo de entrada
    Data = Loader(fileName);

    ////////////////////////////////
    // CARGA DE TODOS LOS PLUGIN's //
    ////////////////////////////////
    argvPlugIn = loadPlugIns(argvPlugIn, VarConfig.PathPlugIns);
    if(getResponse(argvPlugIn)==ERROR)
    {
       printf("Carga de PlugIns. [ERROR]: \%s", getError(argvPlugIn));
       exit(1);
    }

    ...

}
```

En *MAPiCO.c* se definió una función *loadConfiguration* que carga la configuración inicial de la máquina, la cual se encuentra en el archivo *MAPiCO.conf*, que contiene el OpCode de la instruccíon que realiza el cambio de proceso, la ruta donde se encuentra el archivo con la lista de los *PlugIn's* y el máximo número de *PlugIn's* que se cargaran. La opción *schedule* no es obligatoria definirla en la configuración ya que se puede configurar en los *PlugIn's*.

```
#Archivo de Configuración de MAPiCO

#OpCode de la funcion que realizara el Schedule de los procesos
schedule= OPCODE_SCHED

#Path donde se encuentra archivo que contiene las rutas de los Plugin's
pathPlugInList="../doc/PlugIns.list"

#Maximo numero de Plugin's
maxOpenPlugIns=5
```

```
// Lee la configuraciOn de MAPiCO.conf y la guarda en memoria
Configuration loadConfiguration(void)
{
   Configuration varConfig;
   FILE *FileConfig;
   char Line[MAXIMO];
   char *Value, *Key;
   int TamLine=0;
   int Pos=0;

   varConfig.OpCodeSchedule = NOT_FOUND;

   //Abrir archivo de configuracion MAPiCO.conf
   if((FileConfig=fopen("../doc/MAPiCO.conf","r"))==NULL)
   {
      printf("[ERROR]: \%s.\n",NOT_FILE_CONFIG);
      exit(1);
   }
   else
   {
         while(!feof(FileConfig))
         {
            fgets(Line, MAXIMO, FileConfig);

            if(Line[0]!='#' && !iscntrl(Line[0]))
            {
               //Se Elimina el salto de linea
               TamLine = strlen(Line);
               if(iscntrl(Line[TamLine-1]))
                  Line[TamLine-1]='\0';
               else
                  Line[TamLine]='\0';

               //Posicion donde se encuentra el separador "="
               Pos = strcspn(Line,"=");
```

```
                Key = (char *)malloc(sizeof(char)*Pos);
                Value  = (char *)malloc(sizeof(char)*(TamLine - Pos));

                strncpy(Key, Line, Pos);
                strncpy(Value, (Line + Pos + 1), (TamLine - Pos));

                if(strcmp(Key,"schedule")==0)
                   varConfig.OpCodeSchedule = atoi(Value);
                else
                {
                   if(strcmp(Key,"pathPlugInList")==0)
                     strncpy(varConfig.PathPlugIns,(Line+Pos+1),(TamLine-Pos));
                   else
                   {
                      if(strcmp(Key,"maxOpenPlugIns")==0)
                         varConfig.MaxOpenPlugIns = atoi(Value);
                   }
                }//else::if(strcmp(schedule...
             }//if(Line[0]...
          }//while
    }//else

    return varConfig;
}
```

Una vez cargados el archivo de entrada o program bytes, los *PlugIn's* y sus instrucciones e inicializadas las colas de ejecución, se crea la lista de variables que servirá de parámetro de entrada y de salida a las instrucciones de la máquina, en ese momento se adicionan las colas de ejecución, el program bytes, el contador de variables, el proceso inicial, el opcode de *schedule* y un flag de finalización de *MAPiCO*.

```
    //////////////////////////////////////////////////////
    // CREACION DE LA LISTA DE VARIABLES PARA LOS PLUGIN'S //
    //////////////////////////////////////////////////////
    //Se agrega el flag Finish que indica si MAPiCO debe seguir ejecutandose
    argvPlugIn = addVblePlugIn(argvPlugIn, Finish);

    //Se agrega el proceso a la lista de variables de los PlugIn's
    argvPlugIn = addVblePlugIn(argvPlugIn, Proc);

    //Se agrega el contador de variables a la lista de variables de los PlugIn's
    argvPlugIn = addVblePlugIn(argvPlugIn, VarCounter);

    //Se agrega el program bytes a la lista de variables de los PlugIn's
    argvPlugIn = addVblePlugIn(argvPlugIn, Data);

    //Se agregan las colas de ejecucion RunQ, ObjQ, MsgQ, AskQ
```

```
argvPlugIn = addVblePlugIn(argvPlugIn, RunQ);
argvPlugIn = addVblePlugIn(argvPlugIn, ObjQ);
argvPlugIn = addVblePlugIn(argvPlugIn, MsgQ);
argvPlugIn = addVblePlugIn(argvPlugIn, AskQ);

//Se agrega el codigo de la funciOn que realiza el Schedule
argvPlugIn = setOpCodeSchedule(argvPlugIn, VarConfig.OpCodeSchedule);
```

Con la lista de parámetros cargada, inicia el ciclo principal de ejecución que solo termina cuando la cola de ejecución *RunQ* esta vacia, condición que es controlada por la variable "*Finish*", la cual cambia de estado en la función *schedule*. Dentro del ciclo, se realiza la selección de la instrucción que se debe ejecutar por medio de la función *getInstruction*, la cual retorna un apuntador a la función de la instrucción, pasandole la lista de parámetro para su ejecución.

```
////////////////////////////////
// EJECUCION DEL PROGRAM BYTES //
////////////////////////////////
while(*Finish != OK)
{

   //Captura los valores de los registros
   PCA = getPC(Proc);

   //Lectura del program bytes
   OpCode = get8(Data, PCA);

   ////////////////////////////////////////////
   // BUSQUEDA Y EJECUCI'ON DE LA INSTRUCCION //
   ////////////////////////////////////////////
   // Busqueda de la instrucciOn
   Instruction = getInstruction(argvPlugIn, OpCode);

   // Ejecuci'on de la instrucciOn enviando la lista de parametros
   argvPlugIn = Instruction(argvPlugIn);

   ////////////////////////////////////////////
   //RECUPERACION DE LAS VARIABLES DE MAPiCO //
   ////////////////////////////////////////////
   // Recupera el flag de ejecuci'on
   Finish = getAtVblePlugIn(argvPlugIn,0);

   // Recupera el proceso actual
   Proc = getAtVblePlugIn(argvPlugIn,1);

   // Recupera el PC
   PCA = getPC(Proc);
```

```
}//while
```

4.5. Implementación de Instrucciones

El ciclo de ejecución principal consta de una serie de instrucciones que la máquina ejecuta de acuerdo a las reglas de reducción de la máquina.

La instrucción **NEWV** crea un nuevo nodo en el árbol de traducción de variables y actualiza el **PVA**.

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

FUNC determina el inicio de la ejecución de una función.

Hay tres instrucciones para manejar el paso de parámetros. Dos de ellas, **PUSHV** y **PUSHM**, adicionan parámetros a la pila de parámetros (**PUSHV** adiciona variables y **PUSHM** adiciona métodos) y **POP** obtiene los parámetros de la pila.

```
Parameters PUSHV(Parameters argv)
{
  //Registros que se necesita de MAPiCO
  int     PCA;            //Puntero a Codigo Actual
  NodeList PVA;           //Puntero a Variables Actual
  Auxiliar PAA;           //Puntero a Argumentos

  Proceso Proc;           //Vble para definir un proceso
  Program Data;           //Vble para el program bytes
  NodeArg NodeTempArg;    //Vble para guardar el parametro

  //Otras Variables
  int Longitud=0;
  int *Variable;
  int ArgumentoPushv;
  int *Finish;
  NodeArg FPAUX;

  //Recuperar el primer parametro (Proceso)
  Proc = getAtVblePlugIn(argv,1);
```

```
//Recuperar el tercer parametro (Data)
Data = getAtVblePlugIn(argv,3);

//Recuperar el valor de los registros
PCA = getPC(Proc);
PVA = getPV(Proc);
PAA = getPA(Proc);

Variable = (int *)malloc(sizeof(int));
LENGTH(PVA, Longitud);
ArgumentoPushv = get16(Data, PCA+1);

if(Longitud >= ArgumentoPushv)
{
  //Valor de Variable de la lista del PV
  GETDATAAT(PVA, ArgumentoPushv, Variable);
  NODEARG(NodeTempArg, Variable, VAR);

  if(GETTYPE(PAA)==NOT_TYPE)
  {
    //Salvar argumentos en PAA
    SETNODEARG(PAA, NodeTempArg);
    SETTYPE(PAA, NODEARGM);
  }
  else
  {
    //Agregar el nuevo argumento
    FPAUX = GETNODEARG(PAA);
    SETNEXT_ARG(NodeTempArg, FPAUX);
    SETNODEARG(PAA, NodeTempArg);
    SETTYPE(PAA, NODEARGM);
  }

  //Incrementar el PCA (siguiente instruccion)
  PCA = PCA + INST_BYTE + VAR_BYTE;

  //Asignacion de nuevos valores
  setPC(Proc, PCA);
  setPA(Proc, PAA);
}
else
{
  Finish = (int *)malloc(sizeof(int));
  *Finish = OK;
  argv = setAtVblePlugIn(argv,Finish,0);
}

//Guardar proceso en la lista de vbles
argv = setAtVblePlugIn(argv,Proc,1);

//Retorno de argumentos
return argv;
```

}

Los procesos objeto pueden definirse como replicados **COBJCC** o no replicados **OBJCC**, por tal razón se cuenta con instrucciones que insertan el objeto en la cola $RunQ$ y los mensajes que se encuentran en $MsgQ$ los adiciona en la cola de ejecución.

La instrucción **CALL** evalua si la cola de objetos $ObjQ$ contiene elementos, si no los tiene, el proceso es insertado en la cola de mensajes suspendidos $MsgQ$, en caso contrario selecciona el primer objeto que aún no haya sido evaluado para el mensaje, (obteniendo de sus parámetros el **PC** donde inicia la función *sender* del objeto, si el objeto es replicado se conservará en $ObjQ$, de lo contrario será eliminado de la misma y creará un nuevo proceso en $RunQ$ para evaluar el mensaje.

La instrucción **COMM** busca el método al que hace referencia el mensaje en el objeto si lo encuentra crea un nuevo proceso en $RunQ$ apuntando a la primera instrucción del método requerido con el **PV**, **PM**, **PO**, y **PA** del proceso actual. Si no encuentra el método se finaliza la ejecución de la instrucción **COMM**.

La instrucción **REE** crea un nuevo proceso en $RunQ$ apuntando a la primera instrucción de la función $forward$ del objeto.

Para el manejo de restricciones hay varias instrucciones, algunas de ellas como **TERMV**, **TERMC**, **TERMF**, **ATOM** son para construir las restricciones y otras como **TELL**, **ASK** y **ASKF** para adicionar y evaluar la información del Sistema de Restricciones.

Las instrucciones **TERMV**, **TERMC**, **TERMF**, **ATOM** tienen el mismo comportamiento pero su diferencia radica en la información que almacenan.

```
Parameters TERMV(Parameters argv)
{
  int       PCA;    //Program Counter
  NodeList  PVA;    //Program Variable
  Auxiliar  PAA;    //Program Argument Actual
  Auxiliar  PAUX;   //Puntero Auxiliar
  Proceso   Proc;   //Vble para definir procesos
  //Otras Variables
  int VariableTERMV, Longitud;
  FrameConstMA faux, FPAUX;
```

```
Program Data;
int *Variable;
int *Finish;

Var = faux = FPAUX = NULL;
Variable = (int *)malloc(sizeof(int));

//Recuperacion del primer parametro
Proc = getAtVblePlugIn(argv,1);

//Recuperacion del tercer parametro
Data = getAtVblePlugIn(argv,3);

//Se recuperan los valores de los registros
PCA =  getPC(Proc);
PVA =  getPV(Proc);
PAA = getPA(Proc);
PAUX =  getPAUX(Proc);

//Longitud de la lista de Variable
LENGTH(PVA,Longitud);
VariableTERMV = get16(Data, PCA+1);

if(Longitud >= VariableTERMV) {
  //Valor de Variable de la lista del PV
  GETDATAAT(PVA, VariableTERMV, Variable);

  //Se recupera el FrameConst que contiene el PAUX
  FPAUX = GETFRAMECONST(PAUX);

  //Validacion del FrameConst de PAUX
  argv = ValidFrameConst(argv, FPAUX);
  if(getResponse(argv)==ERROR) {
    return argv;
  }

  faux = InicFrameConstMA(VARIABLEMA, *Variable);

  if(getLefSonFrameConstMA(FPAUX)==NULL) {
       setLefSonFrameConstMA(FPAUX, faux);
  }
  else {
    setRigSonFrameConstMA(FPAUX, faux);

    if(GETFRAMECONST(PAA)==GETFRAMECONST(PAUX)) {
      SETFRAMECONST(PAUX, faux);
      SETTYPE(PAUX, FRAMECONST);
    }
  }

  //Incrementar el PCA para la siguiente instruccion
  PCA = PCA + INST_BYTE + VAR_BYTE;
```

```
    faux = NULL;
    //Asignacion de nuevo valor del Frame al que debe apuntar PAUX
    SETFRAMECONST(PAUX, FPAUX);
    SETTYPE(PAUX, FRAMECONST);

    //Asignacion de los nuevos valores
    setPC(Proc, PCA);
    setPAUX(Proc, PAUX);

    //Se guardan las variables en la lista de vbles de los PlugIn's
    argv = setAtVblePlugIn(argv,Proc,1);
  }
  else {
     Finish = (int *)malloc(sizeof(int));
     *Finish = OK;
     argv = setAtVblePlugIn(argv,Finish,0);
  }

  //Retorno de los argumentos
  return argv;
}
```

Las instrucciones **ASK**, **ASKF** y **TELL**, evaluan y adicionan información al Sistema de Restricciones. Todas usan el apuntador **PAA** como referencia al árbol de restricciones.

```
Parameters TELL(Parameters argv)
{
   ...

   sMA = tellMA(sMA, FPAA);

   if(LENGTH_LIST(MsgQ)!=0)
   {
      ADD_AT_END(RunQ,MsgQ);
      LIST(MsgQ);
   }

   if(LENGTH_LIST(AskQ)!=0)
   {
      ADD_AT_END(RunQ,AskQ);
      LIST(AskQ);
   }

   ...

   return argv; //Retorno de argumentos
}
```

La instrucción **ASK** evalua la información (respuesta) del sistema de restricciones: si el sistema responde *SUSPENDED* quiere decir que no puede deducir ni verdadero ni falso, entonces el proceso es adicionado a la cola **AskQ**; si el sistema retorna *UNENTAILED* (falso) continua con la ejecución de otro proceso y el proceso actual es eliminado; y si responde *ENTAILED* (verdadero) continua con la siguiente instrucción del proceso actual.

```
Parameters ASK(Parameters argv)
{
    ...

   sMA = askMA(sMA, FPAA);

   switch (getResponseMA(sMA))
   {
      case ENTAILED_MA:
         //Incrementar PC
         PCA++;
         break;
      case UNENTAILED_MA:
         //Buscar instruccion encargada de realizar el schedule
         Instruction = getInstruction(argv, getOpCodeSchedule(argv));

         //Ejecucion de la instruccion
         argv = Instruction(argv);
         break;
       case SUSPENDED_MA:
         if(LENGTH_LIST(AskQ)!=0) {
            ADD_AT_END(RunQ,AskQ);
            LIST(AskQ);
         }
         break;
    }

    ...

   return argv; //Retorno de argumentos
}
```

4.6. Módulos Dinámicos - Extensión de Instrucciones

Las librerías dinámicas son una forma práctica de construir software extensible y modular, estas son utilizadas para implementar $PlugIn's$ o módulos que son cargados por un pro-

grama en tiempo de ejecución.

Las instrucciones que ejecuta *MAPiCO* fueron desarrolladas utilizando librerías dinámicas (o $PlugIn's$) permitiendo que el ejecutable de la máquina sea mas pequeño y que la modificación de las instrucciones se pueda realizar sin tener que recompilar la maquina. Otra ventaja que tiene la implementación de las instrucciones como $PlugIn's$ es, que si la maquina es invocada concurrentemente se cargara solo una vez el código de las instrucciones ahorrando espacio en memoria.

Para el desarrollo de los $PlugIn's$ se evaluaron las librerías de **Guile**, **Gnome** y **Glibc**, optando por hacerlos por medio de **Glibc** debido a que **Gnome** y **Guile** la utilizan como base para sus librerías y también porque **Glibc** es una librería portable para otros sistemas operativos, como $MacOSX$, $UNIX(Linux)$ y $Windows(Cygwin)$.

Para acceder a los $PlugIn's$ por medio de **Glibc** se utilizan las funciones *dlopen()* que abre una librería dinámica, *dlerror()* que informa del último error ocurrido, *dlsym()* que se utiliza para encontrar un simbolo en la librería apuntada por el manejador obtenido en dlopen() y *dlclose()* que descarga la librería dinámica referenciada.

En la función *loadPlugIns* de la librería *PlugIns.c*, de *MAPiCO*, se utiliza las funciones de **Glibc** para realizar el cargue de los $PlugIn's$ que la máquina tiene configurados.

```
1. Parameters loadPlugIns(Parameters argv, char *PathPlugInList)
2. {
3.   FILE *FileConfig;
4.   void *Handle;
5.   char *Error;
6.   char PathLibrary[MAX_PATH];
7.   int TamPathLibrary, NumPlugIn = 0;
8.
9.   Parameters (*Catalogue)(Parameters, char *, void *, int);
10.
11.  //Abrir archivo de configuracion
12.  if((FileConfig=fopen(PathPlugInList,"r"))==NULL)  {
13.    printf("[ERROR]: No se pudo cargar el abrir el Archivo de PlugIn's\n");
14.    exit(1);
15.  }
16.
17.  while(!feof(FileConfig)) {
18.    //Leer cada uno de los nombres de los PlugIn's
```

```
19.     fgets(PathLibrary ,MAX_PATH, FileConfig );
20.
21.     if(PathLibrary[0]!='#') {
22.       //Eliminar salto de linea
23.       TamPathLibrary = strlen(PathLibrary);
24.
25.       if(iscntrl(PathLibrary[TamPathLibrary-1]))
26.         PathLibrary[TamPathLibrary-1]='\0';
28.       else
27.         PathLibrary[TamPathLibrary]='\0';
28.
29.       //Abrir PlugIn
30.       Handle = dlopen(PathLibrary , RTLD_LAZY);
31.
32.       //Validar error
33.       Error = dlerror();
34.       if(Error) {
35.         argv = setResponse(argv,ERROR);
36.         argv = setError(argv,NOT_OPEN_LIBRARY);
37.         fclose(FileConfig);
38.         return argv;
39.       }
40.
41.       //Localizar funcion Catalogue del plugin
42.       Catalogue = dlsym(Handle,"Catalogue");
43.
44.       //Validar error
45.       Error = dlerror ();
46.       if(Error) {
47.         argv = setResponse(argv,ERROR);
48.         argv = setError(argv,NOT_CATALOGUE);
49.         fclose(FileConfig);
50.         return argv;
51.       }
52.       NumPlugIn++;
53.       argv = Catalogue(argv,PathLibrary,Handle,NumPlugIn);
54.     }
55.   }
56.   fclose(FileConfig);
57.   return argv;
58.}
```

En la línea 9, se declara un apuntador a una función, que corresponde a la función **Catalogue** que debe tener todo *PlugIn MAPiCO* (**Catalogue**, se detalla en el *Anexo B. Guía de Adición de PlugIn's - Extensibilidad.*

En la línea 17, se declara un ciclo el cual recorrerá línea por línea el archivo que contiene la lista con las rutas de los *PlugIn's*. Cada ruta es pasada como parámetro a la función

dlopen definida en la línea 30, la cual retorna un apuntador o "Handle"de la librería; con este apuntador se busca la función *Catalogue* del $PlugIn$ (línea 42), el cual retorna el apuntador a la función definida en la línea 9 y con esto ya se puede ejecutar la función **Catalogue** del $PlugIn$ pasandole los párametros definidos en la función.

Con el ejemplo anterior se puede observar que el uso de $PlugIn's$ es muy sencillo y ofrece el gran potencial a la máquina de adicionar, eliminar o modificar instrucciones sin necesidad de afectar ni una sola línea de código.

4.7. Complejidades de Funciones

En la siguiente Tabla se presenta un comparativo de las complejidades de las funciones implementadas en la primera versión *MAPiCO JAVA* y en la versión actual *MAPiCO C*.

Tabla de Complejidades			
Nombre TAD	**Nombre Función**	**MÁQUINAS**	
		Máquina Anterior (MAPiCO JAVA)	**Máquina Actual (MAPiCO C)**
List	LENGTH_LIST	O(1)	O(1)
	INSERT_AT_HEAD	O(1)	O(1)
	INSERT_AT_END	O(1)	O(1)
	FIRST_NODE	O(1)	O(1)
	END_NODE	O(1)	O(1)
	ADD_AT_END	O(1)	O(1)
	INSERT_AT	O(n)	O(n)
	DEL_FIRST_NODE	N/A	O(1)
	FREE_LIST	N/A	O(1)
	DEL_AT	O(n)	O(1)
	GET_AT	O(n)	O(1)
	SET_AT	O(n)	O(1)
	FREE_MEMORY_LIST	N/A	O(n)
	PRINT_LIST	N/A	O(n)
NodeList	INIT_NODELIST	N/A	O(1)
	NODELIST_CONTINUATION	O(1)	O(1)
	SETNEXT	O(1)	O(1)
	SETDATA	O(1)	O(1)
	GETNEXT	O(1)	O(1)
	GETPREV	O(1)	O(1)
	GETDATA	O(1)	O(1)
	GETDATAAT	O(n)	O(n)
	LENGTH	O(n)	O(n)
	FREE_MEMORY_NODELIST	N/A	O(n)
	FREE_NODE	N/A	O(1)
	PRINT_NODELIST	N/A	O(n)
Loader	Loader	O(1)	O(1)
	Get8	O(1)	O(1)
Continua en la próxima página			

Tabla 4.2 Complejidades de Funciones – continua desde la anterior página

Nombre TAD	Nombre Función	MÁQUINAS	
		Máquina Anterior (MAPiCO JAVA)	Máquina Actual (MAPiCO C)
	Get16	O(1)	O(1)
	Get32	O(1)	O(1)
NodeArg	NODEARG	N/A	O(1)
	NODE_CONTINUATION_ARG	O(1)	O(1)
	SETNEXT_ARG	O(1)	O(1)
	SETDATA_ARG	O(1)	O(1)
	GETNEXT_ARG	O(1)	O(1)
	ISVALIDNEXT_ARG	N/A	O(1)
	GETTYPE_ARG	N/A	O(1)
	GETPREV_ARG	O(1)	O(1)
	GETDATA_ARG	O(1)	O(1)
	GETDATAAT_ARG	O(n)	O(n)
	LENGTH _ARG	O(n)	O(n)
	FREE_MEMORY_NODELIST_ARG	N/A	O(n)
	FREE_NODE_ARG	N/A	O(1)
	PRINT_NODE_ARG	N/A	O(n)
Process	InitProcess	N/A	O(1)
	NewProcess	N/A	O(1)
	SetPC	O(1)	O(1)
	SetPV	O(1)	O(1)
	SetPM	O(1)	O(1)
	SetPA	O(1)	O(1)
	SetPO	O(1)	O(1)
	SetPAUX	O(1)	O(1)
	SetPTEMP	O(1)	O(1)
	GetPC	O(1)	O(1)
	GetPV	O(1)	O(1)
	GetPM	O(1)	O(1)
	GetPA	O(1)	O(1)
	GetPO	O(1)	O(1)
	GetPAUX	O(1)	O(1)
	GetPTEMP	O(1)	O(1)
	DelProcess	N/A	O(1)
StoreMA	InitStoreMA	N/A	O(1)
	TellMA	O(1)	O(1)

Continua en la próxima página

Tabla 4.2 Complejidades de Funciones – continua desde la anterior página

Nombre TAD	Nombre Función	MÁQUINAS	
		Máquina Anterior (MAPiCO JAVA)	Máquina Actual (MAPiCO C)
	AskMA	O(1)	O(1)
	InicFrameConstMA	N/A	O(1)
	InicpFrameConstMA	N/A	O(1)
	GetLefSonFrameConstMA	N/A	O(1)
	GetRigSonFrameConstMA	N/A	O(1)
	SetLefSonFrameConstMA	N/A	O(1)
	SetRigSonFrameConstMA	N/A	O(1)
	DestroyFrameConstMA	N/A	O(1)
	PrintFrameConstMA	N/A	O(1)
	GetResponseMA	N/A	O(1)
	GetInicMA	N/A	O(1)
	FreeStoreMA	N/A	O(1)
BinaryTree	NewNode	N/A	O(1)
	InsertNode	N/A	O(1)
	LeftNode	N/A	O(1)
	RightNode	N/A	O(1)
	SearchNode	N/A	O(log2n)
	Inorden	N/A	O(log2n)
	FreeTree	N/A	O(log2n)
HeapSort	SearchBinary	N/A	O(log n)
	Exchange	N/A	O(1)
	Left	N/A	O(1)
	Right	N/A	O(1)
	Heapify	N/A	O(log n)
	Build_Heap	N/A	O(log n)
	HeapSort	N/A	O(log n)
ObjectEval	INICOBJECTEVAL	N/A	O(1)
	GETNUMOBJECTS	N/A	O(1)
	SETNUMOBJECTS	N/A	O(1)
	DELPCS	N/A	O(1)
	GETPCS	N/A	O(1)
	GET_PC_AT	N/A	O(1)
	SAVE_PC	N/A	O(1)
	PRINT_PCS	N/A	O(n)
Parameters	NewParameters	N/A	O(1)

Continua en la próxima página

Tabla 4.2 Complejidades de Funciones – continua desde la anterior página

Nombre TAD	Nombre Función	MÁQUINAS	
		Máquina Anterior (MAPiCO JAVA)	Máquina Actual (MAPiCO C)
	addVblePlugIn	N/A	O(1)
	setAtVblePlugIn	N/A	O(1)
	getAtVblePlugIn	N/A	O(1)
	setHandle	N/A	O(1)
	getHandle	N/A	O(1)
	setPathLibrary	N/A	O(1)
	getPathLibrary	N/A	O(1)
	setNameFunction	N/A	O(1)
	getNameFunction	N/A	O(1)
	setResponse	N/A	O(1)
	getResponse	N/A	O(1)
	setOpCodeSchedule	N/A	O(1)
	getOpCodeSchedule	N/A	O(1)
	setStoreMA	N/A	O(1)
	getStoreMA	N/A	O(1)
	setError	N/A	O(1)
	getError	N/A	O(1)
	setNumPlugInsLoad	N/A	O(1)
	getNumPlugInsLoad	N/A	O(1)
	isInstruction	N/A	O(1)
	getIndexPlugIn	N/A	O(1)
	freeParameters	N/A	O(n)
PlugIns	loadInstructions	N/A	O(n)
	loadPlugIns	N/A	O(n)
	getInstruction	N/A	O(1)
	closePlugIns	N/A	O(n)
Auxiliar	INICAUXILIAR	N/A	O(1)
	SETNODEARG	N/A	O(1)
	SETFRAMECONST	N/A	O(1)
	GETTYPE	N/A	O(1)
	SETTYPE	N/A	O(1)
	GETNODEARG	N/A	O(1)
	GETFRAMECONST	N/A	O(1)
	FREEFRAMECONST	N/A	O(1)
	FREEAUX	N/A	O(1)

5 PRUEBAS Y RESULTADOS

A continuación se presentan las pruebas realizadas a la reimplementación de la Máquina Abstracta *MAPiCO*, estableciendo una serie de comparaciones con la primera implementación que permitan comprobar una mejora sustancial de la eficiencia reflejada en tiempos mínimos de ejecución.

Para establecer comparaciones entre la anterior y la nueva implementación de *MAPiCO*, se realizaron pruebas en una máquina *AMD AThlon* a 32 bits con 512 MB en RAM en tres diferentes escenarios:

- **Escenario 1**: Máquina Abstracta *MAPiCO en JAVA* y Sistema de Restricciones en *JAVA*.
- **Escenario 2**: Máquina Abstracta *MAPiCO en JAVA* y Sistema de Restricciones en *C*.
- **Escenario 3**: Máquina Abstracta *MAPiCO en C* y Sistema de Restricciones en *C*.

Denotación referente a los diferentes escenarios:

- *MAPICO JAVA*: Máquina Abstracta *MAPiCO* en *JAVA*
- *MAPICO C*: Máquina Abstracta *MAPiCO* en *C*
- *Store JAVA*: Sistema de Restricciones en *JAVA*
- *Store C*: Sistema de Restricciones en *C*

Se eligieron programas de prueba que implicaran trabajo de la Máquina Abstracta y el Sistema de Restricciones, para tal fin se evaluaron los programas de pruebas de [GV01] y se seleccionaron tres de ellos:

- FACTORIAL
- SEND + MORE = MONEY
- NREINAS

Los programas fueron codificados en el Cálculo *PiCO* Anterior para los **Escenarios 1 y 2** y en el Cálculo *PiCO* Actual para el **Escenario 3**. Los resultados para cada caso son los siguientes:

5.1. FACTORIAL

Al ejecutar el programa Factorial en los diferentes escenarios, se obtuvieron los siguientes tiempos de ejecución en milisegundos (ver Tabla 5.1):

Programa Factorial	**Escenarios de Pruebas**		
	Escenario 1 *Máquina JAVA / Store JAVA*	**Escenario 2** *Máquina JAVA / Store C*	**Escenario 3** *Máquina C / Store C*
0	80	32	0.0009
1	104	34	0.003
2	125	54	0.005
3	182	56	0.008
4	200	59	0.011
5	215	66	0.014
6	244	75	0.020
7	274	81	0.023

Tabla 5.1: Tiempos de Ejecución en milisegundos del Programa Factorial del 0 al 7

Para los diferentes escenarios de pruebas, los resultados de ejecución del programa Factorial para los números del 1 al 7 fueron los esperados, pero en eficiencia considerando

los tiempos de ejecución, se puede apreciar una disminución significativa en el **Escenario 3**, que hace referencia a la reimplementación de *MAPiCO en C* (ver figura 5.1 con respecto a datos de la Tabla 5.1). Cabe mencionar que no se realizaron pruebas de Factorial en números superiores a 7 por una limitante del Sistema de Restricciones, ya que este solo soporta variables de tipo entero (int).

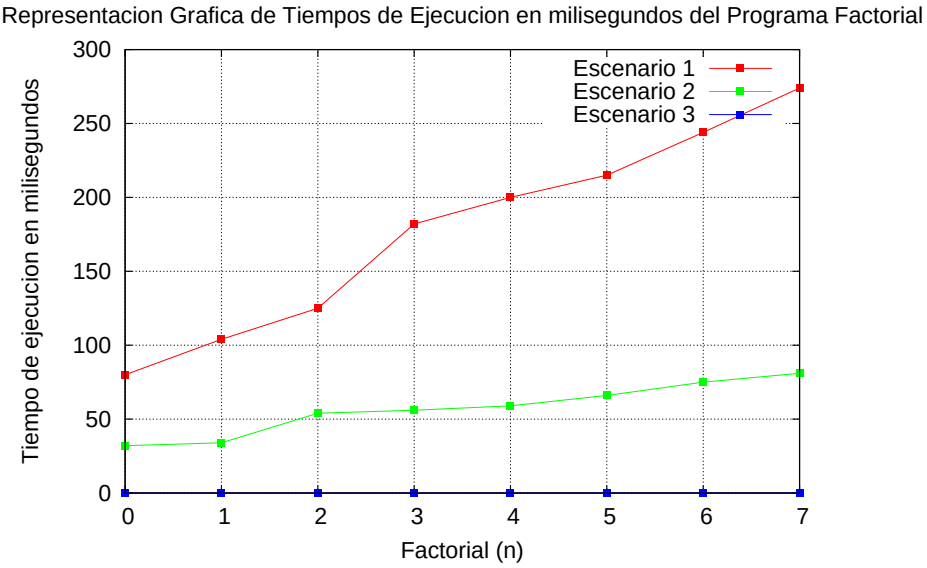

Figura 5.1: Representación Gráfica de Tiempos de Ejecución en milisegundos del Programa Factorial

5.2. SMM: SEND + MORE = MONEY

SEND + MORE = MONEY es un puzzle criptoaritmético de solución única, en el cual la suma SEND + MORE debe ser igual a MONEY, considerando cada letra de esta ecuación como un digito independiente de la siguiente forma:

$$\begin{array}{r} \text{SEND} \\ +\text{MORE} \\ \hline \text{MONEY} \end{array} \qquad \begin{array}{r} 9567 \\ +\ 1085 \\ \hline 10652 \end{array}$$

Los tiempos de ejecución en milisegundos son los siguientes (Ver Tabla 5.2):

Programa SMM	**Escenarios de Pruebas**		
	Escenario 1 *Máquina JAVA / Store JAVA*	**Escenario 2** *Máquina JAVA / Store C*	**Escenario 3** *Máquina C / Store C*
$TELL\ Y = 2$	533	132	0.0690

Tabla 5.2: Tiempos de Ejecución en milisegundos del Programa *SMM*

En esta prueba se fijo la variable Y con el valor de la constante 2 ($TELL\ Y = 2$) y se obtuvieron los siguientes dominios para las variables S, E, N, D, M, O, R, Y en el **Escenario 3** que involucra a *MAPiCO en C* (ver Tabla 5.3):

$$\begin{array}{lll} S & in & \{9,9\} \\ E,D,R & in & \{3,8\} \\ N & in & \{3,6\} \\ M & in & \{1,1\} \\ O & in & \{0,0\} \\ Y & in & \{2,2\} \end{array}$$

Tabla 5.3: Resultados del Programa *SMM* en el **Escenario 3**

Los resultados del **Escenario 3** son mucho mas cercanos a la única solución, pero diferentes a los del **Escenario 2** [GV01], tomando en consideración esto, se confirma el planteamiento de la variación de los resultados debido al manejo de los conceptos de arcoconsistencia parcial y a la propagación de información sobre mínimos y máximos usados en el módelo teórico del Sistema de Restricciones [GV01].

Hay que resaltar que los resultados del **Escenario 1** para las pruebas del *SMM* y *NREINAS* no fueron los esperados, del Store o Sistema de Restricciones en *JAVA* no se pudo deducir ninguna solución, los dominios de las variables quedaron en cero, posiblemente debido a los errores en la implementación del modelo del Sistema de Restricciones Aritmético [GV01].

En la ejecución del programa *SMM* se puede apreciar que el **Escenario 3**, que involucra la reimplementación de *MAPiCO en C*, presenta el mínimo tiempo de ejecución (Ver figura 5.2 con respecto a datos de la Tabla 5.2).

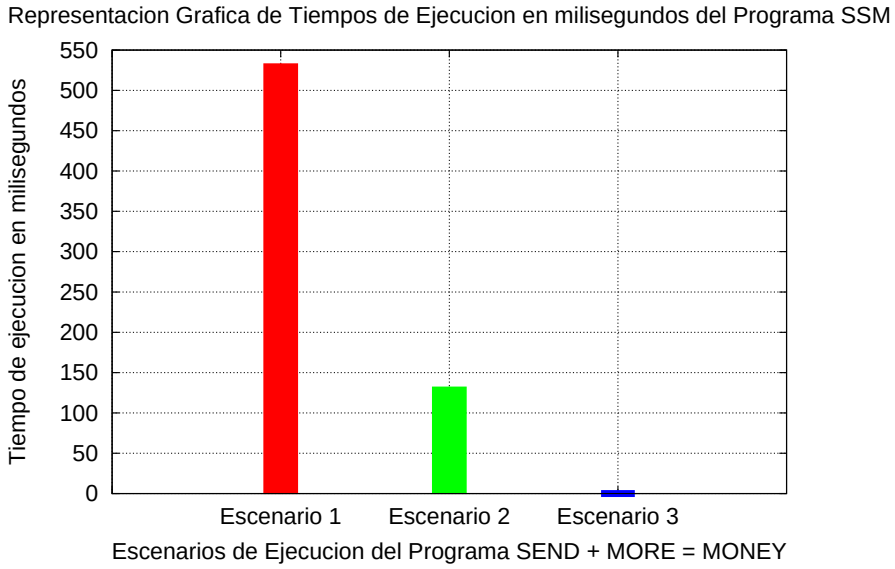

Figura 5.2: Representación Gráfica de Tiempos de Ejecución en milisegundos del Programa *SMM*

5.3. NREINAS

El problema *NREINAS* consiste en colocar n reinas en un tablero de nxn, de forma que no puede haber dos piezas en la misma línea horizontal, vertical o diagonal. Este problema es *NP - completo* y no tiene solución para $n = 2$ y $n = 3$, para $n = 4$ tiene solución única y para $n = 8$ tiene 80 soluciones dependiendo de las restricciones que se impongan.

En esta prueba, considerando las variables X_1, X_2, X_3, X_4 y determinado $X_3 = 4$, se obtuvieron los siguientes tiempos de ejecución en milisegundos (ver Tabla 5.4):

Programa NREINAS	**Escenarios de Pruebas**		
	Escenario 1 *Máquina JAVA / Store JAVA*	**Escenario 2** *Máquina JAVA / Store C*	**Escenario 3** *Máquina C / Store C*
$TELL\ X_3 = 4$	199	83	0.0229

Tabla 5.4: Tiempos de Ejecución en milisegundos del Programa *NREINAS*

Los resultados para el **Escenario 3** son los siguientes (ver Tabla 5.5):

$$\begin{array}{lll} X_1 & in & \{3,3\} \\ X_2 & in & \{1,1\} \\ X_3 & in & \{4,4\} \\ X_4 & in & \{2,2\} \end{array}$$

Tabla 5.5: Resultados del Programa *NREINAS* en el **Escenario 3**

Una vez mas, como se puede notar en la figura 5.3 (representación de los datos de la Tabla 5.4), los tiempos mínimos de ejecución fueron registrados por el **Escenario 3**, en el que interviene de forma significativa la reimplementación de *MAPiCO en el lenguaje C*.

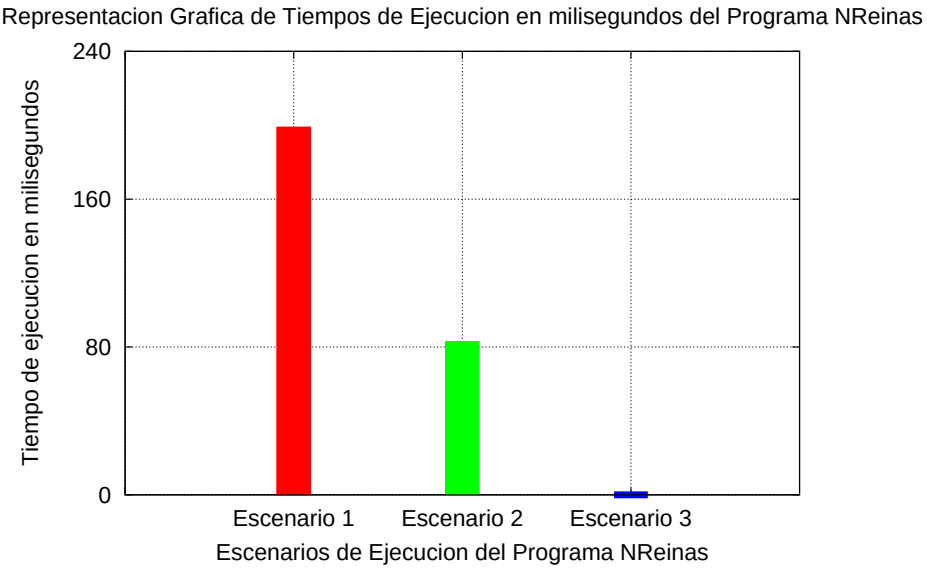

Figura 5.3: Representación Gráfica de Tiempos de Ejecución en milisegundos del Programa *NREINAS*

Adicionalmente, se realizaron comparaciones de los tiempos de ejecución de los programas *Factorial* y *Fibonacci* en diferentes entornos como *JAVA*, *.NET* y *GECODE*, apreciando tiempos competitivos de *MAPiCO* frente a estos entornos. Los correspondientes tiempos de ejecución en milisegundos se presentan en las Tablas 5.6 y 5.7.

Con las anteriores pruebas se demuestra que la reimplementación de la Máquina Abstracta

Programa Factorial	Entornos de Ejecución			
	Java	.Net	GECODE	MAPiCO
0	0	0	0.15	0.0009
1	0	0	0.15	0,0031
2	0	0	0.15	0,0051
3	0.15	0	0.15	0.0081
4	0.15	0,15625	0.15	0.0111
5	0.15	0,15625	0.15	0.0141
6	0.15	0,15625	0.15	0.0201
7	0.16	0,15625	0.15	0.0231

Tabla 5.6: Comparación de Tiempos de Ejecución del Programa Factorial en Diferentes Entornos

Programa Fibonacci	Entornos de Ejecución		
	Java	.Net	MAPiCO
0	0	0	0,001
1	0	0	0,001
2	0	0	0,001
3	0.15	0	0,003
4	0.15	0,15625	0,005
5	0.15	0,15625	0.011
6	0.15	0,15625	0.022
7	0.16	0,15625	0.055
8	0.16	0,15625	0,133
9	0.16	0,15625	0,329
10	0.16	0,15625	0,851

Tabla 5.7: Comparación de Tiempos de Ejecución del Programa Fibonacci en Diferentes Entornos

MAPiCO en el lenguaje C, representa una disminución significativa en los tiempos de ejecución de los diferentes escenarios y entornos, lo que determina una mejora en la eficiencia de la etapa de ejecución del Lenguaje *Cordial*.

6 CONCLUSIONES

En el presente trabajo se reimplementó la máquina abstracta *MAPiCO*, máquina soportada por el cálculo lógico, correcto y formal *PiCO*. La reimplementación de *MAPiCO* surge de la necesidad de mejorar la eficiencia, contemplar las modificaciones del cálculo y extender de manera dinámica el conjunto de instrucciones de la máquina.

Aunque la implementación inicial cumple con los propósitos y requisitos para los cuales fue creada, el lenguaje en el que fue implementada, *JAVA*, hace que se pierda eficiencia en el tiempo de ejecución de la máquina, ya que por ser un lenguaje interpretado es mas lento que un lenguaje compilado, lo cual se evidencia en las pruebas realizadas.

La actual implementación se realizó en el lenguaje de programación *C*, considera directrices de precompilación y modularidad en el desarrollo, lo que garantiza tiempos de ejecución mucho mas eficientes, funcionalidad modular y practicidad en la modificaciones que deban realizarse en un futuro a la máquina.

Debido a que la reimplementación de *MAPiCO* soporta la extensibilidad en cuanto al conjunto de instrucciones, se hace posible implementar cualquier modificación al cálculo *PiCO*, de forma modular y sin reimplementar de nuevo la Máquina Abstracta.

Otra de las características de la actual implementación es que gracias a su extensibilidad, flexibilidad y modularidad puede ejecutarse como una máquina de máquinas. En este caso, se podría implementar la funcionalidad de otra máquina mediante los *PlugIn's* y ejecutarlos en *MAPiCO*; la Máquina Abstracta *LMAN* [AM04] podría implementarse bajo este criterio.

La alternativa de Eliminación de Variables planteada en el capitulo **3** (sección 3.4)

podría significar una mejora considerable del *Store*, con respecto a la propagación de variables.

Se realizaron pruebas que determinaron la eficiencia de la reimplementación de la máquina abstracta *MAPiCO*:

- Pruebas en las que se verificó las complejidades de cada una de las funciones utilizadas para las estructuras de listas, árboles, algoritmos de búsqueda, de almacenamiento de memoria, de precompliación y modularidad. Cada una de las complejidades de estas funciones se encuentran en en el **capitulo 4** y en los archivos de cabecera de la implementación.

- Pruebas de carga de archivos de programa de gran tamaño en la memoria estática y de *PlugIn's* en tiempo de ejecución; que confirmaron la capacidad de almacenamiento y eficiencia en modularidad/extensibilidad de la reimplementación de *MAPiCO* (capitulo 4).

- Pruebas funcionales que determinaron la correcta reimplementación de la Máquina Abstracta *MAPiCO*.

- Las pruebas realizadas en el **capitulo 5** demostraron que:

  - El **Escenario 2**, determinado por *MAPICO JAVA* y *Store C* es un 66 % más eficiente que el **Escenario 1** constituido por *MAPICO JAVA* y *Store JAVA*.
  - El **Escenario 3**, determinado por *MAPICO C* y *Store C* es un 99 % más eficiente que el **Escenario 1** constituido por *MAPICO JAVA* y *Store JAVA*.
  - El **Escenario 3**, determinado por *MAPICO C* y *Store C* es un 99 % más eficiente que el **Escenario 2** constituido por *MAPICO JAVA* y *Store C*.

  Con respecto a las anteriores pruebas se puede concluir que *MAPiCO C* cumple con los propósitos para los cuales fue reimplementada: es eficiente ya que presenta los mínimos tiempos de ejecución, es extensible y modular en cuanto al conjunto de instrucciones y brinda una alternativa de eliminación de variables que no se necesitan mas en la ejecución.

7 RECOMENDACIONES

- Implementar la alternativa de Eliminación de Variables en el Sistema de Restricciones (ver sección 3.4). En esta alternativa se debe considerar que la máquina guarda la referencia del proceso que la creo y que esa referencia se puede utilizar para su reconocimiento, propagación y posterior eliminación. En caso de implementar la Eliminación de Variables, se debería considerar reimplementar o modificar el *Explorador de Cordial* [MZ01], para mejorar la eficiencia de éste al momento de la reducción del dominio de las variables en la búsqueda de la solución.
- Cambiar el Sistema de Restricciones a uno mas genérico como *GECODE - Generic Constraint Development Enviroment* [Sch99], ambiente abierto, libre, portable y accesible para desarollar sistemas de restricciones y aplicaciones.
- Realizar pruebas funcionales de programas ejecutados desde *Cordial* y no solo desde *MAPiCO*, para comparar tiempos y resultados.
- Reutilizar la Máquina Abstracta *MAPiCO* para implementar otros cálculos como π o $NTCC$.
- Implementar un Explorador de Restriciones similar al trabajo de grado *Explorador de Cordial* [MZ01].
- Implementar el Compilador *PiCO - MAPiCO* tratando de optimizar la generación de variables.
- Implementar un Garbage Collector dentro de *MAPiCO* de las variables que no se requieren mas durante la ejecución, tomando en consideración la altermativa de Eliminación de Variables (sección 3.4).
- Implementar un cache de objetos que permita minimizar los tiempos de búsqueda de los objetos más utilizados.

- Implementar el soporte de hilos en la Máquina Abstracta *MAPiCO*, con el fin de que la ejecución de *MAPiCO* se realice en un hilo diferente al de la eliminación de variables por parte del *Store*.

Bibliografía

[AM04] Hurtado A.R. and Muñoz M.P. LMAN: Máquina Abstracta del cálculo NTCC para programación concurrente de robots LEGO, 2004. Pontificia Universidad Javeriana - Seccional Cali.

[ASS99] A.J. Garcia A.G. Stankevicius and J.R. Simari. Una Arquitectura para la ejecución de Programas Lógicos Rebatibles, 1999.

[Ber01] Felipe Bergo. autotut: Using GNU autoconf, make, header, 2001.

[BH99] Antal Alexander Buss and Mauricio Heredia. MAPiCO: Máquina Abstracta para el Cálculo PiCO, 1999. Pontificia Universidad Javeriana - Seccional Cali.

[Bla01] Jim Blandy. GUILE: Project GNU-s extension language, 2001.

[DJ04] Fernandez D. and Quintero J. VIN: Lenguaje Visual basado en el cálculo NTCC, 2004. Pontificia Universidad Javeriana - Seccional Cali.

[DR01] Juan Francisco Diaz and Camilo Rueda. Modelos para la Computación Móvil. *Revista Colombiana de Computación*, 1:29–45, September 2001.

[FJ03] Rocha F. and Chalá J. Compilador NTCC-LMAN. Reporte del curso de compiladores, 2003. Pontificia Universidad Javeriana - Seccional Cali.

[Fon01] María Ángeles Díaz Fondón. Núcleo de Seguridad para un Sistema Operativo Orientado a Objetos soportados por una Máquina Abstracta, 2001. Tesis Doctoral, Universidad de Oviedo.

[Gee03] Geekos. Geekos Operating system for x86, 2003.

[GF98] Gloria Inés Alvarez, Juan Francisco diaz, Luis Omar Quesada and Frank D. Valencia. PiCO: A Calculus of Concurrent Constraint Objects for Musical Applications. Technical report, Pontificia Universidad Javeriana - Seccional Cali, 1998. AVISPA Research Team.

[GG99] Gloria Alvarez, Juan Francisco Diaz, Camilo Rueda, Luis Omar Quesada, Frank Valencia and GeRard Assayag. Integrating Constraint and Concurrent Objects in Musical Applications: A Calculus and its Visual Language. Programming. Technical report, Pontificia Universidad Javeriana - Seccional Cali, 1999. Submitted to Constraints, Kluwer Academic Publishers.

[GIAD97] Camilo Rueda Gloria Inés Alvarez and Juan Francisco Diaz. Tutorial del Cálculo TyCO. Technical report, Pontificia Universidad Javeriana, Seccional Cali, October 1997.

[GV01] Catherine García and Alejandra Maria Vasquez. Implementación eficiente de un Sistema de Restricciones para el Lenguaje Cordial, 2001. Pontificia Universidad Javeriana - Seccional Cali.

[JF04] Jose Emilio Labra Gayo, Juan Manuel Cueva Lovelle, Raúl Izquierdo Castanedo, Aquilino Adolfo Juan Fuente, Maria Candida Luengo Diez and Francisco Ortín Soler. Intérpretes y Diseño de Lenguajes de Programación. Technical report, Universidad de Oviedo, 2004.

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

[LG97] Luis Omar Quesada, Camilo Rueda and Gabriel Tamura. The Visual Model of Cordial. Technical report, Pontificia Universidad Javeriana - Seccional Cali, 1997. AVISPA Research Team.

[LL97] Vasco T. Vascocelos Luis Lopes. TyCO Abstract Machine - The Definition -. Technical report, Departamento de Ciencias de Computadores - Facultad de Ciencias, Universidade du Porto, 1997.

[LV97] Luis Lopes and Vasco T. Vascconcelos. An Abstract Machine for Object Calculus. Technical report, Departamento de Ciencias de Computadores - Facultad de Ciencias, Universidade du Porto, 1997.

[Mai00] Pier R. Mai. Abstract Machine for Oz and Curry, 2000.

[Mar01] Lourdes Tajes Martinez. Sistema de Computación para un Sistema Operativo Orientado a Objetos basado en una Máquina Abstracta Reflectiva Orientada a Objetos, 2001. Tesis Doctoral, Universidad de Oviedo.

[MC95] Michael Melh, Ralf Scheidhauer and Christian Schulte. An Abstract Machine for Oz, 1995.

[Mil80] Robin Milner. A Calculus of Communicating Systems. Lecture Notes in Computer Science, LNCS 92, 1980.

[Mil92a] Robin Milner. Communicating and Mobile Systems: The ϕ *Calculus*. *Journal of Information and Computation*, 100:1–77, September 1992.

[Mil92b] Robin Milner. Functions as Processes. *Journal of Mathematical Structures in Computer Science*, 2(2):119–141, 1992.

[Moz06] The Mozart Programmig System, 2006.

[MZ01] Elbert A. Messa and Ana Maria Zuñiga. Explorador de Cordial, 2001. Pontificia Universidad Javeriana - Seccional Cali.

[Pop97] Konstantin Popov. A Parallel Abstract Machine for the Thread-Based Concurrent Language Oz, 1997. PS Lab, University of Saarland.

[PT94] Benjamin C. Pierce and David Turner. Concurrent Objects in a Process Calculus. In *Theory and Practice of Parallel Programming (TPPP), Sendai, Japan*, November 1994.

[RD92] 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.

[Rue94] Camilo Rueda. A Visual Programming Enviroment for Constraint-Based Musical Composition. XIV Congresso da Sociedade Brasileira de Computacao. Caxambu - Brasil, 1994.

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

[Sch99] C. Schulte. GECODE. Generic Constraint Development Environment, 1999.

[Sch03] Eric Schonoebelen. Dynamic life of Plug-Ins. *Linux Magazine*, 1:29–45, February 2003.

[Shi01] Herbert Shildt. C Manual de Referencia, January 2001. Osborne McGraw-Hill.

[Vas94] Vasco T. Vasconcelos. Typed Concurrent Objects, 1994. Phd thesis, Keio University.

[vH97] Dimitri van Heesch. DOXYGEN. Documentation system for C, C++, Java, Objective-C, Python, IDL, PHP, C# and D, 1997.

Anexo A. Uso y Guía de Implantación de la Máquina Abstracta *MAPiCO*

Anexo A. Uso y Guía de Implantación de la Máquina Abstracta *MAPiCO*

MAPiCO ha sido desarrollado utilizando los estándares establecidos por **GNU** para el ordenamiento de proyectos, herramientas como *automake* y *autoconf* ayudan a la portabilidad de la máquina a nivel de código fuente, abstrayéndose en la medida de lo posible, de las versiones de las herramientas tradicionales disponibles en cada sistema operativo.

MAPiCO se encuentra empaquetada y cuenta con un script llamado $configure$, el cual al ejecutarse realiza todas las verificaciones y definiciones necesarias para que la máquina se pueda compilar y posteriormente instalar con éxito.

En la figura D.1 se muestra el árbol con el contenido de los directorios y archivos de la máquina:

El directorio raiz contiene los archivos que permiten mantener control sobre el proyecto.

- Carpeta **asmapico**: Contiene los fuentes del Assembly *MAPiCO* que permite generar el código en bytes de entrada de la máquina.
- Carpeta **constsys**: Incluye el código fuente del sistema de restricciones utilizado.
- Carpeta **doc**: En esta carpeta se ubican los archivos de configuración de la máquina.
- Carpeta **include**: Contiene las cabeceras de las librerías que requiere *MAPiCO* (PlugIns.h, Parameters.h, Loader.h, etc.).
- Carpeta **mapico**: Almacena el código fuente de la máquina y de todas las librerías implementadas (MAPiCO.c, PlugIns.c, BinaryTree.c, etc.).
- Carpeta **PlugIn's**: Incluye el código de los $PlugIn's$ desarrollados.

Los archivos de información obligatorios son:

- *NEWS*: Registra los cambios mas recientes.

- *README*: Contiene la descripción del proyecto.
- *MAPiCO*: Incluye algunas instrucciones especiales.
- *AUTHORS*: Contiene los nombres de los autores de la máquina.
- *CHANGELOG*: Contiene un registro con todos los cambios que se han efectuado.
- *COPYING*: Aquí se especifica el tipo de licencia sobre el cual se definió la máquina.
- *INSTALL*: Proporciona todas las instrucciones de instalación de *MAPiCO*.

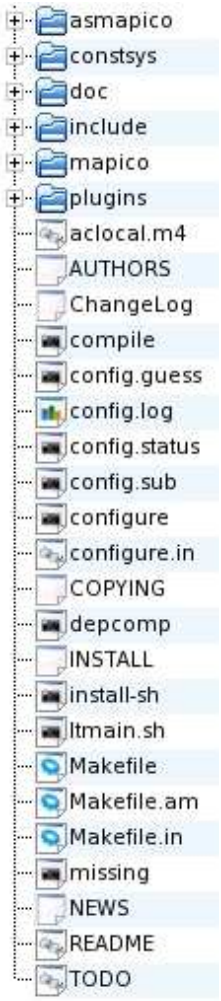

Figura D.1: Estructura de Directorios *MAPiCO*

La ejecución de la nueva *MAPiCO*, se realizó a través de línea de comandos y requiere como argumento el archivo de entrada o *bytecodes* en el momento de la invocación. La implementación se realizó en *lenguaje C*, la forma de invocación es la siguiente:

./mapico program.mapico

En donde program.mapico denota un archivo binario con el programa a ejecutar. Este programa tiene la estructura presentada en la sección 3.2.1.5.

Para la reimplementación de la máquina abstracta *MAPiCO*, se tuvieron en cuenta las siguientes utilidades:

- Utilidad *make* de *GNU*.
- El preprocesador *cpp* de *GNU* (*The GNU C−Compatible Compiler Preprocessor*).
- El compilador *gcc*.
- La herramienta *doxygen* para generar la documentación de la implementación en formato HTML.
- Las herramientas *Autotools* de *GNU* que generan los scripts de configuración.

El código fuente de *MAPiCO* se encuentra en: $http://gforge.puj.edu.co/projects/mapico2$, desde esa dirección se puede descargar la última versión de la Máquina Abstracta *MAPiCO*.

Para compilar la máquina abstracta *MAPiCO* se debe invocar las siguientes instrucciones:

```
tar zxvf mapico2-X.Y.Z.tar.gz
./configure
make
make install
```

donde *X.Y.Z* es la versión de *MAPiCO* (al momento de la elaboración de este documento se encuenra en la versión 0.0.1).

La primera línea descarga los archivos fuente de la Máquina Abstracta, la segunda línea ejecuta el script llamado *configure*, que comprueba si el sistema dispone de todos los recursos necesarios para compilar *MAPiCO*: (librerías, compilador, etc.) y genera el archivo *Makefile*.

Para obtener parámetros adicionales ejecutar:

```
./configure --help
```

La invocación de *make* busca el *Makefile* creado por *configure* y compila el código fuente de *MAPiCO* de acuerdo a sus dependencias, generando las librerias dinámicas de la máquina y el ejecutable de la misma.

La tercera línea instala *MAPiCO* en el sistema operativo para que pueda ser utilizado desde cualquier parte.

Anexo B. Guía de Adición de PlugIn's - Extensibilidad

Anexo B. Guía de Adición de PlugIn's - Extensibilidad

Los *PlugIn's* en *MAPiCO* son archivos en *C* los cuales no necesitan archivo de cabecera *.h*, la estructura de un *PlugIn* es la siguiente:

Librerías de cabecera
Función Catalogo
Conjunto de Instrucciones

Las librerías de cabecera estándar de un *PlugIn MAPiCO* son: *Proceso.h*, *Loader.h* y *PlugIns.h*. *PlugIns.h* es de uso obligatorio porque es la que ofrece las funciones para el manejo y/o control de los *PlugIn's* y las instrucciones, en estas se pueden incluir las librerías que se requieran para la ejecución de una instrucción.

```
#include "Proceso.h"
#include "Loader.h"
#include "PlugIns.h"
#include "StoreMA.h"
```

En cada *PlugIn MAPiCO* debe definirse la función catalogo (*Catalogue*) que ofrece la información de instrucciones del *PlugIn*. A continuación se presenta un ejemplo:

```
1.Parameters Catalogue(Parameters argv,char *PathLibrary,void *Handle,int NumPlugIn)
2.{
3.  char *Functions[]={"13:NEWV","14:PAR","15:FINISH","16:FUNC",NULL};
4.
5.  //Cargar las instrucciones del PlugIn
6.  argv = loadInstructions(argv, Functions, PathLibrary, Handle, NumPlugIn);
7.
8.  return argv;
9.}
```

En donde:

Línea 1: *Catalogue* recibe como argumentos *argv* que corresponde a la lista de parámetros creada por la máquina en el bloque principal, *PathLibrary* que contiene la ruta física donde se encuentra el *PlugIn*; *Handle* es el apuntador al *PlugIn* y *NumPlugIn* que corresponde a un consecutivo que identifica la librería dinámica.

Línea 3: La función *Catalogue* define un arreglo de cadenas que describe la información de las instrucciones que contiene el *PlugIn*, las cadenas son de la forma *OpCode* :

NameInstruction donde *OpCode* es el código de la instrucción y *NameInstruction* como lo indica es el nombre de la instrucción que viene en el *PlugIn*, se pueden definir el número de instrucciones que se desee, se le indica a la máquina que el *PlugIn* define *NEWV*, *PAR*, *FINISH*, *FUNC*, *CALL*, *DEL* y *COMM*, las cuales deben estar implementadas de lo contrario la máquina reportara un error, indicando que la instrucción no ha sido encontrada.

La carga de las instrucciones se realiza mediante la función *loadInstructions* (línea 6) definida en la librería *PlungIns.h*, la cual llena el árbol de instrucciones y lo guarda en la lista de parámetros, retornando la lista a la ejecución de la máquina (línea 8).

Una instrucción recibe como argumento la lista de parámetros de la máquina y la retorna con las modificaciones que resultaron de la ejecución de la instrucción, las instrucciones son de la forma:

```
Parameters NameInstruction(Parameters argv)
{
   <Obtencion de parametros>
   <Acciones>
   <Retorno de Parametros>
}
```

Ejemplo:

```
1. Parameters PAR(Parameters argv)
2. {
3.    //Registro que se necesita de MAPiCO
4.    int PCA;              //Puntero a Codigo Actual
5.    NodeList PVA;         //Puntero a Variables Actual
6.    NodeList PMA;         //Puntero a Metodos Actual
7.    Auxiliar PAA;         //Puntero a Argumentos Actual
8.    ObjectEval POA;       //Puntero a Objetos Actual
9.    Auxiliar PAUX;        //Puntero Auxiliar
10.   void *PTEMP;          //Puntero Temporal
11.
12.   //Declaracion de colas
13.   List RunQ;            //Cola de Ejecucion
14.
15.   //Declaracion de la direccion de PAR
16.   int DirPar = 0;
17.
18.   //Vble para definir procesos
19.   Proceso Proc;
20.   Proceso ProcPar;
21.
```

```
22.  //Vble para el Program Byte
23.  Program Data;
24.
25.  //Recuperacion del primer parametro (Proceso)
26.  Proc = getAtVblePlugIn(argv,1);
27.
28.  //Recuperacion del tercer parametro (Data)
29.  Data = getAtVblePlugIn(argv,3);
30.
31.  //Recuperacion de la cola de ejecucion (RunQ)
32.  RunQ = getAtVblePlugIn(argv,4);
33.
34.  //Se recuperan los valores de los registros
35.  PCA = getPC(Proc);
36.  PVA = getPV(Proc);
37.  PMA = getPM(Proc);
38.  PAA = getPA(Proc);
39.  POA = getPO(Proc);
40.  PAUX = getPAUX(Proc);
41.
42.  DirPar = get32(Data, PCA+1);
43.
44.  //Insercion de un nuevo proceso en RunQ
45.  ProcPar =  newProceso(DirPar, PVA, PMA, PAA, POA, PAUX, NULL);
46.  INSERT_AT_END(RunQ, ProcPar);
47.
48.  //Incrementar el PCA para la siguiente instruccion
49.  PCA = PCA + INST_BYTE + DIR_BYTE;
50.
51.  //Asignacion de los nuevos valores
52.  setPC(Proc, PCA);
53.
54.  //Se guardan las variables en la lista de vbles de los PlugIn's
55.  argv = setAtVblePlugIn(argv,Proc,1);
56.  argv = setAtVblePlugIn(argv,RunQ,4);
57.
58.  //Retorno de los argumentos
59.  return argv;
60.}
```

En esta instrucción en el rango de líneas 3 a 23 se definen las variables que necesitará la instrucción, la recuperación de variables de la lista de parámetros se realiza entre las líneas 26 a 40, la acción de la instrucción se realiza entre las líneas 42 a 49 y por último la asignación y retorno de variables a la lista de parámetros se considera entre las líneas 52 y 59.

Anexo C. Guía de Cambios con Respecto a la Interfaz con el Sistema de Restricciones

Anexo C. Guía de Cambios con Respecto a la Interfaz con el Sistema de Restricciones

MAPiCO interactúa con el *Store* a través de la interfaz *StoreMA.h*, Ésta reimplementación utilizó el Sistema de Restricciones Aritmético de Dominios Finitos hecho en *C* [GV01]. Para cambiar el Sistema de Restricciones solamente se debe modificar la interfaz implementando las mismas funciones que se ofrecen actualmente, esto con el objetivo de no modificar ninguna de las instrucciones.

Los cambios que deben realizarse para cambiar el Sistema de Restricciones son:

1. En *StoreMA.h*

   - Incluir las librerías del nuevo Sistema de Restricciones Aritmético
     Ejemplo:

     ```
     #include "FrameConst.h"
     #include "CS.h"
     ...
     ```

   - Definir las constantes propias de *MAPiCO* (que tienen el sufijo _MA) con base en las del nuevo Sistema Aritmético.
     Ejemplo:

     ```
     //Declaracion de Constantes y Macros publicos
     #define SUCCESS_MA SUCCESS
     #define FAIL_MA    FAIL
     #define ASK_MA     ASK
     #define TELL_MA    TELL
     ...
     ```

   - Modificar la estructura *str_StoreMA* colocando el nuevo Sistema de Restricciones y las variables de inicialización que necesite (si es que las hay).
   - Definir los tipos propios de *MAPiCO* (que tienen el sufijo _MA) con base en los del nuevo sistema aritmetico.
     Ejemplo:

     ```
     typedef FrameConst FrameConstMA;
     ...
     ```

- Agregar las funciones adicionales que se necesiten para interactuar con el nuevo Sistema de Restricciones.

2. En *StoreMA.c*

   - En la función de inicialización del *store*, reservar memoria a la estructura e inicializar sus parámetros asignando los mensajes de error correspondientes si los hay.
     Ejemplo:

```
StoreMA inicStoreMA(void)
{
  StoreMA newStore;
  //Asignacion de memoria para la estructura
  newStore = (StoreMA)malloc(sizeof(struct str_StoreMA));
  //Asignacion de memoria para las variables del store
  newStore->i = (Indexicals *)malloc(sizeof(Indexicals));
  newStore->indextemp = (int *)malloc(sizeof(int));
  newStore->t=(Stamp *)malloc(sizeof(Stamp));
  //Inicializacion de variables
  newStore->i = InicIndexicals();;
  newStore->s = inicStore();

  if(!newStore->indextemp)
  {
    newStore->Error = (char *)malloc(sizeof(char)*100);
    strcpy(newStore->Error,"Fallo la asignacion de memoria (Index_Temp)");
    newStore->Response = StoreERR;
    return newStore;
  }
  *(newStore->indextemp)=-1;

  if(!newStore->t)
  {
    newStore->Error = (char *)malloc(sizeof(char)*100);
    strcpy(newStore->Error, "Fallo la asignacion de memoria (Stamp)");
    newStore->Response = StoreERR;
    return newStore;
  }
  *(newStore->t)=0;

  //Inicializacion del Store Correcta
  newStore->Response = StoreOK;
  newStore->inicializado = StoreOK;
  return newStore;
}
%
```

- Para cada función realizar la llamada correspondiente en el nuevo Sistema de Restricciones, pasandole los parámetros necesarios.
  Ejemplo:

```
StoreMA tellMA(StoreMA store, FrameConstMA frameconst)
{
  tell(store->s, store->i, store->t, store->indextemp, frameconst);
  return store;
}
...
```

Anexo D. Assembly MAPiCO

Anexo D. Assembly MAPiCO

El *Assembly MAPiCO* es una programa que se encarga de traducir, un archivo fuente en formato texto a un archivo de salida en formato binario. Este archivo de salida contiene el código que *MAPiCO* ejecuta (*bytecode*). Este *Assembly* surgió de la necesidad de tener una herramienta que generara el archivo binario de entrada de *MAPiCO*, debido a que el compilador *PiCO - MAPiCO* que debería proporcionarlo, se encuentra actualmente en desarrollo (ver figura D.2); igualmente era necesario desarrollar un conjunto de pruebas de la máquina con base en los archivos de entrada. La realización de este archivo binario se realiza en forma manual.

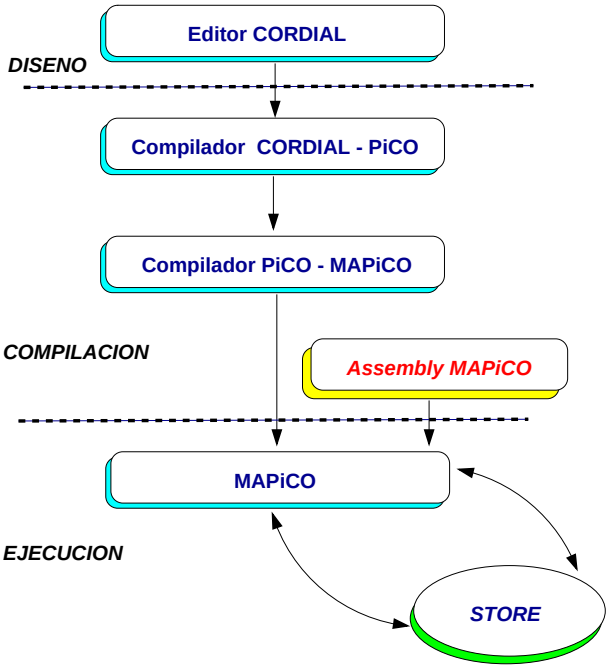

Figura D.2: Componentes del Lenguaje *Cordial*

Esta herramienta no reemplaza el compilador debido a que el *Assembly MAPiCO* es un tipo de ensamblador básico, de bajo nivel, y su tarea consiste en reconocer las directivas de *MAPiCO* y transformarlas a su equivalente código de operación u *OpCode* en formato binario; además el *Assembly* resuelve las direcciones de las instrucciones mediante un parámetro que hace referencia a una posición de memoria en el archivo texto.

El archivo fuente que el *Assembly* convierte a binario, es una transformación equivalente de código *PiCO* a *MAPiCO* (en modo texto), con la variante de que las instrucciones son identificadas por su nombre y las direcciones corresponden al número de línea en el archivo texto donde se encuentran las instrucciones que se quieren referenciar. Realizar manualmente este archivo de entrada es relativamente fácil.

El siguiente es un fragmento de un archivo de entrada del *Assembly MAPiCO*:

```
1.  INIT
2.  NEWV
3.  NEWV
4.  PAR 52
5.  FUNC 17
6.  POP
7.  ATOM MAY 2
8.  TERMV 0
9.  TERMC 0
10. ASKF
11. PUSHV 0
12. CALL
13. POP
14. COMM
15. PUSHM 0
16. REE
17. FUNC 25
18. NEWV

...
```

En la línea 5, el parámetro de la instrucción *FUNC*, hace referencia a una dirección que corresponde al número de la línea donde se encuentra la próxima instrucción a ejecutar. En el anterior ejemplo *FUNC* esta apuntando a la línea 17 donde se encuentra otra instrucción *FUNC*.

El *Assembly* es un programa hecho en *C*, que utiliza librerías XML para su procesamiento y recibe dos parámetros de entrada: un archivo de configuración XML y un archivo fuente en formato texto.

El archivo de configuración XML contiene la descripción de las instrucciones y los códigos de los predicados de las restricciones.

Para describir una instrucción se debe crear un tag XML que debe contener el nombre

de la instrucción y los atributos *OpCode* y *NumParams*; el *OpCode* es el equivalente numérico del nombre de la instrucción y *NumParams* es el número de argumentos que tiene la instrucción. Con respecto a lo anterior, si la instrucción no tiene parámetros se le asigna a *NumParams* el valor cero (0).

```
<RET OpCode="12" NumParams="0"/>
<NEWV OpCode="13" NumParams="0"/>
<POP OpCode="7" NumParams="0"/>
<ASK OpCode="24" NumParams="0"/>
<TELL OpCode="23" NumParams="0"/>
<CALL OpCode="17" NumParams="0"/>
```

Si la instrucción tiene parámetros, dentro del tag se crea otro tag con el nombre **Params** y en su interior se colocan tantos parámetros como lo indique el atributo *NumParams*.

```
<COBJCC OpCode="11" NumParams="3">
    <Params>
        <Param1 Tipo="D" TamBytes="32"/>
        <Param2 Tipo="D" TamBytes="32"/>
        <Param3 Tipo="C" TamBytes="8"/>
    </Params>
</COBJCC>

<ATOM OpCode="22" NumParams="2">
    <Params>
        <Param1 Tipo="P" TamBytes="16"/>
        <Param2 Tipo="C" TamBytes="16"/>
    </Params>
</ATOM>
```

Cada parámetro debe tener dos atributos que definen su comportamiento: el **Tipo** y el **TamBytes**.

- **Tipo**: puede ser de Dirección (**D**) que hace referencia a una dirección en el archivo o Predicado (**P**) que indica que es un predicado de una restricción ó Constante (**C**) para valores estáticos.
- **TamBytes**: Indica es el tamaño en bytes del parámetro.

Los predicados basicamente contienen su *OpCode* equivalente.

Un ejemplo de un archivo de configuración en XML es:

```
<?xml version="1.0"?>
<Configuration>
   <Instructions>
      <RET OpCode="12" NumParams="0"/>

      <METH OpCode="6" NumParams="2">
         <Params>
            <Param1 Tipo="C" TamBytes="8"/>
            <Param2 Tipo="D" TamBytes="32"/>
         </Params>
      </METH>

      <TERMF OpCode="26" NumParams="2">
         <Params>
            <Param1 Tipo="P" TamBytes="16"/>
            <Param2 Tipo="C" TamBytes="16"/>
         </Params>
      </TERMF>

      ...

   </Instructions>

   <Predicates>
      <IGUAL Code="28"/>
      <EQ Code="28"/>
      <DIF Code="1"/>

      ...

   </Predicates>

</Configuration>
```