Universidad Nacional Autónoma de México

U.A.C.P. y P. del C.C.H.

UNA MAQUINA VIRTUAL PARA TM

T E S I S

Que para obtener el grado de

MAESTRO EN CIENCIAS DE LA COMPUTACION

p r e s e n t a

SERGIO RAMON CARDENAS GARCIA

Director:
Dr. J. Miguel Gerzso Cady

México, D. F. 1986

UNAM – Dirección General de Bibliotecas Tesis Digitales Restricciones de uso

CONTENIDO

INTRODUCCION

El tema de esta tesis comprende el diseño e implantación eficiente de una parte del sistema TM llamado máquina virtual. Forma parte de el proyecto TM del Departamento de Sistemas de Cómputo , del I.I.M.A.S. Dicho proyecto a cargo de Miguel Gerzso, creador de TM, está enfocado al desarrollo e implantación de un lenguaje de programación de computadoras digitales para propósito general llamado TM. Este lenguaje se cataloga dentro de los lenguajes orientados a manejo de objetos con mecanismos de envío de mensajes y herencia.

Este tipo de lenguajes representa un avance en la solución de problemas en el campo del diseño asistido por computadora ya que permite definir, manejar y modificar con facilidad los objetos complejos que frecuentemente se presentan en los proyectos de ingeniería, arquitectura, etc.

Las máquinas virtuales son programas (software) que permiten ver a la computadora real donde residen (hardware) como otra computadora con características diferentes. Esta transformación permite definir otro conjunto de instrucciones, diferente forma de obtener los datos de memoria, etc. que sean más adecuados para alguna aplicación particular. Son llamadas "virtuales" ya que en realidad son simulaciones de máquinas y no existen físicamente. Se puede imaginar una máquina virtual como una envoltura de programación que cubre a la máquina real de manera que un usuario sólo ve la parte virtual. Ver fig. 1.1.

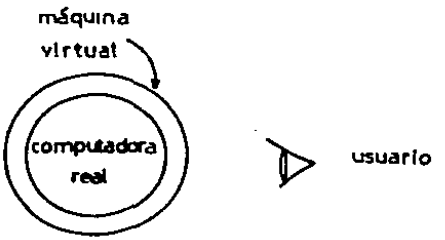

Fig. 1.1 Máquina Virtual

La existencia de máquinas virtuales se basa en el

principio de existencia de "programas universales" o "simuladores". Este principio es una derivación de la tésis de Church-Turing y establece que siempre es posible simular el comportamiento de cualquier computadora usando cualquier otra computadora (aunque se tendrán diferencias en eficiencia y empleo de recursos).

Se ha implantado el sistema TM como máquina virtual con el fin de evitar dependencias innecesarias con el hardware, de manera que el compilador sea transportable y se pueda tener un conjunto de instrucciones más adecuado para el manejo de objetos que el que podría proporcionar una computadora convencional. En el capítulo siguiente se explican más detalladamente las razones que determinaron la construcción de la máquina virtual para TM.

i) Lenguajes Orientados a Objetos

Parece ser que el termino "lenguaje orientado a objetos" se empleó por primera vez para describir al sistema de programación diseñado en Xerox PARC llamado Smalltalk [Krasner 84]. Muchas de las ideas de este sistema provienen del lenguaje Simula [Dahl 66]. Desde entonces han surgido varios sistemas, entre ellos TM [Gerzso 83], que por sus características comunes se catalogan dentro de esta categoría.

Las características que presentan los lenguajes orientados a objetos son (según [Stefik 86] y [Pascoe 86]):

- Uso de Objetos como entidades que combinan propiedades de procedimientos y datos; existen objetos que contienen códigos o instrucciones ejecutables y objetos que almacenan datos.

- Soporte a la Abstracción de Datos. Se tienen mecanismos para encapsular juntas todas las declaraciones de los procedimientos que actúan sobre un tipo de datos. Otras rutinas no pueden trabajar directamente sobre los objetos de ese tipo. En TM la declaración de un administrador permite definir un tipo de datos con las rutinas que actúan sobre el mismo. El soporte a la abstracción de datos permite asegurar confiabilidad, seguridad y modificabilidad de los programas al reducir interdependencia entre componentes y permitir el "ocultamiento de la información".

- Mecanismos de Herencia (o Especialización). Permiten definir nuevos tipos de datos cuyos objetos son casi iguales a otros objetos (de algún tipo de datos ya definido). La herencia reduce la necesidad de especificar información redundante y simplifica las actualizaciones y modificaciones, ya que la información puede insertarse y cambiarse en un solo lugar.

- Envío de Mensajes. Esta característica no se presenta en la totalidad de los lenguajes orientados a objetos. El envío de

mensajes es una forma de llamada a un procedimiento. En lugar de invocar un procedimiento para que efectúe alguna operación sobre algún objeto, se envía un mensaje al objeto. En el envío del mensaje aparece un "selector" que especifica la operación a realizar y la lista de parámetros requeridos.

ii) Antecedentes de este Trabajo

Esta tesis no es la primera que está relacionada con el proyecto TM. Además de la labor realizada por Miguel Gerzso, los trabajos que directamente preceden al presente son los que llevaron a cabo Salvador Barra Arias y Fernando Jiménez Fraustro.

El trabajo de Salvador B. constituye el primer esfuerzo para realizar un compilador para TM, desgraciadamente no se ha publicado como tesis y se puede considerar que la documentación es nula. El compilador se construyó en lenguaje Pascal y las únicas fuentes de información sobre el mismo son el propio Salvador B. y los listados del programa fuente.

Fernando J. realizó una nueva versión del compilador, haciendo modificaciones y adiciones al código original de Salvador Barra, y construyó la primera máquina virtual para TM. Dicha máquina fue concebida como un subsistema de prueba del compilador sin fines definitivos. La documentación accesible sobre lo hecho por Fernando J. ha sido muy limitada, no contandose con datos hasta que terminó su tesis, con la que obtuvo su grado el día 8 de agosto de 1986 [Jiménez 86].

Tuve la oportunidad de instalar los programas realizados en lenguaje Pascal por Fernando J. Dicha instalación la llevé a cabo en la computadora VAX 730 del Departamento de Sistemas de Cómputo. La instalación de esos programas me permitió conocer las características generales de la máquina virtual y su interacción con el compilador. En general esa labor fue beneficiosa ya que pude introducirme en los detalles del problema.

iii) Referencias

[Dahl 66] Dahl, O. J. Hygaard, K. "SIMULA - an algol-based simulation language" Communications of the ACM. 9, 1966, pp 671-678.

[Gerzso 83] Gerzso, M. "Report on the Language TM: its design and definition"; IIMAS UNAM 1983.

[Jiménez 86] Jiménez, F. "Diseño e Implantación de un Sistema Orientado a Objetos" Tesis de Maestría, UNAM Agosto 1986.

[Krasner 84] Karsner, G. "Smalltalk-80: Bits of History, Words of

Advice". Reading MA. Addison-Wesley 1984.

[Pascoe 86] Pascoe, G. "Elements of Object-Oriented Programming" Byte, August 1986 pp.139-144

[Stefik 86] Stefik, M. Bobrow D. "Object-Oriented Programming: Themes and Variations" The AI Magazine, March 1986, pp. 40-50.

CAPITULO I

Descripción del Problema

El presente trabajo, enmarcado por el proyecto TM, pretende:

- Contribuir a la difusión de TM.
- Presentar el diseño e implantación eficientes de una parte del sistema TM.
- Explorar los problemas que presenta la instrumentación de un sistema orientado al manejo de objetos en una computadora no orientada al manejo de los mismos.
- Mostrar una manera de llevar a cabo los mecanismos que permite la expresividad de TM.

Una de las metas principales del proyecto TM consiste en obtener implantaciones del sistema que reflejen las características actuales del diseño. Se podría experimentar con la expresividad del lenguaje y emplear las implantaciones como herramientas en el desarrollo de sistemas, ampliando en lo posible el número de usuarios programadores.

La difusión de un sistema como TM es importante por varias razones:

a) Permitiría introducir a los programadores en el enfoque de la programación orientada a objetos.

b) Se podrían obtener más críticas y opiniones sobre las ventajas y desventajas del lenguaje.

c) Como TM es un diseño mexicano, se evitaría caer en un rezago tecnológico en esta área relativamente nueva de la computación.

Se considera que para lograr la difusión del lenguaje es necesario contar con la implantación del mismo.

I.1 Máquina Virtual

El enfoque de manejo de objetos en computación está afectando el diseño de lenguajes, de sistemas operativos y de arquitecturas de computadoras. Dicho enfoque promete influir en gran medida la manera como se producirán programas en el futuro.

Actualmente el uso de computadoras diseñadas expresamente para soportar el manejo de objetos no se ha generalizado (estos sistemas son llamados "capability systems" o sistemas de capacidades). Según Henry Levy [Levy 84] estamos viviendo

apenas la primera generación de sistemas base objetos que ha aparecido en el mercado comercial. En la misma referencia se menciona: "Es el éxito o fracaso de la programación basada en objetos lo que determinará eventualmente el éxito o fracaso de las arquitecturas de capacidades". Mientras tanto, los lenguajes que manejan objetos pueden ser implantados en computadoras convencionales, de manera que al generalizarse su uso, se descubran sus ventajas, y se vea si permiten al programador manejar la complejidad inherente en aplicaciones sofisticadas.

Construir un sistema que soporte el manejo de objetos en una computadora convencional puede considerarse como la simulación de un sistema de capacidades. Es por ello que se necesita diseñar una máquina virtual que proporcione las características arquitectónicas necesarias para el manejo adecuado de los objetos.

Ligada a la necesidad de difusio'n del lenguaje está la conveniencia de que el sistema TM cuente con un alto grado de transportabilidad, con el fin de no encadenarlo a algún modelo de computadora. Por ello se decidió que TM debe contar con un compilador que en lugar de generar código para alguna computadora en particular, genere código ejecutable por una máquina virtual. De esta manera el compilador se puede instalar en cualquier computadora (sin tener que modificarse) en la que resida la máquina virtual.

El presente tema de tesis consiste en el diseño e implantación de la máquina virtual de TM en una computadora VAX-11 (Digital Equipment Corporation). Se pretende obtener la especificación y documentación que permitan construir el compilador que genere código para la máquina virtual, y además sirva como modelo en la construcción de máquinas virtuales de TM que acepten el mismo conjunto de códigos alojadas en otras computadoras.

En particular, para la presente instrumentación del sistema se eligió la computadora VAX ya que es una de las minicomputadoras de mayor aceptación y se cuenta con dos de ellas en el I.I.M.A.S. (Específicamente una VAX 730 en el Departamento de Sistemas de Cómputo).

Actualmente las computadoras diseñadas especialmente para el manejo de objetos (llamadas "capability systems") no han tenido gran difusión, y muchas de sus características arquitectónicas están aún en fase de experimentación. La creación de máquinas virtuales permite la experimentar con nuevas arquitecturas de manera relativamente económica y probar los programas que con el tiempo residirán en una verdadera computadora orientada a objetos. Por otro lado, es posible que proyectos como el presente demuestren que no es necesario contar con un hardware especial para el manejo eficiente de objetos.

Es factible que la máquina virtual diseñada pueda ser

usada para soportar otros lenguajes orientados al manejo de objetos diferentes de TM.

1.2 Rapidez de Ejecución

Otro aspecto del problema a resolver con este trabajo consiste en que los prototipos del sistema TM elaborados hasta el presente no son completos ni eficientes. Esto se debe en gran medida a que fueron diseñados con fines netamente experimentales y se basaron en versiones primitivas del lenguaje.

Se ha dicho que la velocidad lenta de ejecución es una de las características predominantes de los sistemas base objetos. Como ejemplos podemos citar las críticas hechas a los sistemas de Smalltalk-80 y al microprocesador iAPX 432 de Intel que es una arquitectura orientada a objetos. Es por ello que uno de los criterios de diseño que se enfatizó es el que corresponde a la velocidad de ejecución de dicha máquina.

Uno de los aspectos que más contribuyen a la lentitud de ejecución de los sistemas orientados a objetos (sobre todo en el caso de Smalltalk) está ligado al mecanismo de herencia. El problema consiste en la determinación de la rutina adecuada (tambien llamada método o respuesta) que efectuará alguna operación sobre un objeto.

La búsqueda de respuestas a mensajes, que está intimamente relacionado al mecanismo de herencia, tuvo gran influencia en el diseño de esta máquina virtual.

El concepto de herencia en lenguajes orientados a objetos se usa para definir nuevos tipos de objetos que son parecidos a objetos definidos anteriormente. El mecanismo de herencia es importante ya que hace posible la declaración de atributos y funciones que son compartidas por muchos tipos diferentes de objetos, con lo cual se logra construir programas más cortos y organizados.

Para definir un nuevo administrador cuyos objetos son parecidos a otros ya definidos, se declara al administrador de los ya definidos como superclase del administrador a definir. En la definición de un nuevo administrador se puede declarar un número cualquiera de superclases con características que se desea hereden a los objetos del nuevo administrador.

Los objetos del nuevo administrador seran diferentes a los de la superclase por adición y sustitución. Con la adición se introducen nuevas variables y rutinas de respuesta a mensajes que no aparecen definidas en la(s) superclase(s). Con la substitución se puede especificar una variable o una respuesta con el mismo nombre o identificador de otra variable o respuesta que aparezca en la(s) superclase(s). Todo se hereda de la(s) superclase(s) exceptuando aquello que se añade o se substituye.

El envío de un mensaje a algún objeto lleva implícita la realización de la búsqueda en la red de superclases de la rutina de respuesta que contestará al mensaje.

Se ha llegado a considerar que los lenguajes orientados a objetos deben contar con un intérprete que a tiempo de ejecución lleve a cabo la búsqueda de las rutinas que dan servicio a los mensajes. Una de las desiciones que más afectó el diseño de la máquina virtual y que promete aumentar notablemente su velocidad de ejecución consiste en considerar que la mayoría de las búsquedas pueden realizarse exitosamente a tiempo de compilación.

I.3 Referencias

[Levy 84] Levy H.M. "Capability-Based Computer Systems", Digital Press, 1984.

CAPITULO II

Diseño Conceptual del Sistema

II.1 División Modular del Sistema

Un sistema TM consiste escencialmente de dos partes: la llamada imagen virtual o medio ambiente de objetos, compuesta por todos los objetos del sistema, y la máquina virtual que consiste del hardware y las rutinas que manejan los objetos de la imagen virtual. A su vez la máquina virtual puede dividirse en dos subsistemas: el intérprete, que es el encargado de ejecutar el pseudocódigo, las instrucciones que genera el compilador, y el otro susbsistema, el administrador de la memoria. La función principal de este administrador de memoria consiste en proporcionar al intérprete la interfaz necesaria para que éste pueda manejar la imagen virtual. Los objetos que contienen códigos deben ser almacenados en la memoria de objetos para que puedan ser ejecutados por el intérprete.

El compilador produce códigos que son almacenados en archivos. Los códigos deben residir en la memoria de objetos para poder ser ejecutados por el intérprete. Es por ello que se utiliza un módulo extra llamado cargador ligador que toma de los archivos las descripciones de los objetos de código y crea los objetos en la memoria necesarios para contener los códigos.

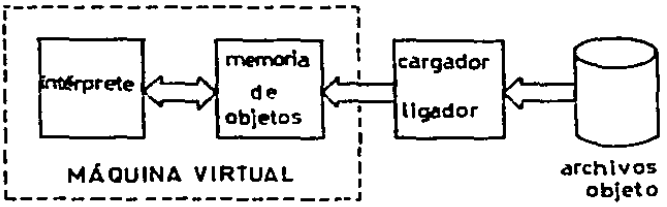

Fig. II.1 Diagrama de Bloques del Sistema

II.2 Concepto de Objeto y su Relación con la Máquina Virtual

Antes de continuar con la descripción del sistema es importate detenernos en el concepto de objeto.

El concepto de "objeto" en computación lleva consigo una serie de aspectos que de acuerdo a la experiencia debe de cumplir toda implantación de los mismos. En lo sucesivo se empleará el termino "objeto" para designar la representación en la memoria de la computadora de algún concepto o entidad que el programador desea modelar. Un "objeto" no es entonces más que una colección de datos almacenados en cierta cantidad de celdas de memoria. Las operaciones sobre un objeto se definen en función de operaciones de lectura y escritura sobre las celdas de memoria asignadas a él. Peter B. Bishop en su tesis [Bishop 77] da una lista de las restricciones que son inherentes a los objetos. Estas restricciones sirvieron como guía en el diseño de los módulos de la máquina virtual, especialmente para el diseño del manejador de la memoria de objetos. Las restricciones son:

a) Para identificar a cada objeto se utiliza una especie de "nombre" del objeto, diferente para cada objeto. A estos nombres se les denomina apuntadores (o referencias) El apuntador a un objeto se crea al ser creado el objeto correspondiente.

b) Un objeto solo puede manejarse a través del uso de su apuntador. Si se posee el apuntador a un objeto se pueden realizar operaciones sobre él. El apuntador debe entonces contener la dirección de las celdas de memoria del objeto. Gracias a la existencia de los apuntadores, los usuarios del objeto no necesitan conocer la dirección del objeto en la memoria.

c) Se prohiben las modificaciones al contenido de la referencia a un objeto a los usuarios del objeto (Generalmente las referencias a objetos constan de varios campos, como la dirección del objeto, que son usados por el sistema y que deben ser transparentes a los usuarios)

d) Cuando se utiliza la referencia a un objeto como medio de acceso a la memoria, solo se podrá leer o escribir sobre la memoria destinada a almacenar la representación del objeto a que corresponde la referencia.

e) Una referencia a objeto deberá estar siempre relacionada al mismo objeto.

f) Cada objeto es el modelo de un objeto ideal conocido en forma intuitiva por el programador; todas las operaciones sobre el objeto deben ser consistentes con el objeto ideal que se esta modelando.

Además de las restricciones anteriores, existe una serie de criterios que deben seguirse para lograr que el costo del mecanismo de referencia a objetos no se eleve demasiado. Son:

g) La referencia a un objeto no deberá ser mucho mayor que una dirección de memoria. Esta restricción no marca el límite exacto máximo del tamaño de una referencia, simplemente nos indica que la información mínima que debe contener una referencia es la dirección del objeto en la memoria y que la información adicional necesaria para tener acceso a un objeto no debe ser

centenares de veces mayor que una dirección.

h) Toda referencia a objeto que esté almacenada en algún lugar del sistema deberá corresponder a un objeto almacenado en otra parte del sistema. No existe referencia sin objeto y cada referencia corresponde a un solo objeto.

i) Los objetos pueden tener cualquier tamaño, desde un bit (valores lógicos) hasta millones de palabras.

j) Las referencias a objetos pueden ser copiadas libremente.

k) Siempre se podrá definir un nuevo tipo de objeto.

l) Cuando se define un objeto, se podrán definir las operaciones existentes en el sistema que podrán trabajar sobre ese objeto así como las nuevas operaciones.

Algunos de estos criterios de diseño especifican propiedades del esquema de direccionamiento del sistema, otros especifican el grado de flexibilidad que se necesita, y otros especifican restricciones en el uso de referencias a objetos, de direcciones dentro de esas referencias, y de la habilidad de definir nuevas operaciones sobre un objeto.

II.3 Manejador de la Memoria de Objetos

Teniendo ahora un poco más claro el concepto de objeto, podemos proseguir con la máquina virtual, en particular con el manejador de la memoria de objetos.

Las actividades que debe realizar el administrador de la memoria pueden ser agrupadas en dos categorías. La primera categoría consiste de aquellas actividades que proporcionan el acceso a los objetos. La segunda consiste de las actividades que permiten alojar y desalojar los objetos.

Como se ha mencionado, el manejador de la memoria de objetos debera facilitar el manejo de objetos al intérprete. Las funciones generales del manejador de la memoria de objetos son las siguientes:

1. Crear nuevos objetos. Dada la clase del objeto y la longitud (número de campos) se regresara el objeto (en realidad el apuntador al objeto).

2. Accesar cualquier campo de un objeto. Dados el apuntador al objeto y el número del campo se regresará el valor del campo.

3. Actualizar cualquier campo de un objeto. Dados el apuntador al objeto, el número del campo (desplazamiento dentro del objeto) y el nuevo valor, reemplazar el valor anterior por el nuevo.

Ademas deberá realizar funciones de mantenimieto, como creación de espacios en la memoria para poder alojar más objetos y destrucción de objetos que ya no se necesiten. Para llevar a cabo esas funciones deberá:

4. Compactar los objetos. Desplazar los objetos residentes en la memoria con el fin de crear un espacio libre más amplio.

5. Recolectar basura. Desalojar de la memoria aquellos objetos que ya no se necesiten.

II.4 Intérprete

El compilador del sistema TM se encarga de producir las respuestas compiladas (RC) que pueden convertirse en respuestas ejecutables (RE) si son alojadas en la memoria de objetos por el cargador-ligador. Las RE son objetos del tipo primitivo CODIGO (ver capítulo IV que habla sobre la memoria de objetos). Cada objeto RE contiene una secuencia de bytes que forma las instrucciones para un intérprete. El intérprete ejecuta los códigos de ocho bits para realizar las acciones descritas en la respuesta fuente.

Es necesario recordar aquí que se ha implantado el sistema TM como máquina virtual con el fin de evitar dependencias innecesarias con el hardware, de manera que el compilador sea transportable y se pueda tener un conjunto de instrucciones más adecuado al manejo de objetos que el que podría proporcionar una computadora convencional.

Para realizar cualquier inspección o alteración de un objeto en TM es necesario mandarle un mensaje. Debe existir un objeto de código o bien una "rutina primitiva" que de "respuesta" al mensaje. El envío de mensaje es un mecanismo análogo a la invocación de una subrutina y se puede considerar a las rutinas primitivas como funciones intrínsecas del sistema y a los objetos de código como código objeto de rutinas programadas por los usuarios. Una de las funciones más importantes del intérprete es la de controlar el envío de mensajes.

II.5 Cargador Ligador de Código

En general, los cargadores son programas que aceptan código objeto de programas compilados o ensamblados, preparan este código para que sea ejecutado por la computadora e inician la ejecución del mismo. Las funciones que debe efectuar un cargador son (según [Donovan 72]):

a. Obtener espacio en la memoria para alojar los programas (alojamiento).
b. Resolver las referencias simbólicas que se presenten entre módulos del programa (ligado).
c. Ajustar todas las direcciones dentro del programa de acuerdo a la posición en la memoria donde se logró alojar al mismo (relocalización).
d. Depositar físicamente los códigos de máquina dentro de la memoria.

Para ejecutar un programa en TM es indispensable alojar en la memoria de objetos a todos aquellos objetos a los que se hace referencia en el programa. La lista de objetos necesarios está compuesta por los administradores de todos los tipos de objetos usados, las respuestas que aparezcan en los mensajes que se envíen dentro del programa y todos aquellos objetos globales que son usados a lo largo del programa.

II.6 Referencias

[Donovan 72] Donovan J. John, "Systems Programming" McGraw-Hill, 1972.

[Bishop 77] Bishop, B. Peter, "Computer Systems with a Very Large Address Space and Garbage Collection", Ph.D. Thesis, M.I.T. Lab. for Comp. Scien. May 5, 1977.

CAPITULO III

Memoria de Objetos

III.1 Introducción

La memoria de objetos es el lugar donde residen los objetos en TM.

El objetivo principal del manejador de la memoria de objetos consiste en permitir al interprete actuar sobre los objetos. Para lograr esto, el manejador debe permitir:

-Crear nuevos objetos.
-Accesar cualquier campo de un objeto.
-Actualizar cualquier campo de un objeto.

y como se explicará en este capitulo, tambien deberá realizar funciones de mantenimiento como:

-Compactar los objetos.
-Recolectar basura.

Para llevar a cabo sus objetivos el manejador de la memoria de objetos tiene que resolver las siguientes dificultades:

-El tamaño de los objetos no es fijo. Existe una gran variedad en el número de campos que puede tener un objeto (No existe limitante en el lenguaje).

-La cantidad de objetos que se requerirán durante la ejecución de un programa es impredecible.

-Las relaciones entre los diversos objetos (referencias o apuntadores entre objetos) pueden ser cíclicas (se pueden formar redes recursivas).

En este capitulo se incluye primeramente la especificación formal de la memoria de objetos como un tipo de datos abstracto, despues se presenta la selección de técnicas de implantación que permitieron construir la memoria de objetos cumpliendo con las especificaciones y las características de la memoria de objetos ya construida. El capitulo termina con la descripción de las operaciones que se pueden realizar sobre la memoria de objetos implantada y la revisión de que las operaciones cumplen con las especificaciones.

III.2 Especificación Formal de la Memoria de Objetos

El diseño de sistemas computacionales ha pasado de ser una tarea "artesanal" a una profesión ingenieril. Es necesario contar con la especificación del sistema que se desea construir para poder emprender la implantación del mismo. Una especificación clara y no ambigua permite revisar si el sistema implantado satisface las necesidades para las que se construyó. Si un sistema cuenta con una especificación clara y completa de su funcionamiento (especificación que sí cumpla el sistema), se puede utilizar al sistema para la construcción de otros sistemas. La especificación sirve como una documentación del sistema, que entre más clara y completa sea, será más útil.

Hoare menciona en [Hoare 82] "ahora podemos construir especificaciones de programas con la misma precisión con la que un ingeniero puede establecer el lugar adecuado para un puente o una carretera; y sobre esta base podemos construir programas que cumplan con las especificaciones con la misma certeza con que el ingeniero asegura que su puente no caerá"

Las especificaciones formales e informales se complementan. Las informales son más faciles de leer y entender (al menos para las personas que entienden los tecnicismos que frecuentemente se emplean en este tipo de especificación) mientras que las formales tienden a ser más claras, precisas y no ambiguas.

El propósito de esta tesis no consiste en la elaboración de una especificación formal de la máquina virtual en su conjunto, sino del diseño e implantación de la misma cumpliendo algunas restricciones y especificaciones descritas de manera informal. Se eligió al módulo de la memoria de objetos para realizar su especificación formal ya que se le consideró como la pieza conceptualmente más compleja de la máquina virtual. Del funcinamiento correcto del módulo de la memoria depende el funcinamiento de los demás módulos ya que el interprete debe trabajar continuamente sobre los objetos de datos y código y el cargador ligador debe depositar en la memoria los objetos de código para que sean ejecutados por el interprete.

Existen diversas técnicas para realizar especificaciones formales de sistemas. La técnica que se emplea en la presente especificación es la algebraica. Para una introducción a las técnicas de especificación formal se recomienda [Liskov 75] y [Guttag 77]. En general las especificaciones algebraicas constan de dos partes:

a) Sintaxis - Se especifican las operaciones del sistema indicando cuantos argumentos requieren y de que tipos deben ser tanto los argumentos como el resultado de las operaciones (Existe cierta analogía entre la parte sintáctica de la especificación algebraica y la parte pública en la definición de un administrador en TM. En ambas aparece la

descripción de los dominios de las operaciones o respuestas que actúan sobre el tipo que se está definiendo, pero no se especifica como actúan la operaciones o respuestas).

b) Semántica - Se establecen ecuaciones algebraicas (axiomas) que relacionan los valores que se obtienen de las operaciones cuya funcionalidad fue descrita en la parte de sintaxis definendo el comportamiento de las mismas para cualquier juego de argumentos de entrada.

III.2.1 Tipo de Datos Abstracto de la Memoria de Objetos

La memoria de objetos queda caracterizada por la siguiente tupleta de 12 elementos :

M = (MemOb, Ap, N, inicia_mem, libres, es_asignado, crea_obj, long_obj, clase_de_obj, campo_de_obj, asig_en_campo, Nil, Lim)

Los tres primeros elementos son los conjuntos:

MemOb : Conjunto de estados de la memoria de objetos.
Ap : Conjunto de apuntadores.
N : Naturales incluyendo al cero.

Sintaxis

Los elementos restantes corresponden a funciones con los siguientes dominios:

Nil	:	-> Ap
Lim	:	-> N
inicia_mem	:	-> MemOb
asig_en_campo	: MemOb X Ap X Ap X N	-> MemOb
crea_obj	: MemOb X Ap X N	-> MemOb X Ap
libres	: MemOb	-> N
es_asignado	: MemOb X Ap	-> Booleano
long_obj	: MemOb X Ap	-> N
clase_de_obj	: MemOb X Ap	-> Ap
campo_de_obj	: MemOb X Ap X N	-> Ap

Semántica

Las operaciones que realiza cada una de las funciones anteriores están definidas a través de una serie de axiomas. En la presentación de los axiomas aparecen variables que pertenecen a alguno de los tres conjuntos de la tupleta. Las variables son:

apu1, apu2, cla y b1, b2 etc., c1, c2 etc. que pertenecen a Ap

n, m que pertenecen a N y M, M' que pertenecen a MemOb

Las primeras tres funciones son constantes que tienen el siguiente significado intuitivo:

Lim - Número de celdas en la memoria.
inicia_mem - Estado inicial de la memoria.
Nil - Apuntador especial que hace referencia a un objeto nulo.

Los Axiomas son:

i) libres(inicia_mem) = Lim
- El número de celdas libres en el estado inicial de la memoria es igual al número de celdas de la memoria.

ii) long_obj(M, Nil) = 1
- La longitud del objeto apuntado por Nil es uno en cualquier estado de la memoria.

iii) es_asignado(M, Nil)
- El apuntador Nil está asignado en todos los estados de la memoria.

iv) clase_de_obj(M, Nil) = Nil
- La clase del objeto apuntado por Nil es Nil.

v) no es_asignado(inicia_mem, apu1) para todo apu1, apu1 diferente de Nil.
- No existe apuntador asignado en el estado inicial de la memoria excepto Nil.

vi) campo_de_obj(asig_en_campo(M, apu1, apu2, n), apu1, m) =
apu2 si m = n
campo_de_obj(M, apu1, m) en otro caso
- Si almacenamos cualquier valor en el campo n'ésimo de un objeto y despues preguntamos por el contenido de ese campo, se contestará que el contenido es el valor que almacenamos previamente. Los demás campos permanecen inalterados.

vii) [crea_obj(M, cla, n) = (M', apu1) =>
[(0 < n ≤ libres(M)) y (apu1 ≠ Nil)
y (es_asignado(M', apu1))
y (long_obj(M', apu1) = n)
y (libres(M') = libres(M) - n)
y (campo_de_obj(M', apu1, m)
= Nil; 0 < m ≤ n-1)
y (clase_de_obj(M', apu1) = cla)
y NoVariaExcepto(M, M',apu1)
]
ó
[(n ≤ 0 ó n > libres(M))
y (apu1 = Nil)

y (M = M')

)

donde
NoVariaExcepto(M, M', apu1) =
(es_asignado(M', apu2) = es_asignado(M, apu2) para toda apu2 en Ap, apu2 ≠ apu1)
y (long_obj(M', apu2) = long_obj(M, apu2) si es_asignado(M, apu2) y apu2 ≠ apu1)
y (campo_de_obj(M', apu2, m) = campo_de_obj(M, apu2, m) si es_asignado(M, apu2) y apu2 ≠ apu1)
y (clase_de_obj(M', apu2) = clase_de_obj(M, apu2) si es_asignado(M, apu2) y apu2 ≠ apu1)

- En la creación de un objeto de longitud n se pueden presentar dos casos:

a) Existen suficientes celdas de memoria para satisfacer el requerimiento (el número de celdas libres es mayor que n) y el requerimiento es legal (n es un número mayor que cero). En este caso el apuntador correspondiente al nuevo objeto será diferente de Nil y se incluirá dentro de los apuntadores asignados. La longitud del objeto será n, el número de celdas libres de la memoria disminuirá en n elementos. Todos los campos del objeto tendrán un valor inicial de Nil, la clase del objeto será el objeto apuntado por cla (parámetro pasado a la operación de creación). El estado de la memoria despues de la creación de un objeto permanecerá igual exceptuando los cambios antes mencionados (no se afectarán los objetos que ya residan en la memoria)

b) No existen suficientes celdas de memoria o el requerimiento es ilegal (se pide un objeto de tamaño menor o igual a cero). En este caso el apuntador resultante será igual a Nil y el estado de la memoria se mantendrá sin variaciones.

Restricciones

Algunas de las funciones mencionadas son parciales. A continuación aparecen los casos para los cuales la aplicación de las funciones no es exitosa.

* no es_asignado(M, apu2) =>
failure(long_obj(M, apu2))
y failure(clase_de_obj(M, apu2))
y failure(campo_de_obj(M, apu2, n))
para toda n

y failure(asig_en_campo(M, apu1, apu2, n))
para toda n y para todo apu1
y failure(asig_en_campo(M, apu2, apu1, n))
para toda n y para todo apu1

- Para poder realizar con éxito las operaciones que permiten obtener o modificar la información que contiene un objeto, es necesario que el apuntador al objeto este definido, que apunte a un objeto que este alojado en la memoria.

* (long_obj(M, apu1) < n) ó (n < 0) =>
failure(asig_en_campo(M, apu1, apu2, n))
y failure(campo_de_obj(M, apu1, apu2, n))

- Las operaciones que hacen referencia a un campo de algún objeto fracasarán si el campo en cuestión no se localiza dentro del objeto. Esta situación se presenta cuando el índice del campo excede al tamaño del objeto o cuando el índice es negativo.

III.2.2 Modelo de la Memoria de Objetos

Una vez definido el ADT de la memoria, es importante verificar que es posible formular un modelo que cumpla cabalmente con la especificación. La existencia de un modelo nos asegura que los axiomas propuestos en el ADT no se contradicen o son imposibles de cumplir. Es deseable que el modelo sea similar a la implantación real del manejador de la memoria con el fin de tener mayor seguridad respecto a la validez de la implantación.

El modelo que se presenta a continuación sigue la especificación dada en el ADT de la Memoria de Objetos y es análogo en funcionamiento a la implantación real. Las variables que aparecen en el modelo siguen la misma convención de nombres que se estableció para el ADT. Los nombres de las funciones del modelo son iguales a los nombres de las funciones del ADT excepto en que aparecen con letras mayúsculas.

La tupleta correspondiente a la caracterización del modelo es la siguiente:

Mn = (MO, H, N, INICIA_MEM, LIBRES, ES_ASIGNADO, CREA_OBJ, LONG_OBJ, CLASE_DE_OBJ, CAMPO_DE_OBJ, ASIG_EN_CAMPO, NIL, LIM)

Se tienen las siguientes correspondencias entre los conjuntos en de la tupleta del ADT y en el modelo:

MemOb --> MO
Ap --> N
N --> N

donde

MO = ((libres, asignados) :
libres pertenece a Fin(N);
asignados pertenece a Fin(N X N X Fseq(N X N))
)

donde
Fin(N) representa los conjuntos finitos de N.
Fin(N X N X Fseq(N X N)) representa los conjuntos finitos de N X N X Fseq(N X N); Fseq(N X N) representa las secuencias finitas de N X N.
Cada estado de la memoria es una pareja (libres, asignados) cuyo primer elemento contiene las celdas libres que están disponibles para construir nuevos objetos. El segundo elemento contiene los objetos que existen en ese estado de la memoria. Cada objeto se representa por una tupleta de tres elementos. El primer elemento del trio es el apuntador (o identificador) del objeto. El segundo elemento es el apuntador a la clase del objeto y el tercer elemento es una secuencia de parejas que corresponden a los campos del objeto. Cada pareja de la secuencia representa al identificador de un campo (o número de celda de memoria) y al apuntador al contenido del mismo.

NIL = 0
LIM es una constante natural que representa el tamaño de la memoria de objetos.

En el modelo se utiliza la siguiente operación sobre secuencias:

n'ésimo(sec, n) - Es una función que dada una secuencia sec y un índice n regresa el elemento que ocupa la posición n dentro de la secuencia. Ejem: n'ésimo(<a b c>, 2) = b. Es una función definida solamente cuando la longitud de la secuencia es mayor o igual al índice n y cuando el índice es mayor que cero.

Tambien se utiliza una operación sobre parejas ordenadas
codom((a, b)) = b
que dada una pareja nos regresa el segundo elemento de la misma.

- INICIA_MEM = ([1,...,LIM], [(NIL, NIL, <>)])

donde <> es la secuencia vacia.
Se observa que están libres todas las celdas (de 1 a LIM) y el único objeto asignado es el objeto nulo.

- LIBRES((lib, asi)) = | lib |

| lib | es la cardinalidad de lib
Existen tantas celdas libres como elementos contenga el conjunto lib.

- ES_ASIGNADO((lib, asi), apui) = Existe un elemento (apui, cla, seq) en asi

- CREA_OBJ((lib, asi), cla, n) =
  ((lib', asi'), bi) si 0 < n ≤ LIBRES((lib, asi))
  ((lib , asi), NIL) en caso contrario

donde:

lib' = lib - {b1, b2,..., bn} donde bi pertenece a lib
1 ≤ i ≤ n
bi ≠ bj para i ≠ j
1 ≤ i = j ≤ n

asi' = asi U
{(bi, cla, <(b2, NIL) (b3, NIL)..(bn, NIL)>)}

- LONG_OBJ((lib, asi), apui) =
  :: seq :: + 1 si existe (apui, apu2, seq) en asi
  no definido en otro caso

:: seq :: representa el número de elementos en la secuencia seq. El número de celdas que utiliza un objeto corresponde al número de elementos en su secuencia de campos más la celda destinada a almacenar la clase del objeto.

- CLASE_DE_OBJ((lib, asi), apui) =
  cla si existe (apui, cla, seq) en asi
  no definido en otro caso

- CAMPO_DE_OBJ((lib, asi), apui, n) =
  codom(n'ésimo(seq, n))
  si existe (apui, cla, seq) en asi
  no definido en otro caso

Cada elemento de la secuencia de campos de un objeto es una pareja de celda-contenido. La operación CAMPO_DE_OBJ obtiene el contenido de un campo.

- ASIG_EN_CAMPO((lib, asi), apui, apu2, a) =
  (lib, asi') si ES_ASIGNADO((lib, asi), apui)
  y ES_ASIGNADO((lib, asi), apu2)
  y 1 ≤ a < LONG_OBJ(lib, asi), apui)
  no definido en otro caso

donde
asi' = asi - {ele_apui}
U {ele_apui'}

ele_apui es el elemento
(apui, cla, <(bi, ci) (b2, c2)..(ba, ca)..(bn, cn)>)
y ele_apui' es
(apui, cla, <(bi, ci) (b2, c2)..(ba, apu2)..(bn, cn)>)

Es decir así' = así excepto en que se sustituye el elemento ele_apui por el elemento ele_apui'.

Es importante notar en este punto que la formulación de las operaciones sobre la memoria de objetos no necesitó en ningún momento de la realización de una "compactación" (ver sección III.3.4). Esto se debe a la modelación de las celdas disponibles como un conjunto y no como una secuencia. Los sistemas que manejan objetos se simplificarían si se contara con memorias de conjuntos de celdas y no de secuencias de celdas.

III.2.2.1 Revisión de que el Modelo cumple la Especificación

El axioma (i) se refiere al número de celdas libres de la memoria inicial. En el modelo se tiene el siguiente resultado al aplicar la función LIBRES a INICIA_MEM:

```
LIBRES(INICIA_MEM)  =  LIBRES({{1,..,LIM},{(NIL,  NIL,  <>)}})
                    =  ;  {1,..,LIM}  :
                    = LIM
```

por lo que se cumple el axioma (i).

Los axiomas (ii), (iii) y (iv) aseguran que tanto la longitud como la clase del objeto NIL se mantendrán constantes a lo largo de todos los estados de la memoria así como que dicho objeto estará siempre alojado. Para asegurarse del cumplimiento de estos axiomas podemos partir de que se cumplan en el estado inicial de la memoria para luego analizar si las operaciones que efectúan cambios en el estado de la memoria permiten que se sigan cumpliendo los axiomas.

Para el estado inicial de la memoria tenemos:

```
LONG_OBJ(INICIA_MEM,  NIL)
        =  LONG_OBJ({{1,..,LIM},  {(NIL,  NIL,  <>)}},  NIL)
        =  ;  <>  ;  +  1
        = 1
```

por otro lado

```
ES_ASIGNADO(INICIA_MEM,  NIL)
        =  ES_ASIGNADO({{1,..,LIM},  {(NIL,  NIL,  <>)}},  NIL)
        =  VERDADERO
```

y

```
CLASE_DE_OBJ(INICIA_MEM,  NIL)
        =  CLASE_DE_OBJ({{1,..,LIM},  {(NIL,  NIL,  <>)}},  NIL)
        = NIL
```

por lo que los axiomas (ii) a (iv) se cumplen para el estado inicial de la memoria. Las operaciones que cambian el estado de la memoria son CREA_OBJETO y ASIG_EN_CAMPO.

La operación CREA_OBJETO puede modificar el conjunto de objetos asignados al aumentarle un nuevo elemento (no quita

elementos que ya existan) por lo que el elemento que representa a NIL permanecerá inalterado y dentro del conjunto. El único peligro consistiría en que el nuevo elemento fuera de la forma (NIL, cla, seq) con NIL como primer elemento de la tripleta. Al crear un objeto, la operación CREA_OBJETO toma celdas que son elementos del conjunto de libres. Una de ellas será el primer elemento de la tripleta que define al objeto. El conjunto de libres contiene inicialmente a los números del 1 a LIM y no contiene a NIL = 0. Al conjunto de libres solo se le quitan elementos, nunca se le añaden nuevos, por lo que es imposible que se forme un nuevo objeto con NIL como primer elemento de la tripleta. Por lo tanto la operació CREA_OBJETO no impide que se sigan cumpliendo los axiomas (ii) a (iv).

La operación ASIG_EN_CAMPO no está definida si se aplica al objeto NIL ya que dicho objeto no contiene campos. Por lo cual las aplicaciones exitosas de la operación ASIG_EN_CAMPO no afectan al objeto NIL y no impiden que se sigan cumpliendo los axiomas (ii) a (iv).

El axioma (v) se refiere a los objetos que están asignados en el estado inicial de la memoria. En el modelo se tiene:

ES_ASIGNADO(INICIA_MEM, apui)
= ES_ASIGNADO(({1,..,LIM}, {(NIL, NIL, <>)}), apui)
= Existe un elemento (apui, cla, seq) en el conjunto {(NIL, NIL, <>)}
= Verdadero si apui = NIL
= Falso en otro caso ya que no existe otro objeto en el conjunto

por lo que el axioma (v) se cumple.

El axioma (vi) nos muestra las propiedades de las operaciones de asignación y lectura de campos de objetos. Para verificar si el modelo lo cumple solo debemos revisar los efectos de las operaciones en el modelo. Cuando la asignación de un objeto apu2 a un campo a-ésimo del objeto apui está definida se modifica el conjunto de objetos asignados al sustituir el elemento

(apui, cla, <(b1, c1) (b2, c2)..(ba, ca)..(bn, cn)>)

por el elemento

(apui, cla, <(b1, c1) (b2, c2)..(ba, apu2)..(bn, cn)>)

Si el estado de la memoria antes de llevar a cabo la asignación es la pareja (lib, asi) y el estado resultante de la asignación es (lib, asi'), la operación de lectura de campos tiene los siguientes resultados:

CAMPO_DE_OBJ((lib, asi), apui, a) = codom((ba, ca)) = ca
CAMPO_DE_OBJ((lib, asi'), apui, a) = codom((ba,apu2))= apu2

y como los elementos restantes de la secuencia no se alteran se

tiene:

CAMPO_DE_OBJ((lib, asi), apui, n) =
CAMPO_DE_OBJ((lib, asi'), apui, n) para n ≠ a

por lo que se cumple el axioma (vi).

El axioma (vii) se refiere a los efectos que ocurren al realizar la creación de un objeto. Al crear un nuevo objeto en el modelo se tiene:

CREA_OBJ((lib, asi), cla, n) =
((lib', asi'), bi) si 0 < n ≤ LIBRES((lib, asi)) . . . caso (a)
((lib , asi), NIL) en caso contrario . . . caso (b)

donde:

lib' = lib - {b1, b2,..., bn} donde bi pertenece a lib
1 ≤ i ≤ n
bi ≠ bj para i ≠ j
1 ≤ i = j ≤ n

asi' = asi U
{(bi, cla, <(b2, NIL) (b3, NIL)..(bn, NIL)>)}

En el caso (a) se cumplen las siguientes propiedades:

- bi ≠ NIL ya que bi se tomó del conjunto lib que no contiene a NIL.
- ES_ASIGNADO((lib', asi'), bi) ya que a asi' contiene al elemento (bi, cla, <(b2, NIL) (b3, NIL)..(bn, NIL)>).
- LONG_OBJ((lib', asi'), bi)
= :: <(b2, NIL) (b3, NIL)..(bn, NIL)> :: + 1
= n - 1 + 1
= n
- LIBRES((lib', asi'))
= ; lib' ;
= ; lib - {b1, b2,..,bn} ;
= ; lib ; - n
= LIBRES((lib, asi)) - n
- CAMPO_DE_OBJ((lib', asi'), bi, m)
= codom(n'ésimo(<(b2,NIL) (b3,NIL)..(bn,NIL)>, m))
= NIL para 0 < m ≤ n-1
- CLASE_DE_OBJ((lib', asi'), bi)
= cla
- Los objetos que están en memoria no se modifican ya que el nuevo conjunto de asignados es igual al anterior con la única diferencia de que tiene un elemento más que corresponde al objeto creado.

En el caso (b) se cumple que:

- La pareja que la función devuelve como resultado tiene como primer elemento al estado de la memoria que se pasó como entrada (no se modifica el estado de la memoria) y como segundo elemento al apuntador NIL (se regresa el objeto nulo ya que no se pudo crear el objeto pedido).

Se puede concluir que el modelo cumple totalmente con la especificación formal dada en el ADT.

III.3 Selección de Técnicas para Implantación

En las secciones anteriores se establecieron las restricciones y características que especifican el comportamiento que debe tener la memoria de objetos. En esta sección se discuten las técnicas que permitieron construir la implantación de la memoria de objetos cumpliendo con las especificaciones.

El primer problema a resolver consiste en encontrar la representación más adecuada de los objetos tomando en cuenta las características de las memorias reales disponibles (arreglos secuenciales de celdas de memoria).

Si ya se dispone de una forma de representar los objetos, se debe resolver el problema referente al alojamiento y borrado dinamico de los objetos dentro de la memoria.

III.3.1 Representación de los Objetos

Se presentaron dos alternativas para la representación de los objetos en la memoria. Una consiste en la utilización de nodos del mismo tamaño (como en el espacio de celdas en

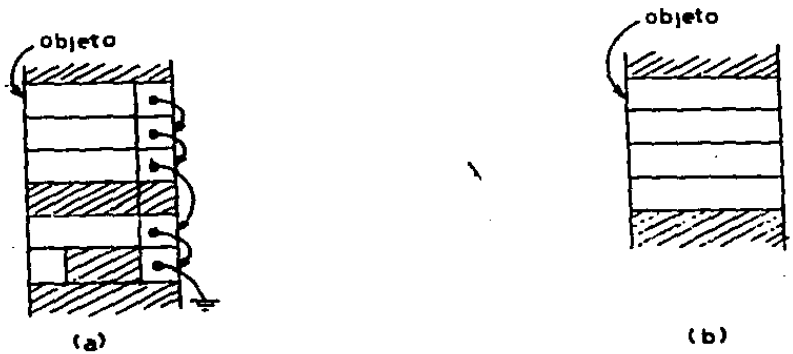

Figura III.1. Representación de objetos utilizando nodos de tamaño fijo (a) y nodos de tamaño variable (b).

sistemas LISP, McCarthy[1962]) y la otra se basa en el uso de

nodos de tamaño variable (ver fig. III.1).

-Nodos del mismo tamaño. Si se sigue esta opción se pueden acomodar objetos de diferentes longitudes apartando el número de nodos necesario y ligando los nodos entre sí.

Esta solución presenta ineficiencia respecto al tiempo requerido en el acceso de la n-ésima celda, ya que es necesario recorrer la cadena de apuntadores. Si el objeto contiene muchas celdas es necesario recorrer grandes cadenas de apuntadores. Otro problema consiste en que se requiere que cada celda tenga espacio reservado para el (o los) apuntador(es) que la conectan con otras celdas por lo cual existe una alta proporción de espacio que no se utiliza para almacenar los datos reales del objeto (dicha proporción dependerá del tamaño que se fije a las celdas y si las celdas tienen uno o dos apuntadores a las demás celdas).

Las celdas de tamaño fijo tienen espacio para un número fijo de apuntadores, por lo que las estructuras multiligadas con diverso número de apuntadores por nodo no se pueden representar convenientemente.

Si el tamaño de los nodos es mayor que uno, se presenta el fenómeno de fragmentación interna que consiste en que si el espacio requerido no es múltiplo de la longitud del nodo, se tendrá espacio desperdiciado en el último nodo destinado al objeto.

-Nodos de diferentes tamaños. Se incrementa la eficiencia en la utilización del espacio de almacenamiento. El tiempo de búsqueda de un campo dentro del objeto puede decrecer ya que para accesar cualquier campo es necesario solamente sumarle un desplazamiento a la dirección de inicio del objeto. Tiene la desventaja de que los algoritmos para manejar estructuras multiligadas de tamaños variables son más complicados.

En esta implantación del sistema TM se decidió la utilización de nodos de diferentes tamaños debido a las ventajas antes mencionadas. Además se tiene los beneficios adicionales de que la partición de la memoria es lógica y no física ya que cada nodo representa un solo objeto que está almacenado en localidades adyacentes del heap. No existe fragmentación interna ya que a cada objeto se le asigna únicamente el espacio que necesita.

El interprete hace la petición al manejador de la memoria de alojar un objeto cuando se ejecuta una instrucción que explícitamente sirve para la creación de algún objeto (create, new, etc.). Las rutinas que realicen el manejo de estas situaciones deberán realizar las acciones siguientes. Primero deberán asignar un apuntador identificador que servirá para referirse al objeto nuevo. Despues deberán encontrar espacio libre suficiente de almacenamiento dentro de la memoria de objetos para contener al objeto que se requiere. En seguida deberán dar valor inicial a las estructuras de datos internas

(por ejemplo una entrada en la tabla de objetos o el campo de longitud del objeto) usadas para representar al objeto. Finalmente es deseable que los campos dentro del objeto tengan un valor inicial conocido (nil).

III.3.2 Técnicas de Alojamiento Dinámico en Memoria

Las técnicas de alojamiento dinámico en memoria se utilizan para resolver el problema consistente en administrar regiones lineales contiguas de memoria de manera que cuando se requiera alojar bloques de n palabras contiguas se pueda satisfacer el pedido para n variable, además de permitir que los bloques que ya no se necesiten puedan ser liberados y realojados de acuerdo a alguna demanda posterior.

Podemos dividir las técnicas de alojamiento dinámico en tres categorías: Métodos de entrada secuencial (Sequential-Fit), sistemas compañeros (Buddy-Systems) y métodos de almacenamiento segregado. Todas dividen a los bloques de memoria en dos clases: libres y reservados.

A continuación se explicará brevemente el funcionamiento de cada una de las categorías antes mencionadas.

Métodos de Entrada Secuencial. En esta categoría caen métodos tales como el primero que sirva (first fit), el mejor que sirva (best fit), el óptimo que sirva y el peor (que sirva). Estos métodos se caracterizan por que los bloques libres se ligan en una lista secuencial. Cuando se requiere un bloque de memoria de tamaño n se realiza una búsqueda a través de la lista ligada hasta encontrar un bloque del tamaño deseado. El bloque se separa de la lista y se convierte en un bloque reservado. La política para realizar la búsqueda es la que define las diversas variedades de métodos en esta categoría.

Sistemas Compañeros. En estos métodos la memoria es dividida en bloques de tamaños preestablecidos. Entre los tamaños más usados están las potencias de 2 ($1, 2, 4, 8, \ldots, 2^n, \ldots$) y los números de Fibonacci comenzando desde el 3 ($3, 5, 8, 13, 21, \ldots, Fn, \ldots$).
Los bloques con estos tamaños tienen la peculiaridad de que se pueden subdividir en bloques más pequeños cuyos tamaños son números que tambien están en la secuencia. Los bloques libres de los tamaños elegidos se colocan en listas de bloques para cada tamaño. Para satisfacer un requerimiento de tamaño n, se localiza un bloque de tamaño $k \geq n$. De ser posible se subdivide la porción sobrante de tamaño $n - k$ en bloques más pequeños que son colocados en las listas adecuadas. Para conocer estos métodos se pueden seguir las sigs. referencias: [Knowlton 65], [Knuth 1973], pp. 442-445 o [Standish 80] pp. 257-264.

Métodos de Almacenamiento Segregado. Existen diversas

modalidades para esta categoría. Una consiste en dividir la memoria en páginas que contendrán bloques de tamaño uniforme. A cada página corresponde un tamaño de bloque. Se mantiene un índice a cada página y un conjunto de listas de bloques libres de tamaño uniforme. Otra variante consiste en mantener un número dado de listas cada uno de las cuales liga bloques del mismo tamaño (o en un rango de tamaños).

Las pruebas realizadas por Weinstock indican que el método del mejor que sirva (best-fit) proporciona la mejor utilización de memoria, mientras que los sistemas compañeros (buddy systems) pueden satisfacer los requerimientos de manera más rapida. Para combinar las ventajas de los dos métodos Wulf, Weinstock y Johnsson diseñaron un método llamado quickFit, el cual consisten en mantener n listas (15 en el caso de ese diseño) con bloques de tamaño uniforme 1 a n respectivamente. Un requerimiento de bloque en el rango de 1 a n se maneja consultando primero la lista apropiada para encontrar un bloque de tamaño exacto. Si la lista correspondiente está vacia, un bloque de mayor tamaño se busca en las listas siguientes. Finalmente si no se encuentra ninguno en las listas, se busca en otra lista de los bloques restantes que contiene bloques de tamaño mayor que n. La última búsqueda se realiza por primero que sirva (first-fit). El método se basa en el hecho de que los objetos que son más solicitados son aquellos cuyo tamaño está dentro del rango que cubren las listas de 1 a n.

Las pruebas realizadas por Weinstock muestran que el tiempo requerido promedio realizado para reservar o liberar un bloque es ligeramente menor para el método de Quick-Fit que para sistemas compañeros y mucho menor que para los métodos secuenciales. Lo mismo sucede con el número de bloques que es necesario examinar antes de satisfacer un requerimieto. El desperdicio de memoria por fragmentación interna en sistemas compañeros va de 25 a 41 porciento de la memoria mientras que en quickFit no hay desperdicio.

Para nuestra implantación elegimos el método quickFit con listas para tamaños de 2 a 20 (aunque no es difícil variar estos límites). Este rango fue elegido en forma arbitraria será necesario realizar pruebas posteriores sobre aplicaciones del sistema para determinar un rango óptimo.

III.3.3 Recolección de Basura

La actividad de reclamar espacio de memoria es la segunda función importante del administrador de la memoria. En TM se permite que un programa pida explícitamente que se cree un objeto, sin embargo, no es necesario que el programador haga la petición de desalojar los objetos. Una vez que un objeto se ha depositado en la memoria debe seguir existiendo mientras pueda ser accesado por otros objetos. Un objeto puede ser desalojado solamente cuando no existen referencias a el. Es responsabilidad

del administrador de la memoria desalojar automáticamente todos los objetos inaccesibles. A este proceso se le denomina recolección de basura.

La razón principal de que no exista en TM una operación de borrado o desalojo de objetos estriba en que no puede desaparecer un objeto sin afectar la estructura reticular a la que pertenece. Se tendrían que localizar todas las referencias al objeto para apuntarlas a un objeto nulo. Todos los objetos que son referidos por el objeto a ser borrado deberían ser a su vez borrados en caso de que solo pudieran ser accesados a través del objeto inicial.

Antes de realizar la ejecución de un programa en TM existe un estado inicial de la memoria de objetos. El programa actúa sobre los objetos del estado inicial de la memoria para producir un estado final. Dicho estado final de la memoria de objetos será a su vez el estado inicial del siguiente programa TM que se ejecute. El estado actual de la memoria de objetos depende de la ejecución anterior de un número (tal vez grande) de programas. Comunmente los programas habrán sido tecleados y ejecutados interactivamente y no se contará con los textos fuente.

Si se permitiera al programador borrar objetos a su antojo, se podría crea una inconsistencia dentro de la red de objetos y sería necesario restaurar el sistema.

En sistemas de programación en los que los programas no parten de un estado de la memoria que depende de la ejecución de los programas anteriores es común que el programador pueda borrar explícitamente los objetos, teniendo la responsabilidad de evitar inconsistencias, como la de que existan apuntadores a un objeto ya borrado. Si se llegara a crear alguna inconsistencia, el programador es el responsable de encontrar el error que la produjo, corregirlo y restaurar el sistema.

En un medio ambiente como TM, no es deseable que se necesite restaurar el sistema cada vez que apareciera un error común, ya que ello implicaría el tener que rehacer una cantidad potencialmente grande de trabajo.

Los únicos objetos que se pueden desalojar sin causar cambios en la red son los que se convierten en inaccesibles. Los objetos que se vuelven inaccesibles, es decir, que no son apuntados por otros objetos, deben ser identificados de alguna manera para que puedan ser desalojados de la memoria. La labor de reconocer y borrar los objetos debe realizarse de manera automática.

Existen dos enfoques tradicionales para identificar los objetos inaccesibles, el conteo de referencias y el marcado por búsqueda exhaustiva.

i) Marcado por búsqueda exhaustiva. Consiste, como su nombre lo

indica, de una búsqueda de todos los objetos que son accesibles a partir de las raíces del sistema. Todos los objetos que son accesibles se marcan. Esta marca puede hacerse destinando un bit para cada objeto que se "marca" al encenderlo o apagarlo. Una vez realizado el marcado se procede a recorrer toda la memoria en busca de los objetos no marcados, que son los inaccesibles, y se desalojan de la memoria. Este tipo de recolectores se han venido usando desde finales de los años cincuentas cuando surgieron los primeros lenguajes de procesamiento de listas.

ii) Conteo de referencias. Este esquema mantiene un contador que indica cuantos apuntadores, cuantas referencias existen hacia el objeto. Cuando el contador de algún objeto alcanza el valor cero, se sabe que el objeto ha quedado inaccesible y que el espacio que ocupa puede ser reclamado (este esquema fue usado por primera vez por Collins y Weizenbaum [Cohen 81]).

Los sistemas que utilizan conteo de referencias no funcionan adecuadamente cuando se presentan las llamadas estructuras cíclicas. Este tipo de estructuras ocurre cuando existen referencias de un objeto a si mismo en forma directa o bien a través de otros objetos que se refieren a el. Cuando una estructura cíclica se vuelve inaccesible los contadores de los objetos que la forman tienen valores diferentes de cero debido a la forma de la estructura. Estos objetos gastan espacio pero no causan ningun efecto en los demás objetos del sistema. Otro problema de este esquema reside en el tiempo que se debe gastar para actualizar los contadores cada vez que se realiza una nueva referencia a algún objeto. Por otra parte los recolectores que marcan deben realizar búsquedas a traves de la memoria que pueden consumir una cantidad considerable de tiempo. Como la activación del recolector es automática, el programador no sabe el momento en que se iniciará ese proceso, de manera que la ejecución del programa se verá sujeta a interrupciones de duración relativamente larga que pueden resultar muy molestas en un sistema interactivo.

La elección del esquema a usar depende de las características del hardware de que se disponga. Para sistemas con poca memoria, menos de 256 kbytes, no es apropiado el uso de contadores de referencias ya que gastan espacio de memoria para los contadores y dejan en la memoria las estructuras cíclicas. En esos casos es muy apropiado el esquema de marcado ya que el recorrido de la memoria no requiere de mucho tiempo. Si el sistema tiene memoria del orden de los millones de bytes no es apropiado el uso del recolector que marca ya que el tiempo que tarda en realizar el marcado se vuelve intolerable. Si es un sistema con memoria virtual, el marcado se vuelve más costoso ya que se gasta mucho tiempo en accesos aleatorios a memoria secundaria mientras que el esquema de contadores se hace menos costoso ya que las estructuras cíclicas no reclamadas se quedan en memoria secundaria donde el desperdicio de espacio es menos costoso.

La presente versión del sistema TM en VAX utiliza ambos

esquemas. El conteo de referencias se realiza durante la operación normal y la recolección por marcado se efectúa periodicamente con el fin de desalojar posibles estructuras cíclicas no accesibles. Se aprovecha el recorrido de la memoria durante el marcado de los objetos para realizar una compactación en la memoria de los objetos accesibles.

Existen otras soluciones al problema de recolección de basura, como la propuesta por Bishop en su tésis, que están basadas en nuevas arquitecturas. Bishop propone una estrategia de recolectores de basura que operan en grandes memorias virtuales, del orden de los billones de bits, en los que sería impractico realizar la recolección en toda la memoria en una sola pasada. La solución consiste en recolectar solo en parte del espacio de dirección. La memoria se divide en áreas que pueden ser recolectadas independientemente. El sistema mantiene listas de referencias entre las áreas (referencias que salen y entran a las áreas). La administración de las tablas debe realizarse automáticamente y de manera que se minimicen las referencias entre áreas. Para un usar eficientemente este esquema se necesita de hardware especial para el manejo de memoria virtual.

III.3.4 Compactación

Para mantener un sistema interactivo que preserve un medio ambiente dinámico por intervalos largos de tiempo sin tener la

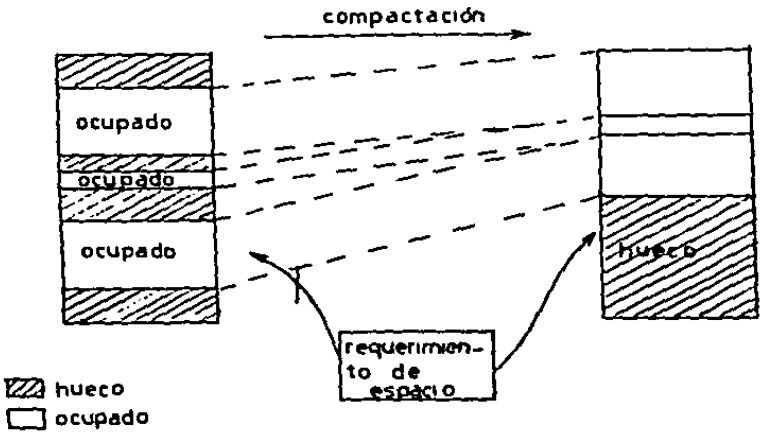

Figura III.2. Requerimiento de espacio antes y despues de la compactación.

necesidad de reiniciar el sistema es necesario resolver el problema de la fragmentación externa. Este tipo de fragmentación se presenta cuando existen "huecos", regiones no ocupadas por los objetos, en la memoria. Esos huecos pueden surgir al desalojar objetos inutiles. El problema surge al tratar de asignar memoria a un objeto nuevo. Puede ser que la suma del los espacios libres en la memoria sea mayor que la cantidad de memoria que requiere el objeto, sin embargo, no existe memoria contigua suficiente para ser asignada al objeto. La compactación proporciona la solución a este problema. Este proceso consiste en mover los objetos que están en uso (objetos ocupados) hacia un extremo del espacio de memoria dejando libre un solo bloque formado por el espacio restante. Lo anterior se ilustra en la figura III.2.

Cuando un objeto es movido de lugar durante la compactación, todos los apuntadores al mismo deben ser actualizados. Si existen muchos objetos que contienen apuntadores a la dirección vieja, se requiere de mucho tiempo para encontrar y actualizar todas las referencias. Para agilizar este proceso solo se permite un apuntador a la dirección del objeto. El apuntador es almacenado en una tabla llamada tabla de objetos. Los apuntadores a los objetos proporcionan índices a la tabla de objetos. En la tabla se encuentran las direcciones reales de los objetos.

Como ya se ha mencionado, la dirección de un objeto se obtiene a traves de encontrar la posición en la tabla de objetos que corresponde a ese objeto y extraer de allí la dirección correspondiente (ver fig. III.3). El uso de la tabla

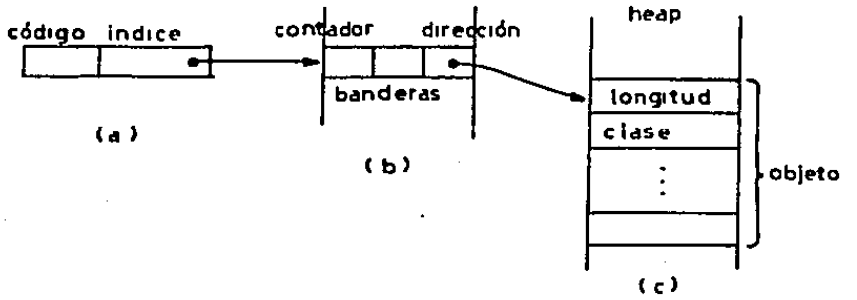

Figura III.3. En esta figura se aprecia en el extremo izquierdo a un apuntador (a) el cual sirve para localizar la posición en la tabla de objetos (b) donde se localiza la dirección del objeto en el heap (c).

además de facilitar el movimiento de los objetos en memoria permite desligar el tamaño del espacio de direccionamiento en

la memoria donde residen los objetos o heap del espacio que proporcionarían los apuntadores.

III.4 Características de la implantación

En la sección anterior se describen las técnicas que se eligieron para llevar a cabo la implantación de la memoria de objetos. En esta sección se describen las características concretas de la implantación tales como formato y longitud de los apuntadores, campos específicos de la representación de los objetos, número máximo de objetos que pueden existir en la memoria de objetos, etc.

Apuntadores

Los apuntadores a objetos tienen una longitud de 32 bits de los cuales se utilizan 28 (bits 0 a 27) para formar un índice a la tabla de objetos (ver fig. III.4) o para almacenar el valor del objeto. Los restantes cuatro bits se utilizan para almacenar un código que permite identificar a que tipo de objeto primitivo corresponde el apuntador. Al distinguir los objetos primitivos desde sus apuntadores se ahorran accesos a memoria, ya que no es necesario llegar hasta el objeto en sí para conocer el tipo primitivo al que pertenece el objeto, basta tener el apuntador.

El tamaño máximo de la tabla de objetos es de 2^{28}, y cada entrada de la tabla consta de tres campos, el campo que contiene la dirección en memoria del objeto, un campo con banderas de control y un campo que contiene el contador de referencias al objeto.

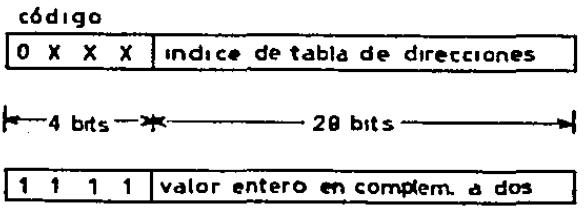

Figura III.4. Formato de un apuntador. Generalmente los apuntadores contienen índices de la tabla de direcciones (a). Para la representación de los enteros, el valor reside en el apuntador (b) con el fin de agilizar la obtención de su contenido y no ocupar espacio extra en el heap.

Campo de dirección del objeto en el heap. Este campo tiene una longitud de 32 bits con lo que se pueden direccionar hasta

4,294,967,296 localidades marcando con esto un límite máximo al tamaño del heap.

Campo de banderas. Este campo de 8 bits contiene las banderas que se utilizan en labores de mantenimiento de la memoria (como recolección de basura y en la compactación). Las banderas y su uso son:

-Libre. Esta bandera indica si esa posición en la tabla de objetos ya ha sido asignada a algún objeto. El bit estará encendido si la entrada no ha sido asignada. Esta bandera ocupa el bit 0 del campo.

-Real. Si el bit correspondiente está encendido indica que el campo de dirección al heap contiene el valor de un objeto de tipo Real y no una dirección.

-Primitivo_2. Este bit está encendido para objetos cuya representación dentro del heap solo consta de dos palabras.

-Marca. Para el recolector de basura de marcado por búsqueda exhaustiva se necesita una bandera que sirva para distinguir los objetos accesibles (marcados) de los que no lo son. Si la bandera de marca está encendida el objeto correspondiente es accesible.

-Banderas de Acceso. Los restantes cuatro bits del campo de banderas sirven para alojar el código de acceso al objeto.

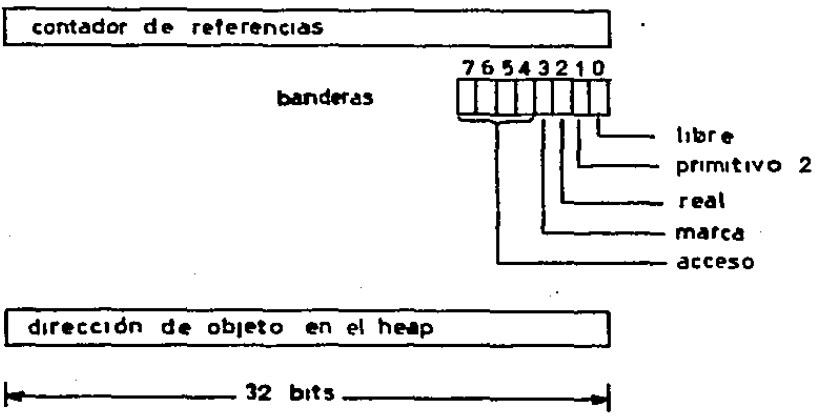

Figura III.5. Campos de la tabla de objetos

Campo de Contador de Referencias. Para la recolección de objetos por el esquema de conteo de referencias se necesita emplear un contador de referencias a cada objeto. Cada vez que se

realiza una nueva referencia a algún objeto es necesario alterara los valores de estos contadores y revisar si alguno ha alcanzado el valor cero o bien si se ha presentado un sobreflujo del contador. Aunque cada operación de conteo no es costosa, el tiempo acumulado en realizar continuamente estas operaciones puede llegar a degradar el funcionamiento del sistema. Para no hacer necesaria la revisión de sobreflujo, se fijo una longitud de 32 bits a este campo. En la figura III.5 aparecen los campos de la tabla de objetos.

Tipos Primitivos de Datos y su Representación en la Memoria de Objetos

En TM existe un conjunto de tipos primitivos de datos que definen la estructura y operaciones sobre los elementos más simples del sistema. Aunque son objetos en si mismos, dichos elementos son tambien los bloques de construcción de cualquier objeto del sistema. El manejo de los objetos primitivos en TM generalmente involucra el uso de funciones primitivas.

Los tipos primitivos son: Integer, Long, ConsCell, Real, Double, String, PC_Code, Record. Los objetos que pertenecen a estos tipos se caracterizan por que se puede identificar facilmente el tipo al que pertenecen ya que en su apuntador aparece un código que los distingue. A continuación se veran los diversos formatos que se utilizan para la representación de objetos de cada uno de esos tipos en las memoria.

Integer. Los enteros tienen una representación especial para hacer más eficiente su acceso. En lugar de contener en su apuntador un índice a la tabla de objetos, dicho campo de 28 bits se destina al almacenamiento del valor del entero en representación de complemento a dos. El valor puede ir de -2^{27} a $+2^{27} - 1$. En la fig. III.4 aparece la representación de un objeto de tipo Integer.

Long. Los enteros proporcionan un rango amplio de valores que pueden ser suficientes para muchas aplicaciones. En los casos en que sean necesarios valores enteros fuera del rango proporcionado por el tipo Integer, se puede utilizar el tipo Long cuya representación en la memoria aparece en la fig. III.6. Se dispone de un campo de 64 bits en el heap que nos permite representar valores enteros en el rango de -2^{63} a $+2^{63} -1$ utilizando complemento a dos. El bit primitivo_2 de la entrada para un objeto de tipo Long en la tabla de objetos se encuentra encendido.

Real. El tipo Real es el único que almacena sus valores en la tabla de apuntadores al heap. El campo LOC que usualmente apunta al heap sirve para almacenar la representación de un real en 32 bits. El formato de un real, que es identico a la convención usada en VAX-11 y PDP-11, es el siguiente (fig. III.7)

La fracción se expresa como un valor positivo de 24 bits,

donde 0.5 ≤ fracción < 1, con el punto binario colocado a la izquierda del bit más significativo. Como ese bit debe ser uno si el número es diferente de cero, no se almacena. Lo anterior permite que la fracción este almacenada en 23 bits. Esta forma de fracción es llamada normalizada. El exponente se almacena como un entero positivo recorrido 8 bits; esto es, cuando se le resta 128 al exponente, el resultado representa la potencia de 2 por la cual la fracción se multiplica para obtener el valor del número real. El signo del número es positivo cuando S, el bit de signo, es 0, y negativo cuando S = 1. El valor del número puede entonces obtenerse mediante la siguiente ecuación:

$$Valor = (1 - 2*S)*fraccion*2^{(exponente-128)}$$

donde

$$2^{-128} = 2.939 * 10^{-39} \leq Valor < 2^{127}*1.701*10^{38}$$

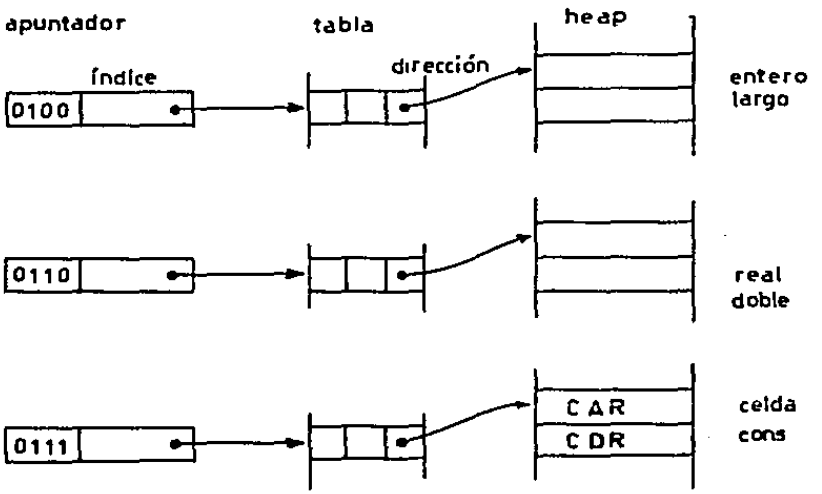

Figura III.6. Tipo de objetos con bandera de Primitivo_2 encendida

Double. Con un número de tipo Real se tienen asegurados siete dígitos decimales de precisión debido al tamaño de la fracción. Para permitir una mayor precisión se tiene el tipo Double. Un número de tipo Double tiene una precisión aproximada de 16 dígitos decimales gracias a que se añaden 32 bits a la fracción. En la fig. III.6 aparece el formato de un objeto Double.

Array. La representación de los arreglos consta de tres secciones como se aprecia en la fig. III.8. La primera sección

es el encabezado que está compuesto de dos palabras de 32 bits

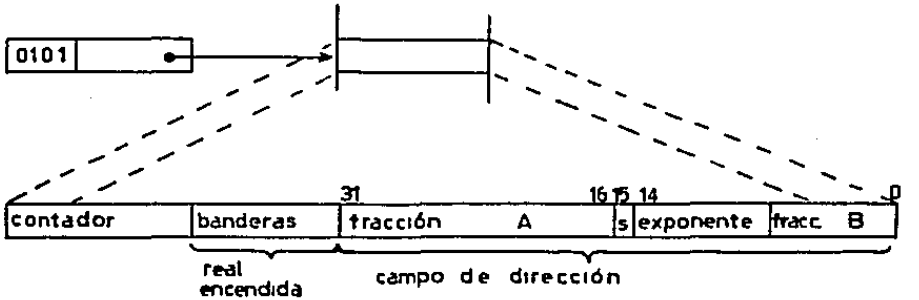

Figura III.7. Representación de un real. El valor reside en el campo de la tabla de objetos que se utiliza normalmente para almacenar la dirección del objeto en el heap

y que sigue las convenciones expuestas anteriormente para los objetos en general. La siguiente sección es la que se refiere a la definición de las dimensiones del arreglo. Esta sección está formada por n + 1 palabras del heap. La primera palabra nos indica el numero de dimensiones del arreglo y las restantes n nos dan las dimensiones del arreglo. La tercera sección es la que contiene los elementos (o posiblemente los apuntadores a los elementos) del arreglo.

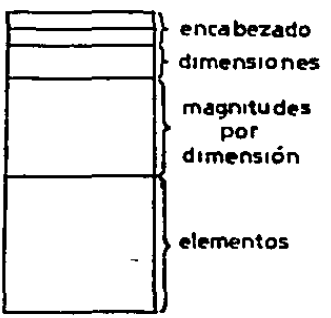

Figura III.8. Representación de un objeto tipo arreglo

ConsCell. Este tipo de objetos proporciona la herramienta utilizada en la construcción de listas. La representación de estos objetos se aprecia en la fig. III.6. Notese que este tipo de objetos carece del campo de encabezado y que debe tener encendido el bit de primitivo_2 ya que solo tiene dos palabras en

el heap. Dichas palabras sirven para almacenar los apuntadores al CAR y al CDR de la lista.

PC_Code. Uno de los tipos de datos más usados por el interprete es el de PC_Code. La representación para PC_Code consiste de tres secciones. La primera es el encabezado, la segunda es la correspondiente al llamado FRAME o conjunto de apuntadores a los objetos que serán utilizados por el código del objeto de tipo PC_Code. Esta sección se inicia por un campo que nos indica el número de apuntadores contenidos en el FRAME. En realidad es un desplazamiento al inicio de la tercera sección del objeto. La tercera sección del objeto es donde están contenidos los códigos de 8 bits que serán ejecutados por el interprete. Existen cuatro de estos códigos por palabra del heap. En la fig. III.9 se muestra la representación en la memoria de este tipo de objetos.

Record. Los objetos de este tipo son los de uso más general y son los que se utilizaran en la construcción de objetos complejos compuestos de diversos campos o atributos. Un objeto de este tipo está formado por dos secciones: la del encabezado y la de los campos del objeto. En aplicaciones de CAD es deseable que cada uno de los campos de un objeto este acompañado de las restricciones que debe de cumplir dicho campo. Es por ello que generalmente los campos pares se refieran a atributos del objeto y los impares a restricciones que deben de cumplir dichos atributos.

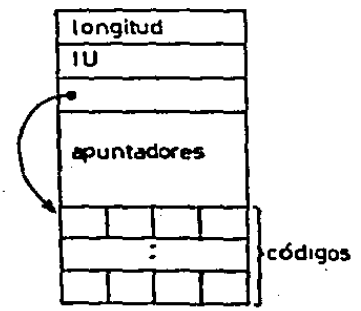

Figura III.9. Representación de objeto tipo código

Códigos correspondientes a los objetos primitivos:

```
0  - Record
1  - PC_Code
2  - *******
3  - Array
4  - Long
5  - Float
6  - Double
7  - ConsCell
8  - *******
.
.
14 - *******
15 - Integer
```

******* No usado

III.5 Interfaz con el Interprete y Comparación con el Modelo de la Especificación

En esta sección se presenta la descripción de las rutinas implantadas más importantes del manejador de la memoria de objetos. En unión con otras más especializadas (muy parecidas a las que aqui se describen pero que manejan la representación de objetos primitivos especiales como enteros o cuerdas de caracteres) definen la interfaz entre la memoria y el interprete. El interprete se comunica con la memoria de objetos exclusivamente a traves de la interfaz.

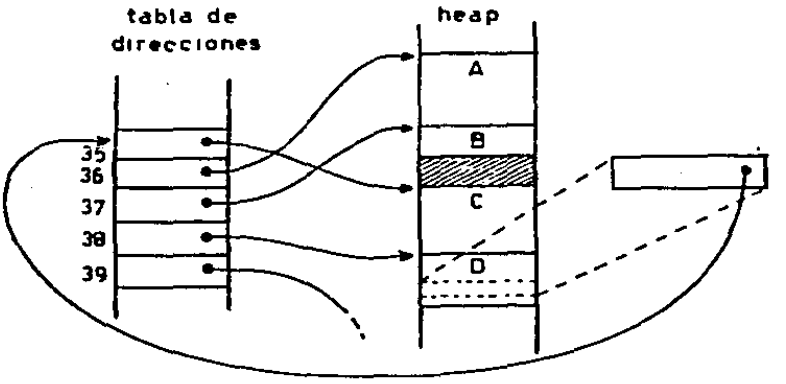

Figura III.10. Estructuras de datos principales de la memoria de objetos. La tabla de direcciones (que es parte de la tabla de objetos) y el heap.

En la especificación de la memoria de objetos se planteó un modelo que cumple con el tipo de datos abstracto de la memoria de objetos. En esta sección se mostrará informalmente que la implantación es "equivalente" al modelo.

En la figura III.10 se representan estructuras de datos principales de la memoria de objetos: la tabla de direcciones y el heap de objetos. Ambos son arreglos unidimensionales. Para tener acceso a un objeto en el heap se debe contar con su apuntador. El apuntador de un objeto es el índice en la tabla de direcciones donde reside la dirección en el heap del objeto. Un objeto posee un solo apuntador durante toda su vida, pero su dirección puede variar. En la figura se pueden observar varios objetos (A, B, C y D) y un hueco que aparece sombreado. Los objetos contienen apuntadores a otros objetos. En la figura se muestra uno de los campos del objeto D que es un apuntador al objeto C.

En la figura III.11 se muestran la partes principales de la representación de un objeto en el heap. Las partes son el encabezado y el cuerpo del objeto. El primer campo del encabezado (campo 0) contiene la longitud del objeto, es decir, el número de celdas de memoria (de 32 bits) destinadas al objeto, incluyendo las celdas del encabezado. El segundo campo (campo 1) contiene un apuntador a la clase o administrador del objeto. La clase o administrador es a su vez otro objeto en el heap. El cuerpo del objeto es una secuencia de campos destinados a almacenar apuntadores a otros objetos.

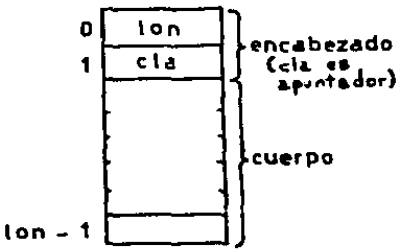

Figura III.11. Partes principales de un objeto en el heap

III.5.1 Funciones Principales de la Memoria de Objetos

Las estructuras de datos que se usan en la descripción de las funciones son:

Heap - Arreglo de celdas de memoria para almacenar los objetos. El tamaño de este arreglo está dado por la constante TamHeap.

TabOb - Tabla de direcciones de los objetos en el heap. Los índices de la tabla sirven como apuntadores a los objetos. El tamaño de esta tabla está dado por la constante TamTabOb. Los índices caen en el intervalo $0 \leq i < TamTabOb$.

Libre - Tabla de bits que indica si el apuntador i-ésimo está definido. Si TabOb[i] contiene la dirección de algún objeto en el heap, entonces Libre[i] = 0; se tendrá Libre[i] = 1 en caso contrario. El número de elementos de esta tabla es TamTabOb.

LisLibre - Lista de apuntadores no definidos. El apuntador i estará en la lista LisLibre si Libre[i] = 1.

ChunkLibre - Lista de huecos no ocupados por ningun objeto en el arreglo heap. En realidad no se implantó como una sola lista, pero por simplicidad se puede considerar así. La implantación consiste de un conjunto de listas cada una con huecos de un tamaño y una lista con huecos de tamaños varios.

Es importante revisar la relación que existe entre la memoria de objetos del modelo teorico y las estructuras de datos de la memoria de objetos real.

Los estados de la memoria de objetos en el modelo son elementos del conjunto MO (ver definición en sección III.2.2). Si m pertenece a MO, entonces m es una pareja de la forma (lib, asi).

El elemeto "lib" es un conjunto finito de números naturales. Si "i" pertenece a "lib" entonces la celda i-ésima de memoria está disponible para crear objetos.

Información equivalente a la que proporciona "lib" se puede obtener de la estructura de datos ChunkLibre. Si la celda i-ésima de memoria se encuentra dentro de un hueco dentro de ChunkLibre entonces la celda está disponible para ser usada en la creación de objetos.

El elemento "asi" es un conjunto finito de tripletas:

asi = { (apu1, cla1, seq1), (apu2, cla2, seq2),..
..,(apun, clan, seqn)}

cada una de las cuales representa un objeto alojado en la memoria. El primer elemento de cada tripleta "apui" sirve como identificador del objeto. El segundo elemento representa la clase del objeto y el tercer elemento es la secuencia de campos del objeto. El elemento "asi" está representado en la implantación por el estado de las estructuras de datos TabOb y Heap.

La información de "asi" aparece en las estructuras de datos de la implantación de la siguiente forma: los primeros

elementos de la tripletas, "apui", corresponden a los índices de TabOb para los cuales Libre[i] = 0. Un objeto queda identificado por un índice de TabOb y no por su dirección en el Heap ya que las direcciones pueden variar (debido a compactaciones en el heap). El segundo elemento "clai" de una tripleta en "asi" corresponde al campo "clase" del encabezado de un objeto en el heap. El tercer elemento "seqi" corresponde con el cuerpo del objeto en el heap.

El estado de la memoria aparece como parámetro en las funciones definidas en el modelo. En la implantación el estado de la memoria está definido por las estructuras de datos globales y no aparece como parámetro.

A continuación se enlistan las funciones principales que operan sobre la memoria de objetos (no se incluyen operaciones especiales para objetos primitivos como enteros u objetos con bytes). Para cada una de las funciones aparecen las siguientes descripciones:

NOMBRE - Es el identificador de la función.
SINOPSIS - Forma como está declarada en lenguaje C.
FUNCION - Breve descripción del objetivo de la rutina.
VALORES QUE REGRESA - Salidas y efectos que se obtienen al invocar la rutina.
COMPARACION CON EL MODELO - Aquí se comparan los efectos de las funciones implantadas y del modelo sobre la memoria de objetos real y teórica respectivamente.

NOMBRE
NilOb - Apuntador al objeto nulo

SINOPSIS
```
#define NilOb 0;
```

FUNCION
NilOb es una constante para identificar el apuntador al objeto nulo.

VALORES QUE REGRESA
El valor de NilOb es el entero 0. Este valor corresponde al índice 0 de la tabla de direcciones de objetos.

COMPARACION CON EL MODELO
En el modelo se cuenta con la constante NIL = 0

NOMBRE
IniciaMem - Crea estado inicial de la memoria de objetos

SINOPSIS
```
IniciaMem();
```

FUNCION

IniciaMem crea el estado inicial de la memoria de objetos, es decir, crea las estructuras de datos necesarias para poder alojar la imagen virtual.
Es necesario invocar esta función antes de utilizar las demás operaciones que afectan al estado de la memoria de objetos tales como CreaObj y AsigEnCa. En el estado inicial de la memoria no existen objetos asignados.

VALORES QUE REGRESA

El valor que se regresa directamente de la función carece de significado. Se regresa indirectamente un conjunto de estructuras de datos globales que conforman el estado inicial de la memoria. En la figura III.12 se pueden apreciar las estructuras de datos principales y su configuración inicial. En la tabla de direcciones todos los apuntadores son libres exceptuando el que corresponde a NilOb (todos tienen un 1 en el bit de libre exceptuando la entrada 0 de la tabla). Los apuntadores no hacen referencia al heap, sino que están ligados formando una lista de apuntadores libres. La lista termina con el elemento que apunta a NilOb. NilOb no es parte de la lista, solo sirve como terminador. El heap no contiene objetos.

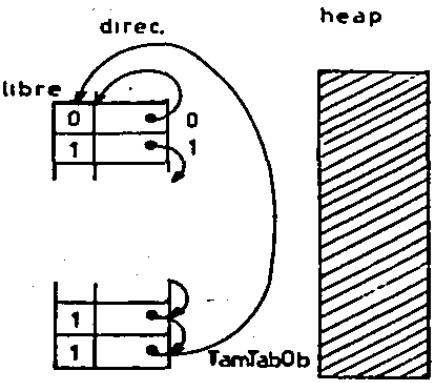

Figura III.12. Configuración inicial de la memoria

En Resumen:

Libre[i] = 1 para 0 < i < TamTabOb
Libre[0] = Libre[NilOb] = 0
LisLibre contiene todos los índices de TabOb

excepto el 0.
ChunkLibre contiene un solo hueco del tamaño del Heap.

COMPARACION CON EL MODELO

En el modelo se tiene el siguiente estado inicial de la memoria:

INICIA_MEM = ([1,..,LIM], [(NIL, NIL, <>)])

donde <> es la secuencia vacia.

El elemento [1,..,LIM] nos indica que todas las celdas están disponibles para ser usadas en la creación de nuevos objetos. Esto concuerda con la implantación ya que en LisLibre se encuentran los apuntadores 1,..,TamTaOb y en ChunkLibre se tiene un hueco del tamaño del Heap.

El elemento [(NIL, NIL, <>)] nos indica que el único objeto alojado es el identificado por NIL. En la implantación se tiene que Libre[NilOb] = 0 es decir que el apuntador NilOb está asignado y Libre[i] = 1 para otros valores de i, mostrando con esto que NilOb es el único asignado.

NOMBRE

CreaObj - Crea objeto

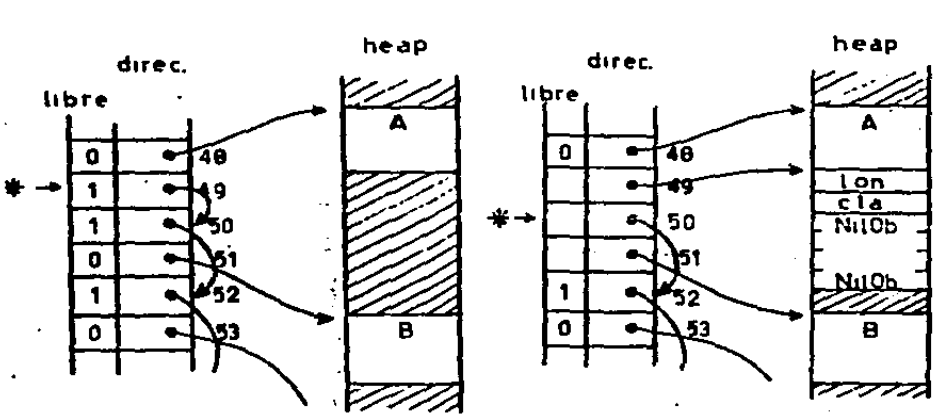

Figura III.13. Operación de creación de un objeto CreaOb(lon,cla) El simbolo * indica el inicio de la lista de apuntadores libres

SINOPSIS

```
long   CreaObj(longitud, clase)
long   longitud, clase;
```

FUNCION

Con esta función se crea un objeto con el número de celdas especificadas en el parámetro "longitud" y con el apuntador a la clase "clase". Ver fig. III.13. En esta figura se observa el efecto de la operación CreaObj. Notese que en el cuerpo del objeto creado todos los campos contienen el valor NilOb.

VALORES QUE REGRESA

Si existe espacio suficiente en el heap para almacenar el objeto de la longitud pedida y existen apuntadores disponibles (la lista de apuntadores libres no está vacia) se crea el objeto y se regresa como resultado el apuntador al mismo. En caso de no existir apuntadores libres o espacio en el heap aun despues de realizar la recolección de basura y la compactación, se despliega el mensaje:

Intento fallido de creación de objeto

Fin del espacio en el heap
y/o ***Fin de apuntadores disponibles***

Si el parámetro longitud es menor que 2 tampoco se puede crear el objeto y se despliga el mensaje:

ERROR, intento de creación de un objeto de longitud menor que 2.

y se regresa NilOb como resultado.

En caso de que se haya creado exitosamente el objeto pedido, se tendrá apunt = CreaObj(longitud, clase) y las las estructuras de datos se afectan de la siguiente manera:

LisLibre ya no cuenta con el elemento "apunt" que estaba a la cabeza de la lista.

Libre[apunt] = 0

ChunkLibre contiene un hueco menos del tamaño requerido o uno de sus huecos fue dividido quedando solo el sobrante o se realizo compactación y ChunkLibre solo posee un hueco donde se reunen todos las celdas no ocupadas del heap.

TabOb[apunt] = dirRes
donde dirRes es la dirección en el heap del hueco encontrado para alojar el objeto.

Heap[dirRes] = longitud
Heap[dirRes + 1] = clase
Heap[dirRes + i] = NilOb $1 < i < longitud$

COMPARACION CON EL MODELO

En el modelo se tiene la operación:

CREA_OBJ((lib, asi), cla, n) =
((lib', asi'), bi) si $0 < n \leq$ LIBRES((lib, asi))
((lib , asi), NIL) en caso contrario

donde:

lib' = lib - {b1, b2,.., bn} donde bi pertenece a lib

$1 \leq i \leq n$
$bi \neq bj$ para $i \neq j$
$i \leq i, j \leq n$

asi' = asi U
{(bi, cla, <(b2, NIL) (b3, NIL)..(bn, NIL)>)}

Para crear el objeto nuevo de longitud n se toman "n" elementos del conjunto "lib" uno de los cuales sirve para identificar al objeto recien creado (b1 es el primer elemento de la tripleta que se añade al conjunto asi). A cada campo del objeto recien creado se le asocia el valor NIL (ver secuencia que corresponde al objeto b1, con los campos identificados por b2, b3,..bn). El identificador del campo se puede interpretar como la dirección del mismo en la memoria de objetos. En el modelo los campos de un objeto no necesitan ser adyacentes (pueden ocupar direcciones no consecutivas del heap), por lo cual para cada campo se necesita especificar su dirección y su contenido.

En la implantación para crear un objeto de longitud n se necesita un hueco del heap de longitud n (que se toma de ChunkLibre) y además para identificar al objeto se require un apuntador que se toma de la lista LisLibre. Existe entonces una ligera diferencia entre el modelo y la implantación. En la implantación se requiere el elemento extra que se toma de LisLibre que tiene el papel de "b1" en el modelo. Es por esto que en la implantación se debe revisar además de la existencia de espacio en el heap la existencia de un apuntador libre en LisLibre antes de poder crear exitosamente un objeto.

La diferencia con el modelo desaparece si la creación de un objeto depende solamente de la existencia de suficiente espacio en el heap. Esto se logra si el tamaño del arreglo TabOb es suficientemente grande (al menos de tamaño TamHeap / 2 ya que el objeto de menor tamão ocupa dos celdas del heap). En esta implantación se tiene TabOb = TamHeap / 8 debido a que se estima que en promedio los objetos tendrán una longitud mayor que 8.

NOMBRE

AsigEnCa - Asigna valor en campo.

SINOPSIS

```
AsigEnCa(obj, val, offset)
long obj, val, offset;
```

FUNCION

Con esta rutina se puede almacenar un valor en el cuerpo de un objeto. El objeto se identifica por el apuntador "obj" que se pasa como parámetro. El valor a almacenar es el parámetro "val" y el campo donde se deberá almacenar el objeto se identifica por el parámetro offset.

VALORES QUE REGRESA

El resultado obtenido directamente carece de significado. En caso de que se cumplan las siguientes condiciones:

$$1 < offset < LongObj(obj)$$

y

$$EsAsig(obj)$$

y

$$EsAsig(val)$$

se obtiene indirectamente una nueva memoria de objetos en la cual el objeto con apuntador "obj" contiene en su campo "offset" el valor "val", es decir:

$$Heap[TabOb[obj] + offset] = val$$

En caso de no cumplirse las condiciones se despliega alguno de los mensajes:

ERROR, Intento de manejo de objeto indefinido
--Si obj o val no están definidos

o ERROR, Intento de acceso fuera del objeto
--Si offset se sale del rango

COMPARACION CON EL MODELO

En el modelo se tiene la operación:

ASIG_EN_CAMPO((lib, asi), apu1, apu2, a) =
(lib, asi') si ES_ASIGNADO((lib, asi), apu1)
y ES_ASIGNADO((lib, asi), apu2)
y 1 ≤ a < LONG_OBJ(lib, asi), apu1)
no definido en otro caso

donde
asi' = asi - {ele_apu1}
U {ele_apu1'}
ele_apu1 es el elemento

(apu1, cla, <(b1, c1) (b2, c2)..(ba, ca)..(bn, cn)>)
y ele_apu1' es
(apu1, cla, <(b1, c1) (b2, c2)..(ba, apu2)..(bn, cn)>)

el parámetro a funciona como un offset dentro de la secuencia de campos del objeto. Se asigna el valor apu2 al campo identificado por "ba".

En la implantación se realizan las mismas revisiones que se estipulan en el modelo: que el offset sea valido y que los apuntadores esten definidos.

NOMBRE

LongObj - Longitud del objeto

SINOPSIS

```
long   LongObj(obj)
long obj;
```

FUNCION

La función LongObj nos permite obtener la longitud de un objeto cuyo apuntador se especifica.

VALORES QUE REGRESA

Si existe algún objeto cuyo apuntador es "obj", es decir

Libre[obj] = 0

se regresa un valor entero que corresponde al número de celdas que ocupa la representación del objeto. Dicha longitud la obtiene del campo 0 del objeto, es decir:

LongObj(obj) = Heap[TabOb[obj]]

Se tiene que LongObj(NilObj) = 1 en cualquier estado de la memoria.

En caso ser un apuntador a algún objeto no definido se regresa el valor -1 y se despliega el mensaje:

ERROR, Intento de acceso a objeto indefinido

COMPARACION CON EL MODELO

En el modelo se tiene la operación:

LONG_OBJ((lib, asi), apu1) =
:: seq :: + 1 si existe (apu1, apu2, seq) en asi
no definido en otro caso

donde

:: seq :: representa el número de elementos en la secuencia seq.

Es equivalente a la implantación ya que ambas proporcionan el número de celdas que requiere el objeto.

En el modelo:

LONG_OBJ((lib, asi), NIL) = 1

En la implantación:

LongObj(NilOb) = 1

NOMBRE
ClaseObj - Clase del objeto

SINOPSIS
```
long   ClaseObj(obj)
long obj;
```

FUNCION
Esta función permite obtener el apuntador a la clase (o administrador de el objeto identificado por el apuntador que se pasa como parámetro.

VALORES QUE REGRESA
Si
Libre[obj] = 0
Se regresa el contenido del campo 1 del objeto. Es decir:
ClaseObj(obj) = Heap[TabOb[obj] + 1]
En el caso de NilObj se tiene:
ClaseObj(NilObj) = NilObj
En caso de ser un apuntador a algún objeto no definido se regresa el valor NilOb y se despliega el mensaje:

ERROR, Intento de acceso a objeto indefinido

COMPARACION CON EL MODELO
En el modelo se tiene:
CLASE_DE_OBJ((lib, asi), apui) =
cla si existe (apui, cla, seq) en asi
no definido en otro caso

Es equivalente a la implantación ya en ambos casos se obtiene como resultado el campo de la clase del objeto si el apuntador está definido y ocurre un error en caso contrario.
En el modelo:
CLASE_DE_OBJ((lib, asi), NIL) = NIL
En la implantación:
ClaseObj(NilOb) = NilOb

NOMBRE
CampoObj - Campo de un objeto

SINOPSIS
```
long   CampoObj(obj, offset)
long   objeto, offset;
```

FUNCION
Esta función regresa el contenido del campo de un objeto especificado por el apuntador "obj". El campo se identifica por el parámetro "offset".

VALORES QUE REGRESA
El valor de la función es el contenido del campo

especificado en caso de que se cumplan las condiciones expuestas para AsigEnCa. El resultado será entonces:

CampoObj(obj, offset) = Heap[TabOb[obj] + offset]

En caso de no cumplirse las condiciones se despliegan los mensajes:

ERROR, Intento de acceso a objeto indefinido

o ERROR, Intento de acceso fuera del objeto

COMPARACION CON EL MODELO

En el modelo se tiene la operación:
CAMPO_DE_OBJ((lib, asi), apui, n) =
codom(n'ésimo(seq, n))
si existe (apui, cla, seq) en asi
no definido en otro caso
que consiste en obtener el valor del n-ésimo campo del objeto identificado por apui en caso de que apui este definido. Esta función es equivalente a la implantación.

NOMBRE
EsAsig - Está asignado un apuntador a un objeto?

SINOPSIS
int EsAsig(obj)
long obj;

FUNCION
Revisa si el apuntador "obj" está asignado a algún objeto, es decir si el apuntador es o no libre.

VALORES QUE REGRESA
Se revisa el contenido del bit libre de la entrada en la tabla de objetos que corresponde a "obj". Si el bit de libre es 1 la función regresa un 0 y viceversa. Es decir:
EsAsig(obj) = 0 si Libre[obj] = 1 y
EsAsig(obj) = 1 si Libre[obj] = 0

COMPARACION CON EL MODELO

En el modelo se cuenta con el predicado:
ES_ASIGNADO((lib, asi), apui) = Existe un elemento
(apui, cla, seq) en asi
En la implantación se tiene una tabla de bits que para todos los apuntadores nos dice si están libres o asignados. Dicha tabla es "Libre".

El modelo cuenta además con la siguiente función

LIBRES((lib, asi)) = ¦ lib ¦

¦ lib ¦ : es la cardinalidad de lib

Esta función sirve para mostrar para un estado de la memoria el número de celdas disponibles para ser utilizadas en la creación de nuevos objetos.

Por razones de eficiencia no se implementó esta operación aunque está implícitamente construida a partir de las funciones de creación de objetos y recolección de basura.

Se puede considerar que esta función es interna.

III.6 Referencias

[Bishop 77] Bishop, B. Peter, "Computer Systems with a Very Large Address Space and Garbage Collection", Ph.D. Thesis, M.I.T. Lab. for Comp. Scien. May 5, 1977.

[Cohen 81] Cohen, Jacques, "Garbage Collection of Linked Data Structures", Computing Surveys ACM, Vol. 13, No. 3, pp. 341-367, September 1981.

[Goldberg 83] Goldberg, Adele, and Robson, David, Smalltalk-80: The Language and Its Implementation, Addison-Wesley, Reading, Mass., 1983.

[Guttag 77] Guttag, John, "Abstract Data Types and Development of Data Structures" Com. A.C.M. 20, 6 June 1977 pp. 240-247.

[Hoare 82] Hoare C. Antony, "Programming is an Engineering Profesion" Oxford University Computing Lab. 1982 pp. 1-19.

[Knowlton 65] Knowlton, K. C."A Fast Storage Allocator", CACM Vol 8 No.11 October 623-625.

[Knuth 73] Knuth, D. E. "The Art of Computer Programming; Vol 1 Fundamental Algorithms" Addison-Wesley, Reading, Mass.

[Krasner 83] Krasner, Glenn, editor, "Smalltalk-80: Bits of History, Words of Advice", Addison-Wesley, 1983.

[Liskov 75] Liskov, B. H., Zilles S. N. "Specification Techniques for Data Abstractions" IEEE Transactions on Soft. Eng. Vol SE-1, No. 1 March 1975 pp 7-18.

[McCarthy 62] McCarthy, J. et Al. "Lisp 1.5 Programmers Manual", MIT Press, Cambridge Massachusetts 1962.

[Standish 80] Standish, Thomas A. "Data Structure Techniques" Addison-Wesley, 1980.

CAPITULO IV

Intérprete

IV.1 Introducción

Ya se ha descrito la memoria de la máquina virtual. Falta describir la parte de la máquina virtual que es equivalente a la unidad central de proceso de una máquina real. El intérprete de la máquina virtual es el encargado de ejecutar los códigos que resultan de compilar los programas en TM.

IV.2 Intérprete de Stack

El intérprete implantado es de tipo stack teniendo una serie de "registros" especiales que permiten almacenar los valores más importantes para el estado de la ejecución.

El uso de arquitecturas tipo stack para máquinas (virtuales o en hardware) que soportan manejo de objetos no es nuevo. Entre los primeros sistemas se encuentra la Burroughs B5000 [Levy 84] que fue diseñada para compilar y ejecutar eficientemente lenguajes de alto nivel apoyandose en un conjunto de instrucciones orientado a stacks para ayudar a la evaluación de expresiones y la activación de procedimientos. Otro ejemplo más reciente lo constituye el intel iAPX 432 [Levy 84] que es un microprocesador diseñado principalmente para soportar programación bazada en objetos. En dicho procesador no existen registros generales y cada contexto (conjunto de datos que definen el estado de la ejecución de un mensaje) contiene su propio stack, pudiendo estar los operandos de las instrucciones en el stack o en la memoria (se permiten operaciones memoria-memoria).

La definición de la máquina virtual para SmaliTalk-80 tambien utiliza un conjunto de instrucciones orientadas al uso de stacks [Goldberg 83].

Existen evidencias [Keedy 83], [Keedy 78], [Tanembaum 78] de que las arquitecturas de stack son adecuadas para soportar lenguajes estructurados en bloques que permitan recursión y para la evaluación de expresiones anidadas sin gran esfuerzo. El uso de stacks permite manejar eficientemente el envío de mensajes en cascada. Cada mensaje necesita, además del receptor, un número diferente de argumentos. El problema de asignación de argumentos a registros de datos desaparece con el uso del stack ya que solo es necesario meter los argumentos en el stack en el orden en que aparecen en el envío del mensaje.

La máquina virtual diseñada por Fernando Jimenez [Jimenez 86] tambien corresponde a una arquitectura tipo stack.

Debido a las razones anteriores se implantó un intérprete de stack.

IV.2.1 Estado del intérprete

El comportamiento del intérprete depende del "estado" en que se encuentra y el valor de los códigos que ejecuta.
El estado del intérprete está definido por los siguientes elementos:

a) La Respuesta al mensaje que contiene los códigos que se están ejecutando.
b) El apuntador al siguiente código a ser ejecutado.
c) El stack del intérprete.
d) El receptor y los argumentos del mensaje al que le corresponde la Respuesta mencionada en el punto (a).
e) El conjunto de variables temporales que requiere la Respuesta.

Las instrucciones de envío de mensaje y de regreso de respuesta causan cambios en todos los elementos del estado del intérprete.

Los registros del intérprete son:

Nombre	Función
TopeContex	Es el apuntador al tope del STACK de contextos.
ApuntInstruc	Es el apuntador al siguiente código de instrucción a ser ejecutado. Es el equivalente al Program Counter.
ContexAct	Es el apuntador al contexto activo. La mayor parte del tiempo coincidirá su contenido con el de TopeContex.
recep	Contiene el apuntador al receptor del mensaje que se está respondiendo.
respu	Contiene el índice a la Respuesta que se está ejecutando. Es necesario notar que este registro no contiene un apuntador, sino un índice a la tabla de objetos.
resnu	Contiene el índice de la Respuesta a un mensaje que está en proceso de ser enviado. Este registro contiene un índice y no un apuntador.
nargu	Contiene el número de argumentos del mensaje que se está ejecutando.

El intérprete efectúa un ciclo repetitivo de cuatro pasos. El ciclo consiste en:

a) Obtener el código indicado por el registro ApuntInstruc de la respuesta referida por el registro respu.
b) Incrementar el contenido del registro ApuntInstruc.
c) Efectuar la instrucción que corresponde al código obtenido en el primer paso. La instrucción puede implicar un cambio en el estado del intérprete y por lo tanto un cambio en el contenido de ApuntInstruc.
d) Revisar si la instrucción no implica terminación de interpretación (instrucción Halt o error en la ejecución). En caso de no terminación regresa al paso (a).

En el paso (a) en realidad se extraen de la respuesta cuatro códigos de ocho bits cada vez (una palabra de 32 bits) con el fin de agilizar la ejecución.

En las siguientes secciones (IV.2.2 - IV.2.5) se describen las estructuras de datos más importantes para el funcionamiento del intérprete.

IV.2.2 Stack de Evaluaciones

El intérprete utiliza un stack para almacenar los argumentos y el receptor de los mensajes así como los resultados obtenidos en la evaluación del mensaje.

El stack del intérprete está implantado como una estructura datos ajena a la memoria de objetos con el fin de evitar la necesidad de tratarlo como otro objeto, lograndose un ahorro en en el número de accesos a memoria necesarios para obtener los datos del stack (recuerdese que para obtener datos de un objeto se necesitan dos niveles de direccionamiento).

Otra ventaja de no considerar al stack de evaluaciones como un objeto consiste en lograr lo que se ha dado en llamar conteo de referencias diferido (ver en capítulo III lo referente a conteo de referencias para recolección de basura). Esta técnica consiste en detectar que la mayoría de los cambios en el número de referencias a un objeto ocurre en forma transitoria, que existe un incremento en el número de referencias momentaneo seguido de un decremento que regresa al estado original. Lo anterior es particularmente cierto para el stack del intérprete ya que si se hubiese implantado como objeto, a cada objeto que se le hiciera un PUSH se le tendría que incrementar su contador de referencias (ya que existe la nueva referencia del stack) para despues decrementarlo al hacerle un POP.

En la siguiente figura (fig. IV.1) se observa como se utiliza el stack de evaluaciones cuando se ejecuta un mensaje genérico

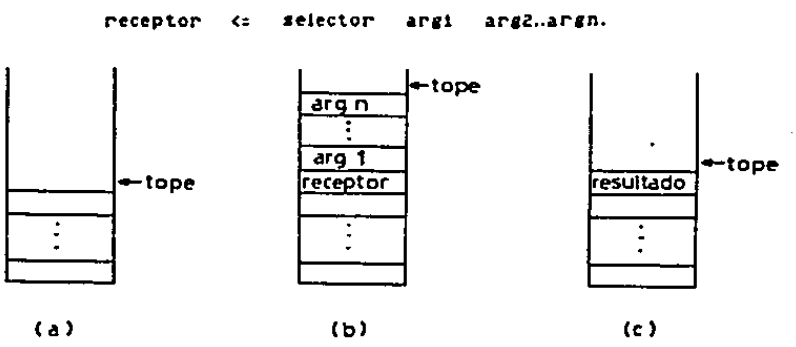

Figura IV.1. Stack de Evaluaciones. El primer paso para evaluar un mensaje consiste en depositar en el stack (a) el receptor y argumentos del mensaje (b). La respuesta o rutina primitiva toma los argumentos y el receptor del stack, evalua el mensaje y deposita el resultado en el stack (c).

IV.2.3 Contextos.

Cada vez que se efecúa una instrucción de envío de mensaje ocurre un cambio radical en el estado del intérprete. Una vez realizadas las operaciones relativas al mensaje, se debe reestablecer el estado en que se encontraba el intérprete antes del envío del mensaje (con un incremento en el apuntador de instrucción y posiblemente con el tope del stack conteniedo el resultado del mensaje). Para almacenar los estados del intérprete se usan estructuras de datos llamadas contextos.

El proceso de activación de la Respuesta correspondiente a un mensaje en TM es muy similar a la activación de un procedimiento en Algol. Dada la semántica del envío de mensajes en TM, la cadena dinámica de activación de Respuestas puede modelarse por un stack. Esto quiere decir que no es necesario conservar los contextos de todos los estados por los que pase el intérprete, solo es necesario conservar aquellos contextos que corresponden a un método o respuesta cuya ejecución no se ha terminado. Despues de realizar la operación final de una respuesta se puede destruir el contexto correspondiente.

En la presente implantación se decidió almacenar los contextos en un stack que por razones de eficiencia se encuentra fuera de la memoria de objetos. El ahorro principal consiste en el tiempo necesario para alojar y desalojar contextos ya que con

el stack solo es necesario desplazar su tope.

No debe confundirse al stack del intérprete que sirve para almacenar los parámetros (argumentos y receptor) y resultados de los mensajes con el stack formado por los contextos.

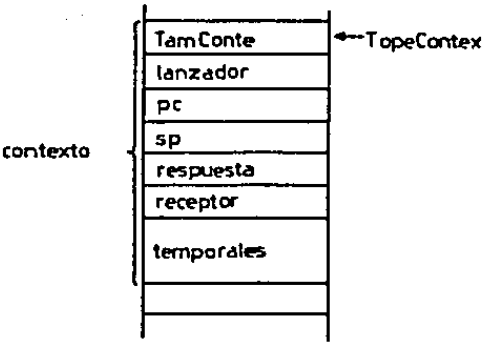

Figura IV.2. Formato de Contextos.

Formato de los Contextos

Los contextos constan de los siguientes campos:

Campo	posición	Contenido
TamConte	TopeContex	Longitud del contexto (6 + núm. de temporales) en la sección de temporales se almacenan los argumentos y las variables temporales.
lanzador	TopeContex - 1	Campo que indica quien envía el mensaje.
pc	TopeContex - 2	Apunt. a sig. Instrucción
sp	TopeContex - 3	Tope del stack de interp.
respuesta	TopeContex - 4	Es la respuesta compilada e instalada que contiene los códigos que el intérprete ejecuta.
receptor	TopeContex - 5	Receptor del mensaje cuya respuesta se está ejecutando.
InTempor	TopeContex - 6	Aquí comienza el marco de temporales que es donde se almacenan las variables temporales y

los parámetros del mensaje.

En la fig. IV.2 se observa el formato de un contexto en el stack de contextos.

IV.2.4 Respuestas Compiladas.

Las respuestas compiladas son objetos que contienen los códigos que ejecuta el intérprete. Este tipo de objetos corresponde al tipo primitivo código que se describió en el capítulo III. Al iniciarse la ejecución de los códigos de una respuesta compilada se crea un contexto que describe el estado de ejecución de esa respuesta. Además de los códigos, las respuestas compiladas contienen diversos campos para almacenar la información necesaria en la creación de los contextos correspondientes. El formato de una respuesta compilada es el siguiente:

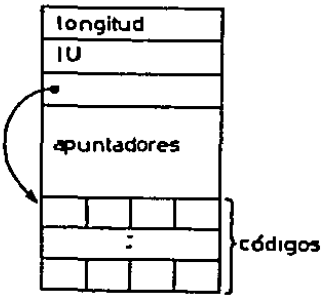

Figura IV.3. Formato de una Respuesta Compilada

Campo	Contenido
0	Longitud del objeto, número de campos de 32 bits
1	Identificador único de la Respuesta.
2	Longitud del marco de apuntadores de la respuesta. En realidad contiene el índice del campo siguiente fuera del marco de apuntadores. El primer apuntador sirve para apuntar al siguiente objeto de tipo respuesta compilada que corresponde a un Administrador.
2 + long. marco	El primer byte corresponde a la longitud del marco de temporales (número de argumentos + receptor + número de variables temporales).

El segundo byte se reserva para colocar el número de variables temporales (hasta ahora no se ha visto utilidad a esta información a tiempo de ejecución, pero el compilador debe revisar que no existan operaciones de Store sobre las los argumentos del mensaje).
Los siguientes bytes corresponden a los códigos de instrucción para el intérprete.

IV.2.5 Administradores.

Los objetos de tipo Administrador, que corresponden a objetos de tipo primitivo record, tienen tres partes principales:

- Lista de Administradores de las superclases del administrador.
- Lista de Respuestas Compiladas del administrador.
- Lista de Variables globales de clase del administrador.

La función de estos objetos consiste en ser los materiales de construcción de la red de clases y superclases que es necesario recorrer cuando se presenta la ejecución de mensajes cuya respuesta o método debe ser buscado (ver la sección IV.5.1 "Busqueda de Respuestas"). Otra aplicación de estos objetos consiste en almacenar las variables globales de clase del administrador. Las variables de clase de un administrador solo son visibles desde las respuestas del mismo administrador.

Los objetos de este tipo no pueden ser creados ni modificados directamente por el usuario, su manejo se restringe exclusivamente al intérprete.

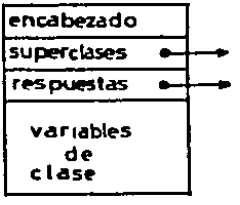

Figura IV.4. Formato de un Administrador.

IV.3 Conjunto de Instrucciones del Intérprete

Cada instrucción tiene asociado un código de 8 bits que la identifica.

Los objetos que pueden ser afectados directamente por las instrucciones son:

a) El receptor del mensaje.
b) Los argumentos del mensaje.
c) Las variables de instancia del mensaje.
d) Las variables temporales del mensaje.
e) Los objetos que se encuentren en el marco de literales del mensaje.
f) Cinco constantes especiales: Los enteros chicos -1, 0, 1, 2 y el apuntador nulo Nil.
g) Las variables globales de clase.
h) Las variable globales generales.

Para no realizar operaciones de corrimiento y enmascaramiento sobre los códigos de instrucción para determinar la operación correspondiente al código se utilizó la siguiente técnica: Cada código sirve como un índice en una tabla que "despacha" a la función correspondiente al código.

IV.3.1 Lista de Instrucciones

A continuación se enlistan las instrucciones del intérprete. cada instrucción se implantó como una función escrita en lenguaje C. Los apuntadores a cada función se almacenaron en un arreglo cuyos índices corresponden con los códigos asociados a la instrucciones. Los nombres de las instrucciones en la lista corresponden a los nombres de las funciones en C correspondientes.

NOMBRE	CODIGO	FUNCION
PushVR	0	Realiza el PUSH de la variable del receptor que se especifica en el siguiente byte.
PushVT	1	Realiza un PUSH de la variable del marco de temporales que se especifica en el siguiente byte.
PushVL	2	Realiza un PUSH de la variable en el marco de literales que se especifica en el siguiente byte.
PushEVR	3	Misma funcion que PushVR pero extendida ya que el campo está especificado por los dos siguientes bytes.
Pushm1	4	PUSH de un entero chico -1.
Push0	5	PUSH de un entero chico 0.
Push1	6	PUSH de un entero chico 1.
Push2	7	PUSH de un entero chico 2.
PushNil	8	PUSH de Nil.

PushRec	9	PUSH del receptor.
******	10	No usado
PopVR	11	Se realiza el POP y STORE en una variable del receptor que se especifica en el siguiente byte.
PopVT	12	POP y STORE en una variable del marco de temporales que se especifica en el siguiente byte.
PopVL	13	POP y STORE en una variable del marco de literales que se especifica en el siguiente byte.
PopEVR	14	Misma función que PopVR pero extendida ya que el campo del receptor se especifica en los dos siguientes bytes.
******	15	No usado
Retu	16	Regreso de mensaje (RETURN) del tope del STACK.
RetuRec	17	RETURN del receptor.
Retu0	18	RETURN del entero chico 0.
Retu1	19	RETURN del entero chico 1.
RetuNIL	20	RETURN de Nil.
******	21	No Usado.
Manda	22	Envio de un mensaje que será contestado por una RESPUESTA COMPILADA. El siguiente byte indica el desplazamiento para encontrar la respuesta en el marco de literales. El siguiente indica el número de argumentos del mensaje.
MandaNC	23	Se hace el envio de un mensaje cuya RESPUESTA no fue identificada en tiempo de compilación. El siguiente byte indica el desplazamiento en el marco de literales necesario para encontrar el IU de la RESPUESTA y el siguiente induca el número de argumentos del mensaje.
MandaPr	24	Activa RESPUESTA PRIMITIVA. El indice de la primitiva está en el siguiente byte.
******	25	No Usado.
Salto	26	Salto Corto Incondicional. De hasta 255 bytes. La longitud del salto se especifica en el siguiente byte.
SaltoM	27	Salto Mediano Incond. De 256 a 511. Long. = sig. byte + 256.
SaltoL	28	Salto Largo Incond. De 512 a

		767. Long. = sig. byte * 512.
******	29	No Usado.
******	30	No Usado.
Salto1	31	Salto de 1 byte.
SaltoT	32	Salto si tope = false.
SaltoF	33	Salto si tope = true.
Stop	34	Fin de interpretacion
NewObj	35	Creación de un nuevo objeto. la longitud la obtiene del siguiente byte y el tipo primitivo del stack. Se debe realizar antes un Push(tipo) antes de esta operación.
PushVA	36	Push de variable de clase del administrador. La variable se especifica en el siguiente byte.
PopVA	37	Pop y almacena en variable de clase del administrador. La variable se especifica en el siguiente byte.
PushVG	38	Push de variable global. La variable se especifica en el siguiente byte.
PopVG	39	Pop y almacena en variable global. La variable se especifica en el siguiente byte.

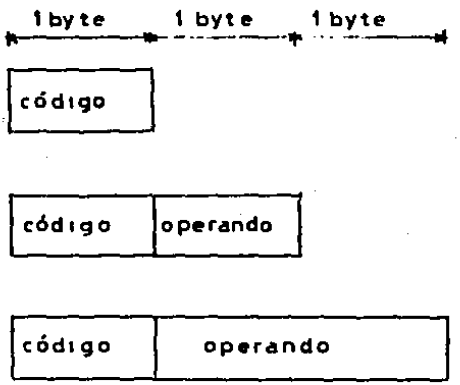

Figura IV.5. Instrucciones de la Máquina Virtual

IV.3.2 Instrucciones de Push y Pop en el Stack

El intérprete utiliza un stack para realizar la evaluación de expresiones, por ello necesita las operaciones de inserción (Push) y desalojo (Pop) del stack. Estas operaciones son simétricas (para cada tipo de Push existe una operación de Pop) en la mayoría de los casos. Las operaciones simétricas son: PushVR, PushVT, PushVL, PushEVR, PopVR, PopVT, PopVL, PopEVR, PushVA, PopVA, PushVG y PopVG.

Existen diversas operaciones de Push y Pop para especificar la fuente o destino de los datos sin necesidad de utilizar un parámetro extra. La especialización de cada operación permite ahorrar tiempo interpretación y permite además que cada operación se diseñe tratando de maximizar su velocidad de ejecución.

El formato general de las operaciones simétricas de Push es el siguiente:

Código (especifica fuente).	8 bits
Especificación del Campo dentro de la fuente.	8-16 bits

las posibles fuentes son: el receptor, el marco de temporales, el marco de literales, marco de variables de clase y marco de variables globales. En el caso de las operaciones de Pop se tiene:

Código (especifica destino)	8 bits
Especificación del Campo dentro del destino.	8-16 bits

los posibles destinos coinciden con las posibles fuentes de las operaciones de Push.

Las operaciones no simétricas (solo son operaciones Push) son aquellas que depositan una constante en el tope del stack. Estas operaciones se añadieron al conjunto de instrucciones por razones de eficiencia ya que se consideró que serán frecuentemente usadas y ahorrarán definiciones de literales y operaciones de Push tomando como fuente al marco de literales.

Otra operación no simétrica es la que deposita al receptor en el stack. No existe operación de Pop al receptor ya que el paso de parámetros al enviarse un mensaje en TM (el receptor puede considerarse como un parámetro) es por valor.

Las operaciones no simétricas son: PushMi, Push0, Push1, Push2, PushNil y PushRec. En estas operaciones solo es necesario especificar el código, ya que el operando está implícito.

IV.3.3 Instrucciones de Salto

Las operaciones de salto sirven para llevar a cabo las

instrucciones de transferencia de control en esta implantación de TM modificando el contenido del registro contador del programa. Todas los saltos son relativos. Existen dos tipos de salto: los condicionados y los no condicionados.

Las operaciones de saltos condicionados deben revisar el tope del stack para determinar el cumplimiento de alguna condición. Solo en caso de que la condición se cumpla, se realiza el salto. Dichas operaciones son: SaltoT y SaltoF.

Las operaciones de salto no condicionado modifican el contador del programa en función de la longitud del salto sin tomar en cuenta el contenido del stack. Las operaciones son: Salto, SaltoH, SaltoL y Saltoi.
El formato de estas operaciones es en general

Código (especifica tipo de salto)	8 bits
Longitud del salto	8 bits

IV.3.4 Instrucciones de Envío de Mensaje y Retorno

Existen instrucciones específicas para realizar el envío de un mensaje. Las modalidades se refieren a envío de mensaje cuya respuesta ya está determinada, envío de mensaje con búsqueda de respuesta y envío de mensaje a primitiva (ver sección IV.5 para mayor información sobre envío de mensajes). Las instrucciones son respectivamente: Manda, MandaHC y MandaPr. El formato de las dos primeras es:

Código	8 bits
Literal donde está el apuntador o el nombre de la respuesta	8 bits
Número de parámetros necesarios	8 bits

MandaPr tiene el siguiente formato:

Código	8 bits
Número de primitiva	8 bits

Un mensaje es una expresión cuyo resultado debe ser depositado en el stack. Existen en el conjunto de instrucciones varias operaciones para realizar esta tarea. Las operaciones de retorno de respuesta son: Retu, RetuRec, Retu0, Retui y RetuNIL. Se incluyeron estas operaciones especializadas por razones de eficiencia con el fin de ahorrar la realización de un Push seguido de un Return del tope del stack (con lo que solo se requeriría la operación Retu).

El formato de estas operaciones solo requiere del código de instrucción ya que el operando queda implícito.

IV.3.5 Instrucciones de Fin de Interpretación y Creación de Objetos

Existe un par de instrucciones que no pueden englobarse en ninguna de los grupos anteriores. La primera es una instrucción de terminación de interpretación que causa que se interrumpa la ejecución del programa. Su formato consiste únicamente del código. La segunda es una instrucción que sirve para crear objetos. El formato de la operación es el siguiente:

Código
Longitud del Objeto a Crear

Esta operación tiene implícitamente un parámetro que indica el tipo primitivo del objeto a crear. Dicho parámetro es un entero que debera estar en el tope del stack para cuando se realice la operación de creación.

IV.4 Operaciones Primitivas de la Máquina Virtual

Existe un conjunto de rutinas de respuesta a mensajes que no están construidas por medio de códigos para la máquina virtual. Este tipo de rutinas se refieren a los mensajes básicos que se mandan a objetos primitivos. Las rutinas están implantadas en lenguaje C y marcan el final de la recursividad en el envío de cualquier mensaje (todas las respuestas se construyen en base a la existencia de este conjunto de primitivas).

Igual que sucede para cualquier expresión de envío de mensaje, el receptor y los parámetros de la operación primitiva se toman del stack. El resultado de la primitiva se deja en el tope del stack. Para accesar al stack, las operaciones primitivas cuentan con las funciones push(argumento) y pop() que no deben confundirse con las diverasas operaciones de Pop u Push que existen en el conjunto de instrucciones del intérprete.

Las operaciones primitivas disponibles son:

Para enteros (chicos y largos)

Nombre de la función en C.	Número	Función
SumaPr	0	Receptor + Argumento
RestPr	1	Receptor - Argumento
MultPr	2	Receptor * Argumento
DiviPr	3	Receptor / Argumento
MenoPr	6	Receptor < Argumento
WriteEn	7	Despliega Receptor
wlent	16	Despliega Receptor y salta de linea
ReadEn	9	Lee valor entero
IguaPr	10	Receptor = Argumento
MenoIPr	12	Receptor ≤ Argumento

Para cuerdas de caracteres

WriteSt	11	Despliega cuerda

Para listas

ListNil	13	Receptor = lista Nula
ConsList	14	Inserta Argumento en lista receptor
head	15	Regresa el primer elemento de la lista receptora.
tail	16	Regresa la sublista sin el primer elem. del receptor.

Para booleanos

and	17	Receptor & Argumento

Para reales

MenoReal	19	Recept < Argumento
IguaReal	20	Receptor = Argumento
MenIReal	21	Receptor ≤ Argum.
WriteReal	22	Despliega Receptor
WriteLnReal	23	Despliega Receptor y salta de linea.
ReadReal	24	Lee Real

El conjunto de operaciones primitivas no se ha definido en la especificación del lenguaje TM. La elección de las operaciones que aparecen en la lista anterior dependió en gran medida de las aplicaciones que se programaron para probar el funcionamiento del sistema. Estas operaciones solo sirven como un modelo existiendo la posibilidad de modificarlas, reemplazarlas o agregar nuevas.

IV.5 Envío de Mensajes

Segun la definición que aparece en la referencia [Stefik 86] "el envío de un mensaje es una forma de llamar a un procedimiento". En lugar de invocar a un procedimiento para que efectue alguna operación sobre un objeto, se manda el mensaje a un objeto. Un selector en el mensaje especifica la clase de operación. Cada selector está asociado a un "procedimiento" que en lenguajes orientados a objetos es llamado respuesta o método.

Como se explica en la sección IV.3, existen dos tipos de instrucciones para envío de mensajes; uno que supone que la RE que contestará al mensaje ya está plenamente identificada desde el momento de compilación y otro tiene que buscar la RE

correspondiente.

El único caso en que se generaran los códigos del segundo tipo aparece cuando se envían mensajes a variables de tipo ANYTHING, mismas pueden ser asignadas a objetos de cualquier clase durante la ejecución. Como no es posible predecir a tiempo de compilación la clase de un receptor tipo ANYTHING, la identificación de la respuesta que corresponde al mensaje se pospone para tiempo de ejecución. Es por esto que se necesita mantener las listas de respuestas asociadas a cada administrador y las cadenas de administradores de clases y superclases.

Cada vez que el intérprete realiza una instrucción de envío de mensaje a receptor no identificado, debe obtener la clase del receptor actual, obtener el objeto del adiminstrador de esa clase, buscar en la lista de respuestas asociadas con ese administrador. Si la respuesta no se encontro en la lista, la búsqueda continua de acuerdo a la política definida en la siguiente sección (IV.5.1). La búsqueda puede continuar por la cadena de administradores de superclases. En caso de no encontrarse la respuesta adecuada en ninguna lista, se genera una respuesta de "no entendimiento de mensaje".

El uso de variables de tipo ANYTHING proporciona cierta flexibilidad que puede ser muy útil, pero es preciso tener en cuenta que se tiene que pagar el precio de las búsquedas en las listas de respuestas. La realización de las busquedas se refleja en un tiempo de ejecución mayor. Es aconsejable que la mayoría de los mensajes sean enviados con primer tipo de instrucción.

IV.5.1 Busqueda de Respuestas

La forma de localizar la respuesta apropiada a un mensaje está intimamente ligada con el concepto de herencia. Todo objeto pertenece a una clase. Cada clase especifica las operaciones (responses) y campos de un objeto. Para definir una clase se puede hacer uso de clases ya existentes tomando de ellas operaciones o definición de campos de los objetos que definan. A las clases que se utilizan para definir otra clase se les llama superclases de la clase a definir y ésta es llamada subclase de aquellas. Se puede formar una red de herencias al considerar a una clase con todas sus superclases y las clases que se usaron para definir a las superclases (superclases de las superclases) etc.

Las búsquedas de respuestas deben realizarse tomando como punto de partida la clase del objeto receptor del mensaje. En caso de que la respuesta no este definida directamente en la clase se deberá realizar la búsqueda en las superclases de esta. Se presenta el problema de que existan varias superclases que contengan repuestas con nombres idénticos a la respuesta buscada. Es por esto que se debe definir la precedencia de superclases o sea el orden en que se iran revisando las superclases para encontrar la respuesta.

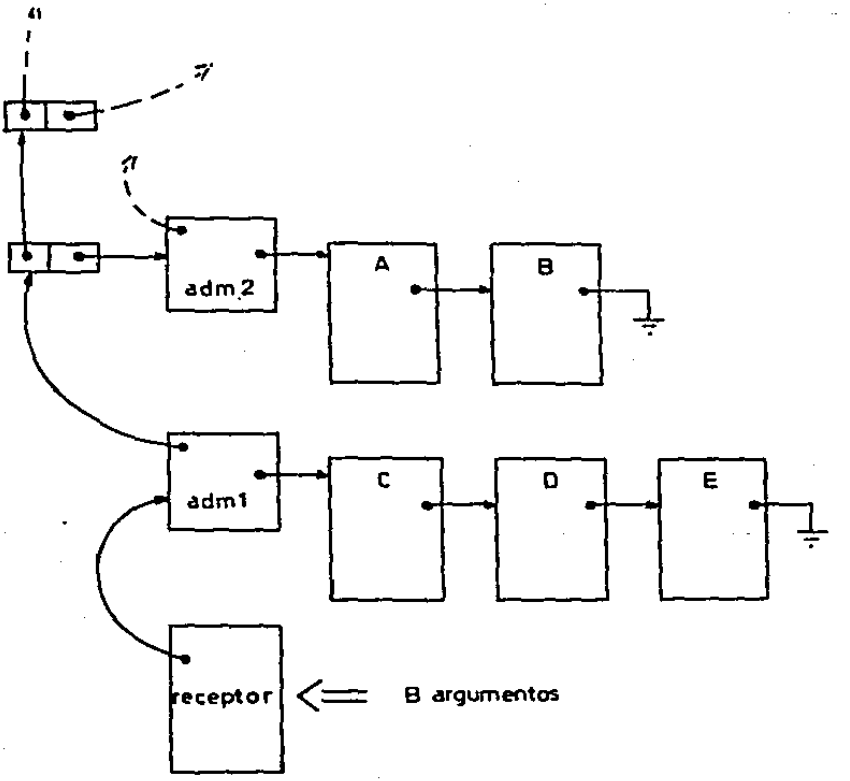

Figura IV.6. Busqueda de respuestas. En esta figura se observa un ejemplo de envio de un mensaje a un objeto. El envio del mensaje se denota por la flecha doble que apunta al objeto receptor. El selector del mensaje es B. La búsqueda de la respuesta comienza en la clase del receptor (cla 1). En la lista de respuestas asociadas a cla 1 no aparece la respuesta B (aparecen C, D y E). La búsqueda continua en la lista de superclases. La primera superclase es cla 2. finalmente se localiza la respuesta B en la lista de respuestas de cla 2.

El intérprete realizará la búsqueda de una respuesta en el siguiente orden:

La clase actual sera la clase del objeto receptor.

1o. Revisará la lista de respuestas asociadas a la clase actual. Si se encuentra se termina la búsqueda. En caso contrario:
2o. Obtendrá la lista de superclases de la clase actual. La lista estará ordenada de acuerdo a alguna regla que no define el intérprete.
3o. Se tomará uno a uno cada elemento de la lista para que sea la clase actual y se realiza el paso 1o. Esto se lleva a cabo mientras no se encuentre la respuesta y la lista no se agote.

Si se realiza el algoritmo sin encontrar la respuesta buscada el sistema responde con el letrero de "mensaje no entendido". En la fig. IV.6 aparece parte de la red de objetos donde se realiza una búsqueda de respuesta a un mensaje.

IV.6 Referencias

[Goldberg 83] Goldberg, A. & Robson D. "Smalltalk-80; The Language and Its Implementation" Addison-Wesley, Reading, Mass., 1983.

[Jimenez 86] Jimenez F. F. "Diseño e Implementación de un Sistema Orientado a Objetos" Tesis de M.C.C. U.N.A.M. Agosto 1986.

[Keedy 83] Keedy J.L. ,"An Instruction Set for Evaluating Expressions" IEEE Transactions on Computers Vol C-32 No.5 May 1983 pp. 476-478.

[Keedy 78] Keedy J.L., "On the Evaluation of Expressions Using Accumulators, Stacks and Store-to-Store Instructions" ACM Comput. Arch. News Vol 7 No.8, pp. 24-27.

[Levy 84] Levy H.M.,"Capability-Based Computer Systems" Digital Press 1984.

[Stefik 86] Stefik M. & Bobrow D."Object-Oriented Programming: Themes and Variations" The AI Magazine March 1986.

[Tanenbaum 78] Tanenbaum A.S., "Implications of Structured Programming for Machine Architecture" Communications ACM Vol 21 No. 3 pp 237-246.

CAPITULO V

Cargador Ligador

V.1 Introducción

Para ejecutar un programa en TM es necesario crear en la memoria de objetos un objeto de códigos para la máquina virtual. Tambien es necesario alojar en la memoria a todos los objetos a los que haga referencia el programa, incluyendo los objetos de código de los métodos o respuestas que se utilicen durante la ejecución del programa. Todas las referencias existentes entre los objetos que se alojen en memoria deben ser resueltas. Las tareas que se han mencionado se realizan durante los procesos de carga y ligado.

El ligado de subprogramas para formar programas compuestos es de gran valor en el desarrollo de software modular. El proceso de ligado puede llevarse a cabo en diferente orden con respecto a la codificación, la compilación la carga y la ejecución. Según (Presser 72) existen siete tiempos diferentes en los que puede efectuarse el ligado: 1) a tiempo de codificación del programa fuente; 2) despues de codificar pero antes de compilar; 3) a tiempo de compilación; 4) despues de compilar pero antes de cargar; 5) a tiempo de carga; 6) despues de cargar pero antes de ejecutar; o 7) a tiempo de ejecución.

El ligado que se efectúa durante o antes de la compilación implica una traducción separada para cada combinación diferente de subprogramas. Esto representa una desventaja seria. Si se lleva a cabo el ligado despues de la compilación se logra la capacidad de componer conjuntos de subprogramas sin necesidad de nuevas compilaciones de los conjuntos; sólo es necesario repetir el ligado para cada conjunto.

El ligado puede efectuarse a tiempo de carga (cargadores ligadores) como ocurre en gran cantidad de sistemas (por ejem. loader en IBM System/360). La popularidad de este tipo de ligadores es el resultado de su simplicidad, ya que la etapa de carga se presenta como el lugar natural para ligar los subprogramas.

Es posible ligar a tiempo de ejecución. Esta estrategia es conocida como segmentación (no confundir con el esquema de partición de memoria que lleva el mismo nombre). Cada segmento es una unidad lógica autocontenida de información relacionada que es definida y nombrada por el programador. Todas las referencias entre segmentos son alcanzadas a través de nombres simbólicos que se resuelven a tiempo de ejecución. La implantación más general de este esquema se encuentra en el sistema Honeywell 645 Multics (GE-645). El principale problema de

la segmentación consiste en tener altos costos de "overhead", aunque presenta más comodidad y flexibilidad que los esquemas anteriores. Para un manejo eficiente de este esquema es importante un diseño integrado de hardware y software.

En este trabajo se eligió seguir el esquema de cargador ligador ya que es una opción que permite tener flexibilidad y eficiencia además de ser sencilla de diseñar e implantar.

V.2 Cargador Ligador implantado

Ya se mencionaron en el capítulo II las funciones que debe efectuar el cargador ligador. Dichas funciones son:

a). Obtener espacio en la memoria para alojar los programas (alojamiento).

b). Resolver las referencias simbólicas que se presenten entre módulos del programa (ligado).

c). Ajustar todas las direcciones dentro del programa de acuerdo a la posición en la memoria donde se logró alojar al mismo (relocalización).

d). Depositar físicamente los códigos de máquina dentro de la memoria.

El programa cargador en esta implantación realiza las operaciones antes mencionadas de la siguiente manera:

a). Obtención de espacio en la memoria de objetos

El espacio necesario en la memoria es obtenido por el cargador ligador a través de la creación de objetos. La creación y acceso a los objetos se realiza mediante peticiones al administrador de la memoria.
Para los administradores se crean objetos del tipo primitivo record con campos para la lista de respuestas y variables de clase; las respuestas se representan en la memoria con objetos de tipo código, que tienen campos para almacenar apuntadores a literales y campos para códigos (Ver sección III.4).

Las literales son objetos constantes que están ligados a una respuesta determinada.

Algunas literales se crean a tiempo de carga de las respuestas en la memoria de objetos, otras son creadas a durante la ejecución de la respuesta. El compilador debe generar código que permita la creación de esas literales. Las literales que se crean a tiempo de ejecución son más complejas y su creación y asignación de valores puede involucrar la invocación de respuestas de uno o varios administradores.

Las literales que se crean a tiempo de carga son: números enteros y reales, valores booleanos, caracteres y cuerdas de caracteres.

Existen además otros objetos auxiliares que deben ser creados durante la fase de carga. Dichos objetos, con los "administradores", sirven para crear las estructuras de datos que describen las jerarquías de clases y superclases. Estas estructuras de datos son necesarias cuando se realizan búsquedas de respuestas a mensajes. Estos objetos auxiliares son del tipo "celda cons" y permiten formar las listas de superclases de cada administrador (Ver sección IV.5.1).

b). Solución de las referencias entre objetos.

En general los objetos contienen apuntadores a otros objetos. Los apuntadores sirven como índices de una tabla de objetos que a su vez contiene las direcciones reales de los objetos en la memoria.

La instalación de un objeto en la memoria implica que le sea asignado un apuntador que sirve para identificarlo durante su estancia en la memoria (las operaciones sobre el objeto se podrán realizar si se cuenta con el apuntador al objeto). Dicho apuntador probablemente será distinto cada vez que el objeto sea instalado.

La descripción de un objeto, que ha de ser depositado en la memoria, debe proporcionar un mecanismo que permita identificar a todos los objetos a los que hace referencia el objeto en cuestión. Dicho mecanismo no debe hacer uso de apuntadores a objetos, ya que un objeto puede ser asociado a diferentes apuntadores.

En la presente implantación del sistema se hace uso de un identificador único (IU) para cada objeto permanente (en realidad representación del objeto en un medio de almacenamiento permanente, fuera de la memoria de objetos) gracias al cual se pueden identificar los objetos independientemente del apuntador que les sea asignado al ser instalados en la memoria.

Los IU se obtienen a partir del nombre del objeto: las respuestas tienen un nombre que está dado por la pareja NOM - ADMINISTRADOR y los administradores por ADMINISTRADOR - ADMINISTRADOR. El elemento NOM puede ser el mismo para varias respuestas de diferentes administradores. Las respuestas que tiene el mismo NOM tienen selectores idénticos. En la presente implantación sólo existen dos tipos de objetos realmente permanentes: las respuestas y los administradores.

El compilador genera la descripción de los administradores y respuestas con un formato casi idéntico a la representación de los mismos en la memoria de objetos. La única diferencia relevante consiste en que los campos en los que deberán existir

apuntadores aparecen los IU de los objetos.

c). Relocalización

Esta tarea se realiza de manera automática ya que los códigos de la máquina virtual sólo contienen referencias relativas (no existen saltos absolutos). Se permite el uso de instrucciones "go to" únicamente dentro de la misma respuesta, y las mismas siempre se pueden expresar como saltos relativos.

d). Colocación de los códigos de máquina dentro de la memoria.

Esta operación se realiza gracias a llamados al administrador de la memoria de objetos que permiten guardar bytes en el cuerpo de un objeto de códigos. Son llamados muy similares a "AsigEnCa" cuya descripción aparece en la sección III.5.1.

La función del Cargador Ligador (CL) consiste en tomar las descripciones de los objetos que se especifiquen (la especificación la dará el usuario, mencionando los nombres de los objetos que desea sean instalados en la memoria), obtener los apuntadores a cada uno de ellos y reemplazar los IU por los apuntadores obtenidos.

Es importante señalar que la imagen virtual (conjunto de objetos que se han ido creando en un sistema orientado a objetos) es por definición permanente. Un objeto desaparecerá del medio ambiente hasta que deje de existir la última referencia al objeto. Idealmente la imagen virtual debe existir dentro de una memoria estática de objetos, que no se borre cada vez que el usuario sale del sistema. Un sistema con sólo un nivel de memoria (sin distinción entre memorias primaria y secundaria) es ideal para almacenar la imagen virtual. En dicho sistema desaparece la necesidad de utilizar el cargador ligador. El compilador generaría directamente los objetos de código correspondientes a los programas fuente.

V.3 Descripción Funcional

El cargador usado en esta implantación es del tipo cargador de ligado directo. En general este tipo de cargadores necesitan la siguiente información proporcionada por el compilador:

a) Longitud del segmento (en nuestro caso, del objeto de tipo código).

b) Lista de los símbolos dentro del segmento a los que se puede hacer referencia dentro de otros segmentos.

c) Lista de símbolos no definidos dentro del segmento.

d) La traducción a código de máquina del segmento.

Esta información aparece almacenada en archivos creados por el compilador (archivos objeto). El formato de los archivos se describe en la sección V.5.

Como el cargador de ligado directo puede encontrar referencias externas dentro de un objeto que no pueden ser evaluadas hasta que un nuevo objeto sea procesado, se requiere la realización de dos pasadas. Las dos pasadas funcionan de forma similar a las que lleva a cabo un ensamblador.

V.3.1 Primera Pasada

La función principal de la primera pasada consiste en crear los objetos necesarios en la memoria (que representaran a los administradores, las respuestas, las literales y los objetos auxiliares), además de depositar dentro de los mismos los contenidos especificados por los archivos generados por el compilador, dejando vacíos únicamente aquellos campos donde existan referencias no resueltas. La pasada inicial tambien crea dos tablas llamadas Simb_Apu y Referencias. La tabla Simb_Apu relaciona los símbolos externos con sus apuntadores; sus campos son:

- IU del objeto que puede ser referenciado externamente (o nombre externo)

- Apuntador asignado al objeto

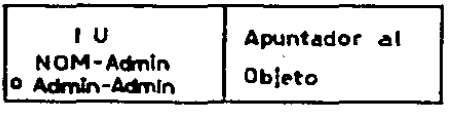

Figura V.1 Entrada de la tabla Simb_Apu

La tabla Simb_Apu se utiliza repetidamente durante la segunda pasada para localizar el apuntador asignado a cada objeto. Para hacer más eficientes las búsquedas en la tabla, ésta se implantó como un árbol binario ordenado por el campo IU.

La tabla "Referencias" contiene los nombres de las

referencias no resueltas de todos los objetos con el siguientes campos:

- Referencia a resolver
- Objeto donde se localiza dicha referencia
- Campo dentro del objeto donde se localiza la referencia.

referencia a resolver (I U)	Apun. al objeto donde se localiza	Posición en el objeto

Figura V.2 Entrada de la Tabla Referencias

V.3.2 Segunda Pasada

Esta pasada consiste simplemente en el barrido secuencial de la tabla "Referencias". Cada una de las entradas de la tabla, que representa una referencia no resuelta, se va considerando y se busca en la tabla "Simb_Apu" si la referencia tiene asignado un apuntador. En caso afirmativo, se deposita el apuntador en el campo del objeto especificado (por el segundo y tercer campos de la entrada en la tabla "Referencias"). En caso contrario se considera que la referencia no se pudo resolver y se despliega un mensaje de advertencia.

V.4 Literales

Las literales que se crean a tiempo de carga pertenecen a alguna de las siguientes categorías:

i) Valores Booleanos
ii) Enteros (chicos y grandes)
iii) Reales (precisión sencilla y doble)
iv) Caracteres y cuerdas de caracteres

En el caso de literales más complejas, es necesario que el compilador genere el código necesario para crearlas. El código puede ser hecho de manera que la literal sea creada cada vez que la respuesta donde aparece sea invocada. Un código más eficiente deberá revisar si el campo donde se debe alojar

referencias no resueltas de todos los objetos con el siguientes campos:

- Referencia a resolver
- Objeto donde se localiza dicha referencia
- Campo dentro del objeto donde se localiza la referencia.

referencia a resolver (I U)	Apun. al objeto donde se localiza	Posición en el objeto

Figura V.2 Entrada de la Tabla Referencias

V.3.2 Segunda Pasada

Esta pasada consiste simplemente en el barrido secuencial de la tabla "Referencias". Cada una de las entradas de la tabla, que representa una referencia no resuelta, se va considerando y se busca en la tabla "Simb_Apu" si la referencia tiene asignado un apuntador. En caso afirmativo, se deposita el apuntador en el campo del objeto especificado (por el segundo y tercer campos de la entrada en la tabla "Referencias"). En caso contrario se considera que la referencia no se pudo resolver y se despliega un mensaje de advertencia.

V.4 Literales

Las literales que se crean a tiempo de carga pertenecen a alguna de las siguientes categorías:

i) Valores Booleanos
ii) Enteros (chicos y grandes)
iii) Reales (precisión sencilla y doble)
iv) Caracteres y cuerdas de caracteres

En el caso de literales más complejas, es necesario que el compilador genere el código necesario para crearlas. El código puede ser hecho de manera que la literal sea creada cada vez que la respuesta donde aparece sea invocada. Un código más eficiente deberá revisar si el campo donde se debe alojar

la literal ya esta ocupado (la literal ya ha sido creada) en cuyo caso evitará la creación repetida de la misma literal.

V.5 Formato de los Archivos

Existen dos clase de archivos que utiliza el CL. El primer tipo corresponde a los archivos donde se almacena la descripción del administrador y sus respuestas y el segundo tipo es donde se localiza la descripción del programa usuario o expresión TM a ser ejecutada.

V.5.1 Archivos para Administradores y Respuestas

El primer tipo de archivo contiene los siguientes campos:

a) Identificador Unico (IU) del administrador.

b) Número de superclases del administrador (NS).

c) Secuencia con IU's de las superclases. Debe haber NS IU's.

d) Número de variables globales de la clase.

e) Número de Respuestas del administrador (NR). Solo se contarán las respuestas que se añadan o substituyan (no se contarán las respuestas que se heredan de las superclases).

f) Secuencia de contenido de cada respuesta. Debe habe NR descripciones del contenido de respuestas.

Cada descripción del contenido de una respuesta contará con los siguientes campos:

f1) IU de la respuesta.

f2) Longitud de la respuesta (número de campos de 4 bytes necesarios para contener a la respuesta).

f3) Número de campos en el marco de literales de la respuesta (NL).

f4) Secuencia de descripciones de las literales. Deberá haber NL descripciones de literales.

f5) Número de campos del marco de temporales de la respuesta.

f6) Número de códigos que contiene la respuesta (NC).

f7) Secuencia de códigos de la respuesta. Existirán NC códigos.

Para describir una literal se sigue la siguiente convención en

los campos. Si la literal es:

-Identificador Unico:

f4.0) Tipo = 1
f4.1) Identificador del nombre del objeto
f4.2) Identificador de administrador

El compilador debe asignar a cada nombre de rutina de respuesta o administrador un identificador. Para convertirlo en único se toma la pareja identificador del nombre del objeto - identificador de clase. Esto permite distinguir entre los selectores iguales que correspondan a administradores diferentes.

-Número Entero, Valor Booleano o Tipo Primitivo de Objeto a Crear

f4.0) Tipo = 2
f4.1) Valor Entero (Si es booleano false = 0, true = 1)

-Cuerda de caracteres (o caracter)

f4.0) Tipo = 3
f4.1) Longitud de la cuerda NCH(Número de caracteres de la cuerda).
f4.2) Secuencia de caracteres de la cuerda en ASCII en decimal (existirán NCH).

-Número Real

f4.0) Tipo = 4
f4.1) Número real en formato libre.

V.5.2 Archivo para Programa Usuario

El segundo tipo de archivo se utiliza para describir al programa usuario (llamado tambien Pseudorespuesta Interactiva).

Cada clase de objetos cuenta con una serie de rutinas o respuestas que efectúan alguna operación sobre los objetos de la clase. Es necesario mandar un mensaje a un objeto de la clase para que se invoque alguna de estas rutinas. Las rutinas de respuesta a su vez mandan mensajes a diversos objetos con lo cual se invoca una serie de rutinas de respuesta. El nivel más externo de envío de mensajes está constituido por el programa usuario. Este programa, parecido sintacticamente a una respuesta, no se invoca al ser enviado un mensaje, sino que se manda ejecutar cada vez que el programador da la orden al sistema. El programa usuario no está encapsulado dentro de la definición de un administrador y su función es similar a la de un programa principal en lenguajes no orientados a objetos.

El archivo que describe al programa usuario es similar a los descritos anteriormente. Sus campos son los mismos que deben

aparecer en la descripción de una respuesta cualquiera dentro de un administrador, con la excepción de que el primer campo no debe aparecer. El programa usuario no tiene asociado un identificador único.

V.6 Referencias Generales

[Barron 78] Barron D. W. "Assemblers and Loaders" McDonald and Jane's Pub. Third Ed. London 1978.

[Donovan 72] Donovan, J.J."Systems Programming" Computer Science Series McGraw-Hill 1972 Capítulo 5 "Loaders".

[Presser 72] Presser, L. & White, J. R. "Linkers and Loaders" Computing Surveys, Vol. 4 No. 3 Sept. 1972.

[Ullman 76] Ullman, J. D. "Fundamental Concepts of Programming Systems" Addison-Wesley Chapter 5 "Loaders and Link Editors".

CAPITULO VI

Programas de Ejemplo para la Máquina Virtual

VI.1 Introducción

En los capítulos anteriores se ha explicado la arquitectura y funcionamiento de la máquina virtual. Este capítulo se refiere concretamente a como se lleva a cabo el objetivo por el que se diseño y construyó dicha máquina.

La máquina virtual de TM debe ejecutar programas en lenguaje TM, en realidad, códigos generados por el compilador que son equivalentes a los programas en TM.

En este capítulo se ven ejemplos de programas en TM con sus equivalentes en código para la máquina virtual. Estos programas corren en el sistema implementado. Aunque la sintaxis de TM es suficientemente clara como para permitir el entendimiento de las acciones descritas en el código, es recomendable que el lector conozca las reglas del lenguaje antes de emprender la lectura de este capítulo. Para ello se sugiere la lectura de [Gerzso 83].

Es necesario tener en cuenta que los ejemplos que aparecen en esta sección fueron traducidos a mano de lenguaje TM a códigos de la máquina virtual. Es probable que los códigos que se obtengan de un compilador sean ligeramente diferentes (especialmente en cuanto al orden de las operaciones) pero las acciones que se lleven a cabo deberán ser las mismas que aparecen aquí.

VI.2 Programa de árboles

El programa en TM que a continuación se analiza sigue el mismo algoritmo que un programa en Pascal que aparece en la página 203 de la referencia [Wirth 76]. Se trata de un programa que nos permite realizar algunas operaciones sobre árboles binarios. Es un ejemplo sencillo (el administrador de árboles solo contará con cuatro respuestas) pero a la vez interesante pues presenta recursividad. En lenguaje TM el programa es el siguiente:

```
{
administrator   .tree
public
      to_itself
            create =>
            <-    instance;
      end_it
```

```
      to_instance
            NullTree?  =>
            <-  <es  el  árbol  nulo?>.boolean;

            search  <key  value>.fix  =>
            <-  instance;

            printtree  <level  number>.fix  =>
            <-;
      end_inst
end_pub

private
      instance  k  :  fix
                l : tree
                r  :  tree  fields

      to_itself
            create  =>  /* Crea  un  árbol  inicial  nulo  */
                  declare  aux  :  tree  temporary

                  aux  :=  .tm  <=  create  'tree';
                  aux  !  k  :=  nil;  /* el  arbol  nulo  se  distinguirá  ya
                                       que  su  llave  es  nula  */
                  aux  !  l  :=  nil;
                  aux  !  r  :=  nil;
                  <- aux;
            end=>
      end_it

      to_instance

            NullTree?  =>  /*  revisa  si  el  receptor  es  el  arbol
                              nulo */
                  <-  IsNil?(  instance  !  k  )
            end=>

            search  k  =>  /*  falta  actualizar  */
                  declare  aux  :  tree  temporary
                  .if  <=  (  instance  <=  NullTree?  )  then
                  {
                        aux  !  k  :=  k;
                        aux  !  l  :=  .tree  <=  create;
                        aux  !  r  :=  .tree  <=  create;
                        <- aux;
                  }
                  else
                  {
                        .if  <=  (  instance  !  k  <=  it  k  )  then
                        {
                              instance!  l  :=  instance  !  l  <=  search  k;
                              <-  instance;
                        }
                        else
                        {
```

```
                        instance; r := instance ; r <= search k;
                        <-  instance;
                    ]
                ]
            end=>


            printtree j =>
                declare i : fix temporary
                .if <= ( instance <= NullTree? <= not ) then
                [ instance ; i <= printtree j <= + 1;
                    .do <= for [i := i;] [i <= incto j;]
                        ['  ' <= write;];
                    instance ; k <= write;
                    instance ; r <= printree j <= +i;
                ]
            end=>
    end_inst
end_priv


declare k : fix
        t : tree temporary

t := nil;
k := 99;

 .repeat <= while
                [ k <= eq 0
                  <= not ]
    [ 'k=' <= write;
      k := k <= readin;
      t := t <= search k;
    ];
t <= printtree i;
<-;
```

Existen dos modalidades en el tipo de código que produce el compilador. La primera se presenta cuando a tiempo de compilación se pueden identificar las respuestas que contestaran a los mensajes que se envien dentro del programa (la búsqueda de respuestas se realizó a tiempo de compilación). La segunda se presenta cuando no es posible identificar las respuestas durante la compilación debido a que se desconoce a que administrador pertenece el objeto receptor del mensaje (la búsqueda de respuestas tiene que efectúarse a tiempo de ejecución).

VI.2.1 Administrador tree

El código correspondiente al programa TREE para la primera modalidad (respuestas ya identificadas a tiempo de compilación) esta contenido en cuatro objetos de tipo código. Los objetos son:

1) Respuesta CREATE. Esta respuesta tiene la función de crear objetos de tipo Tree, colocando valores iniciales a los campos del objeto.

Marco de Literales

CAMPO	CONTENIDO
1	Apuntador a clase del objeto a crear. Clase TREE.
2	Entero Chico 3
3	Entero Chico 4

Se necesita el siguiente marco de temporales para activar esta respuesta:

CAMPO	CONTENIDO
0	Var Temporal tipo TREE

Los códigos son:

	CODIGO	SIGNIFICADO
1)	2	Push objeto de marco de literales.
2)	1	Campo 1 de marco, apuntador a adminis. tree.
3)	35	Crea Objeto (tipo de objeto en tope stack).
4)	5	Longitud del objeto deseado = 5.
5)	12	Pop a marco de temporales.
6)	0	Campo 0 de marco de temporales.
7)	1	Push objeto de marco de temporales.
8)	0	Campo 0 de marco de temporales.
9)	7	Push entero 2.
10)	8	Push NIL.
11)	24	Activa primitiva.
12)	4	Primitiva Guarda. Coloca NIL en campo 2 de objeto creado. (Objeto tree nulo tiene apuntador NIL en campo 0).
13)	1	Push objeto de marco de temporales.
14)	0	Campo 0 de marco de temporales.
15)	2	Push objeto de marco de literales.
16)	2	Campo 2 de literales, entero 3.
17)	8	Push NIL.
18)	24	Activa primitiva.
19)	4	Primitiva Guarda. Coloca NIL en campo 3, 'l'.
20)	1	Push objeto de marco de temporales.
21)	0	Campo 0 de marco de temporales.
22)	2	Push objeto de marco de literales.
23)	3	Campo 3 de literales, entero 4.
24)	8	Push de NIL.
25)	24	Activa primitiva.
26)	4	Primitiva Guarda. Coloca NIL en campo 4, 'r'.
27)	1	Push objeto de marco de temporales.
28)	0	Campo 0 de marco de temporales.
29)	16	Return de tope del Stack.

Los códigos corresponden a la implementación de las

respuestas que aparece en la sección "private" de un programa en TM. A continuación se muestra la respuesta en TM en la que se indica que códigos corresponden a que instrucciones. Los números que aparecen entre paréntesis coresponden con la numeración de los códigos de la respuesta CREATE en la lista de códigos anterior.

```
create =>
      declare aux : tree temporary
                  /* Campo 0 en Marco de Temporales */
      aux :=                                          (5),(6)
            .tm <= create 'tree';                (1),(2),(3),(4)
      aux ! k := nil;                                  (7)..(14)
      aux ! l := nil;                                 (15)..(19)
      aux ! r := nil;                                 (20)..(26)
      <- aux;                                         (27)..(29)
end=>
```

Los valores 3 y 4 que aparecen en el marco de literales sirven para indicar los desplazamientos que para los campos "l" y "r" del objeto creado. Un objeto creado de tipo TREE tiene el siguiente contenido:

campo (de 32 bits)	contenido
0	5 - longitud del objeto.
1	apuntador al administrador TREE.
2	Nil - campo k del objeto.
3	Nil - campo l del objeto.
4	Nil - campo r del objeto.

Se observa que el campo k, que se utiliza para almacenar la llave del nodo, tiene como valor inicial Nil. Esto se debe a que es necesario distinguir entre los árboles vacíos o nulos (aquellos que tienen el campo k con Nil) del los no vacíos (con el campo con un objeto de tipo entero). El distinguirlo con el valor de Nil en el campo de la llave es una decisión del programador.

Es recomendable que el programador de TM defina siempre un elemento "nulo" distinguible dentro de los objetos de la clase que esta^ definiendo, un elemento desde el cual partir para formar estructuras más complejas. Esto es particularmente necesario para los objetos mutables como listas o árboles cuyo número de nodos no es fijo.

Se podría pensar que el papel de elemento nulo para todas las clases es Nil, pero se debe recordar que el único mensaje que se le puede mandar a Nil es el predicado IsNil?, por lo que no se le podrían mandar los mensajes definidos en la clase. Otra desventaja consiste en que Nil pertenecería a todas las clases presentandose ambiguedades respecto a la determinación de la respuesta que corresponde al mensaje.

Es necesario hacer la aclaración de que los objetos en TM al ser creados tienen por "default" en todos sus campos el valor

Nil, por lo que las instrucciones de asignación de Nil a los campos son innecesarias y sólo se realizan por claridad y como una costumbre aconsejable de programación.

ii) Respuesta NullTree?. Esta respuesta permite distinguir al elemento "nulo" dentro de los objetos de la clase Tree.

Marco de Literales - Sin espacio para literales

Marco de Temporales - No utiliza ninguna variable temporal ni parametros, sólo necesita al receptor.

Los códigos son:

	CODIGO	SIGNIFICADO
1)	0	Push de campo del receptor.
2)	2	Campo 2 del receptor (campo "k").
3)	24	Manda Primitiva.
4)	5	Primitiva ISNIL?
5)	16	Return Tope del Stack.

La correspondencia entre códigos e instrucciones aparece a continuación:

```
NullTree?   =>  /* revisa si el receptor es el arbol
                   nulo */
        <-                                          ( 5 )
            IsNil?                              (3),  (4)
              ( instance ; k )                  (1),  (2)
        end=>
```

La única acción que se realiza consiste en revisar si el campo k del árbol contienen el valor de Nil. En caso afirmativo, se trata de un árbol nulo. El resultado de la revisión es el resultado de NullTree?.

iii) Respuesta SEARCH. Con esta respuesta se efectua una búsqueda de una llave en el árbol receptor. La llave se almacena en el lugar adecuado para mantener el orden.

Marco de Literales

CAMPO	CONTENIDO
1	Apuntador a Respuesta CREA-Tree.
2	Apuntador a Respuesta SEARCH-Tree.
3	Apuntador a Resp. NullTree?-Tree.

Se necesita el siguiente marco de temporales para activar esta respuesta:

CAMPO	CONTENIDO
0	Parámetro de llave a almacenar

Los códigos son:

	CODIGO	SIGNIFICADO
1)	9	Push Receptor.
2)	22	Manda Mensaje. Rec. y Args en Stack.
3)	3	Respuesta NullTree? (literal 3)
4)	0	Sin Argumentos, sblo Receptor.
5)	33	Salto si Falso
6)	17	Desplazamiento.
7)	1	Push objeto de marco de temporales.
8)	0	llave a almacenar (temporal 0).
9)	11	Pop a Campo del Receptor.
10)	2	Campo k (campo 2 del receptor).
11)	22	Manda Mensaje. Rec. y Args en Stack.
12)	1	Respuesta Crea (literal 1)
13)	-1	Sin Receptor ni Parhmetros.
14)	11	Pop a Campo del Receptor.
15)	3	Campo l (campo 3 del receptor).
16)	22	Manda Mensaje. Rec. y Args en Stack.
17)	1	Respuesta Crea (literal 1).
18)	-1	Sin Receptor ni Parhmetros.
19)	11	Pop a Campo del Receptor.
20)	4	Campo r (campo 4 del receptor).
21)	17	Return del Receptor.
22)	26	Salto Incondicionado.
23)	30	Longitud del salto.
24)	0	Push Campo del Receptor.
25)	2	Campo k (campo 2 del receptor).
26)	1	Push objeto de marco de temporales.
27)	0	Llave a buscar y guardar.
28)	24	Activa Primitiva.
29)	12	Primitiva Menor o Igual de enteros.
30)	33	Salto si Falso.
31)	12	Longitud del salto.
32)	0	Push Campo del Receptor.
33)	3	Campo l del Receptor.
34)	1	Push objeto de marco de temporales.
35)	0	Llave a buscar y almacenar.
36)	22	Manda Mensaje. Rec. y Args en Stack.
37)	2	Respuesta Search (literal 2).
38)	1	Con un Parhmetro.
39)	11	Pop a Campo del Receptor.
40)	3	Campo l (campo 3 del receptor)
41)	17	Return del Receptor.
42)	26	Salto Incondicional
43)	10	Longitud del salto.
44)	0	Push de Campo del Receptor.
45)	1	Campo r (Campo 4 del Receptor).
46)	1	Push objeto de marco de temporales.
47)	0	Llave a buscar y almacenar (campo 0)
48)	22	Manda Mensaje. Rec. y Args en Stack.
49)	2	Respuesta Search (literal 2).
50)	1	Con 1 parhmetro.
51)	11	Pop a Campo del Receptor.
52)	4	Campo r (Campo 4 del Receptor).

```
53)        17          Return Receptor.
54)        16          Return Tope del Stack.
```

Lo que sigue es la respuesta SEARCH en TH mostrando que códigos corresponden a que instrucciones (los índices de los códigos aparecen entre parentesis)

```
search  k  =>
      /* Parametro k ocupa campo 0 de marco tempo. */
      .if <= (
         NullTree?(instance)                                    (1)..(4)
                                )then                            (5),(6)
      (
            instance ; k :=                                     (9),(10)
                                 k;                             (7),(8)
            instance ; l :=                                    (14),(15)
                               .tree    <=    create;          (11)..(13)
            instance ; r :=                                    (19),(20)
                               .tree    <=    create;          (17)..(18)
            <- instance;                                          (21)
      )
      else                                                     (22),(23)
      ( aux :=                                                  (9),(10)
                  .tree <= create;                           (6),(7),(8)
            aux ; k :=                (13),(14),(15),(16),(17)
                           k;                                  (11),(12)
            <- aux;                                  (18),(19),(20)
      )
      else                                                     (21),(22)
      (
            .if <= (
                  instance ; k                               (24),(25)
                        <= lt                                (28),(29)
                                 k                           (26),(27)
                                      )    then               (30),(31)
            (
                  instance; l :=                             (39),(40)
                        instance ; l                         (32),(33)
                              <= search                (36),(37),(38)
                                    k;                       (34),(35)
                  <- instance;                                  (41)
            )
            else                                                (42),(43)
            (
                  instance; r :=                             (51),(52)
                        instance ; r                         (44),(45)
                              <= search                (48),(49),(50)
                                    k;                       (46),(47)
                  <- instance;                                  (53)
            )
      )
end=>                                                            (54)
```

iv) Respuesta PRINTTREE. Esta respuesta realiza el despliegue del contenido del objeto tipo tree receptor.

Marco de Literales

CAMPO	CONTENIDO
1	Apuntador a Respuesta PRINTREE-Tree.
2	String " "
3	Apuntador a Respuesta NullTree?-Tree.

Se necesita el siguiente marco de temporales para activar esta respuesta:

CAMPO	CONTENIDO
0	Parámetro de nivel del árbol
1	Variable de índice de un ciclo.

Los códigos son:

	CODIGO	SIGNIFICADO
1)	9	Push del receptor.
2)	22	Envía mensaje, receptor en stack.
3)	3	Respuesta NullTree? (literal 3).
4)	0	Sin Argumentos (solo receptor).
5)	24	Envía mensaje a primitiva.
6)	8	Primitiva booleana Not.
7)	33	Salto si falso.
8)	48	Longitud del salto.
9)	0	Push de campo del receptor.
10)	3	Campo 3 del receptor (campo 1).
11)	1	Push de objeto en marco de temporales.
12)	0	Campo 0 de temporales (nivel del árbol).
13)	6	Push de un 1 en el stack.
14)	24	Activa primitiva.
15)	0	Primitiva suma de enteros.
16)	22	Manda mensaje.
17)	1	Respuesta PrintTree (literal 1).
18)	1	Mensaje con un argumento.
19)	6	Push de un 1.
20)	12	Pop y almacena en marco de temporales.
21)	1	Campo 1 de marco de temporales.
22)	1	Push de objeto en marco de temporales.
23)	1	Campo 1 de marco de temporales.
24)	1	Push de objeto en marco de temporales.
25)	0	Campo 0 de marco de temporales.
26)	24	Activa primitiva.
27)	6	Primitiva de menor que.
28)	33	Salto si falso.
29)	13	Longitud del salto.
30)	2	Push de objeto en marco de literales.
31)	2	Campo 2 de marco de literales.
32)	24	Activa primitiva.
33)	11	Primitiva de escritura de string.
34)	1	Push de objeto en marco de temporales.
35)	1	Campo 1 de temporales.

```
36)        6          Push de un 1.
37)        24         Activa primitiva.
38)        0          Suma primitiva.
39)        12         Pop y almacena en marco de temporales.
40)        1          Campo 1 de temporales.
41)        26         Salto incondicional.
42)       -21         Longitud del salto.
43)        0          Push del campo del receptor.
44)        2          Campo 2 del receptor (campo K)
45)        24         Activa primitiva.
46)        7          Primitiva de escritura de enteros.
47)        0          Push de campo del receptor.
48)        4          Campo 4 del receptor (campo r).
49)        1          Push de temporal.
50)        0          Campo 0 del temporal.
51)        6          Push de un 1.
52)        24         Activa primitiva.
53)        0          Suma primitiva.
54)        22         Manda mensaje.
55)        1          Respuesta PrintTree (literal 1).
56)        1          Mensaje con un argumento.
57)        16         Return tope del stack.
```

La respuesta en TM con las correspondencias entre codigos e instrucciones es:

```
printtree  j  =>
        /* j ocupa el campo 0 de marco de temporales */
     declare  i  :  fix  temporary
        /* i ocupa el campo 1 de marco de temporales */
     .if <= (
          instance                                          (1)
                <= NullTree?                          (2),(3),(4)
                <= not                                   (5),(6)
                                ) then                   (7),(8)
     ( instance : l                                     (9),(10)
                <= printtree                     (16),(17),(18)
                     j    <=    +    1;    (11),(12),(13),(14),(15)
          .do <= for                                    (22)..(28)
                                                        (34)..(40)
                (i := 1;)                          (19),(20),(21)
                (i <= incto j;)                          (41),(42)
                (' '                                     (30),(31)
                     <= write;)                          (32),(33)
          instance : K                                   (43),(44)
                <= writeln;                              (45),(46)
          instance : r                                   (47),(48)
                <= printtree .                   (54),(55),(56)
                     j    <=    +1;    (49),(50),(51),(52),(53)
     )
 end=>                                                      (57)
```

Esta respuesta recursiva escribe el contenido del arbol realizando un recorrido infijo. La instruccion de iteracion que contiene sirve para desplegar un numero de "blancos" antes de la

```
36)        6          Push de un 1.
37)        24         Activa primitiva.
38)        0          Suma primitiva.
39)        12         Pop y almacena en marco de temporales.
40)        1          Campo 1 de temporales.
41)        26         Salto incondicional.
42)       -21         Longitud del salto.
43)        0          Push del campo del receptor.
44)        2          Campo 2 del receptor (campo k)
45)        24         Activa primitiva.
46)        7          Primitiva de escritura de enteros.
47)        0          Push de campo del receptor.
48)        4          Campo 4 del receptor (campo r).
49)        1          Push de temporal.
50)        0          Campo 0 del temporal.
51)        6          Push de un 1.
52)        24         Activa primitiva.
53)        0          Suma primitiva.
54)        22         Manda mensaje.
55)        1          Respuesta PrintTree (literal 1).
56)        1          Mensaje con un argumento.
57)        16         Return tope del stack.
```

La respuesta en TM con las correspondencias entre códigos e instrucciones es:

```
printtree  j  =>
        /* j ocupa el campo 0 de marco de temporales */
    declare  i  :  fix  temporary
        /* i ocupa el campo 1 de marco de temporales */
    .if <= (
        instance                                          (1)
                <= NullTree?                          (2),(3),(4)
                <= not                                   (5),(6)
                        ) then                           (7),(8)
    [ instance : i                                      (9),(10)
                <= printtree                      (16),(17),(18)
                    j    <=    +    1;    (11),(12),(13),(14),(15)
        .do <= for                                      (22)..(28)
                                                        (34)..(40)
                [i := i;]                         (19),(20),(21)
                [i <= incto j;]                      (41),(42)
                [' '                                    (30),(31)
                    <= write;];                         (32),(33)
                instance : k                            (43),(44)
                    <= writeln;                         (45),(46)
                instance : r                            (47),(48)
                    <= printtree .                (54),(55),(56)
                        j    <=    +1;    (49),(50),(51),(52),(53)
        ]
    end=>                                                (57)
```

Esta respuesta recursiva escribe el contenido del árbol realizando un recorrido infijo. La instrucción de iteración que contiene sirve para desplegar un número de "blancos" antes de la

llave del nodo. Dicho número es igual al nivel del nodo cuyo contenido se va a mostrar.

VI.2.2 Programa Usuario

Un programa sencillo en TM que utiliza al administrador Tree es el siguiente:

```
declare  k  :  fix
         t  :  tree  temporary

      t  ::  .tree  <=  create;
      k := 99;
      .repeat <= while ( k <= eq 0 <= not )
      (
            " Key= " <= write;
            k  ::  .fix  <=  readln;
            t  ::  t  <=  search  k;
      );
      t  <=  printtree  1;
<=;
```

Este programa crea un árbol que va llenando con datos que lee. El llenado lo realiza mediante envíos de mensajes "search". Se termina de llenar el árbol al leerse el dato entero 0. Finalmente se despliega el contenido del árbol.

Las literales, temporales y códigos del programa anterior son los sigs.:

Marco de Literales

CAMPO	CONTENIDO
1	Apuntador a Respuesta search-tree.
2	Entero Chico 99
3	Apuntador a Respuesta printtree-tree.
4	String " Key : "
5	Apuntador a Respuesta create-tree.

Se necesita el siguiente marco de temporales para activar esta respuesta:

CAMPO	CONTENIDO
0	Variable Temporal k tipo FIX
1	Variable Temporal t tipo TREE

Códigos:

	CODIGO	SIGNIFICADO
1)	22	Envía mensaje.

```
2)        5          Respuesta create (literal 5).
3)        -1         Mensaje sin argumentos ni instancia receptor.
4)        12         Pop y almacena en marco de temporales.
5)        1          Variable t (temporal 1).
6)        2          Push de objeto en marco de Literales.
7)        2          Entero 99 (literal 2).
8)        12         Pop y almacena en marco de temporales.
9)        2          Variable K (temporal 0).
10)       1          Push objeto de marco de temporales.
11)       2          Variable K (temporal 0).
12)       5          Push entero 0.
13)       24         Activa primitiva.
14)       10         Primitiva de igualdad de enteros.
15)       24         Activa primitiva.
16)       8          Primitiva booleana Not.
17)       33         Salto si falso.
18)       19         longitud del salto.
19)       2          Push objeto marco de literales.
20)       4          String " Key : " (literal 4)
21)       24         Activa primitiva.
22)       11         Primitiva de escritura de string.
23)       24         Activa primitiva.
24)       9          primitiva de lectura de entero (FIX)
25)       12         Pop y almacena en marco de temporales.
26)       2          Variable K (temporal 0).
27)       1          Push objeto de marco de temporales.
28)       1          Variable t (temporal 1).
29)       1          Push objeto de marco de temporales.
30)       2          Variable K (temporal 0).
31)       22         Envia mensaje.
32)       1          Respuesta search (literal 1).
33)       1          Respuesta con un argumento (K).
34)       12         Pop y almacena en marco de temporales.
35)       1          Variable t (temporal 1)
36)       26         Salto incondicionado.
37)       -28        Longitud del salto.
38)       1          Push de objeto en marco de temporales.
39)       1          Variable t (temporal 1).
40)       6          Push entero 1.
41)       22         Envia mensaje.
42)       3          Respuesta printtree (literal 3).
43)       1          Respuesta con un argumento (1)
44)       34         Termina interpretacion.
```

La correspondencia entre codigos e instrucciones es:

```
declare K : fix
                /* Campo 0 de temporales */
        t : tree temporary
                /* Campo 1 de temporales */

    t :=                                                    (4),(5)
        .tree <= create;                                  (1)..(3)
    K :=                                                    (6),(9)
```

```
        99;                                                    (6),(7)
      .repeat <= while                                        (17).(18)
                        { k                                   (10),(11)
                            <=.eq 0                           (12)..(14)
                            <= not }                          (15),(16)
      [
            " Key= " <= write;                                (19)..(22)
            k := .fix <= readln;                              (23)..(26)
            t :=                                              (34),(35)
                  t                                           (27),(28)
                        <= search k;                          (29)..(33)
      ];                                                      (36),(37)
      t                                                       (38),(39)
        <= printtree t;                                       (40)..(43)
<=;                                                           (44)
```

Para finalizar este ejemplo se presenta una muestra de los resultados del programa. Primero aparece la lectura de los valores enteros que se desea almacenar en el árbol, terminando con un cero. Despues aparece el despliegue del contenido del árbol tal como se implantó en la rutina printtree.

```
 Key = 34
 Key = 23
 Key = 123
 Key = 25
 Key = 9
 Key = 12
 Key = 467
 Key = -356
 Key = 65
 Key = 0


      467
   123
      65
34
      25
   23    12
      9
         0
           -356
```

VI.3 Programa de listas

El ejemplo que aparece a continuación se refiere al manejo de estructuras de datos de tipo lista.

Este ejemplo es algo mas extenso que el anterior debido a que presenta las siguientes características:

-Contiene la declaración de un administrador de llaves "Key" que define los valores a almacenar en las listas (en el ejemplo

anterior se utilizaron datos tipo primitivo "fix").

-El administrador de llaves "key" tiene una superclase "keyi" de la cual hereda varias de las respuestas.

-Las respuestas del administrador "list_of_keys" contienen variables de tipo "anything" con el fin de que se puedan usar distintos tipos de contenidos de las listas.

-No se maneja directamente el tipo primitivo "list" sino que se define uno nuevo llamado "list_of_keys" que contiene un campo tipo "list". Es interesante notar que se pudo haber usado directamente el tipo "list" o bien se pudo declarar al nuevo tipo "list_of_keys" como subclase de "list". Cualquiera de las opciones es igualmente valida y sencilla de instrumentar.

Este ejemplo se presenta de manera similar al anterior, mostrando primero el codigo en TM, luego los codigos de la maquina virtual y finalmente las correspondencias entre ambos tipos de codigo.

VI.3.1 Administrador key

Las listas de este ejemplo continen datos que no pertenecen a ningun tipo primitivo. Los datos que se usaran son administrados por "key". El codigo del administrador key es el siguiente.

```
!
administrator .key superclass .keyi
public
     to_itself
          create <key value>.fix <count value>.fix =>
          <- instance;

          readkey =>
          <- instance;

     end_it

     to_instance
          eq <K>.fix =>
          <- <Es contenido de key : k?>.boolean;

          setkey <K>.fix =>
          <- instance;

          setcount <K>.fix =>
          <- instance;

          + <c>.fix =>
          <- instance;
     end_inst
```

```
end_pub

private
     instance  count  :  fix
               the_key  :  fix  fields

     to_itself
          create  k  c  =>  /* crea un objeto tipo key con los campos
                            especificados  por  los  parametros  */
               declare  newkey  :  key;

               newkey  ::  .tm  <=  allocate  'key';
               newkey  :  count  ::  c;
               newkey  :  the_key  ::  k;
               <-  newkey;
          end=>

          readkey  =>  /* lee  una  llave  del  archivo  de  entrada  */
               declare  auxkey  :  fix;

               auxkey  ::  .fix  <=  read;
               <-  .key  <=  create  auxkey  1;
          end=>

          eq  k  =>  /* compara  la  llave  de  una  instancia  con  algun
                        valor  entero  que  se  pasa  como  parametro  */
               <-  instance  :  the_key  <=  EQ  k;
          end=>

          setkey  k  =>  /* Asigna  un  valor  al  campo  the_key  */
               instance  :  the_key  ::  k;
               <-  instance;
          end=>

          setcount  c  =>  /* Asigna  un  valor  al  campo  count  */
               instance  :  count  ::  c;
               <-  instance;
          end=>

          +  c  =>  /* Incrementa  el  campo  count  en  c  unidades  */
               instance  :  count  ::  instance  :  count  <=  +  c;
               <-  instance;
          end=>
     end_it
end_priv
```

1) Respuesta create-key1. Esta respuesta construye objetos tipo key1. Los argumentos de la respuesta sirven como valores iniciales para los campos del objeto creado.

Esta respuesta cuenta con los siguientes datos:

Marco de Literales

CAMPO	CONTENIDO
1	Apuntador a Administrador key.

2 Entero 3

Se necesita el siguiente marco de temporales para activar esta respuesta:

CAMPO	CONTENIDO
0	Parametro k tipo FIX
1	Parametro c tipo FIX
2	Variable Temporal newkey tipo key

	CODIGO	SIGNIFICADO
1)	2	Push de objeto en marco de Literales.
2)	1	Apuntador a administrador key1 (literal 1).
3)	35	Crea Objeto. Tipo en tope del Stack.
4)	4	Longitud del objeto.
5)	12	Pop y almacena en campo de temporales.
6)	2	En variable newkey (campo 2).
7)	1	Push objeto de marco de temporales.
8)	2	Variable Newkey (campo 2 temporales).
9)	7	Push 2.
10)	1	Push objeto de marco de temporales.
11)	0	Argumento k (temporal 0).
12)	24	Activa primitiva.
13)	4	Primitiva guarda en record.
14)	1	Push objeto de marco de temporales.
15)	2	Variable newkey (temporal 2)
16)	2	Push de objeto en marco de Literales.
17)	2	Entero 3 (literal 2).
18)	1	Push objeto de marco de temporales.
19)	1	Argumento c (temporal 1).
20)	24	Activa primitiva.
21)	4	Primitiva Guarda en record.
22)	1	Push objeto de marco de temporales.
23)	2	Variable newkey (temporal 2).
24)	16	Return tope del stack.

Correspondencia de codigos:

```
create k c => /* crea un objeto tipo key con los campos
                 especificados por los parametros */
     declare newkey : key;

     newkey :=                                          (5),(6)
              .tm <= allocate 'key';                    (1)..(4)
     newkey ! count := c;                               (7)..(13)
     newkey ! the_key := k;                             (14)..(21)
     <- newkey;                                         (22),(23),(24)
end=>
```

Un objeto de tipo key1 contiene los siguientes campos:

campo (de 32 bits)	contenido
0	4 - longitud del objeto.
1	apuntador al administrador key1.
2	k - campo the_key del objeto.

3 c - campo count del objeto.

ii) Respuesta readKey-Keyi. Respuesta que lee los datos necesarios para la creacion de un objeto tipo keyi. El resultado es el objeto creado a partir de los datos leidos.

Marco de Literales

CAMPO	CONTENIDO
1	Apuntador a Respuesta create-Key.

Se necesita el siguiente marco de temporales para activar esta respuesta:

CAMPO	CONTENIDO
0	Variable Temporal auxKey tipo FIX

	CODIGO	SIGNIFICADO
1)	8	PushNil.
2)	24	Activa primitiva.
3)	9	Primitiva de lectura de entero.
4)	6	Push 1.
5)	22	Envia mensaje.
6)	1	Respuesta Create-Key (literal 1).
7)	2	Con 2 argumentos.
8)	16	Return tope del stack.

```
readKey => /* lee una llave del archivo de entrada */
     declare auxKey : fix;

     auxKey := .fix <= read;                          (1)..(4)
     <- .key <= create auxKey 1;                      (5)..(8)
end=>
```

iii) Respuesta eq-Keyi. Esta respuesta permite un numero entero con el campo the_key de un objeto tipo keyi.

Marco de Literales - No necesita.

Se necesita el siguiente marco de temporales para activar esta respuesta:

CAMPO	CONTENIDO
0	Parametro k tipo FIX

	CODIGO	SIGNIFICADO
1)	0	Push campo del receptor.
2)	2	Campo the_Key (campo 2).
3)	1	Push objeto de marco de temporales.
4)	0	Argumento k (temporal 0).
5)	24	Activa primitiva.
6)	10	Primitiva de Igualdad de Enteros.
7)	16	Return tope del stack.

```
eq k => /* compara la llave de una instancia con algun
```

```
                valor entero que se pasa como parametro */
          <-                                                    ( 7 )
              instance ! the_key                              ( 1 ),( 2 )
                                <= EQ                         ( 5 ),( 6 )
                                      k;                      ( 3 ),( 4 )
      end=>
```

iv) Respuesta setkey-key1. Esta respuesta permite asignar un valor al campo the_key de un objeto tipo key1.

Sin Marco de Literales.

Se necesita el siguiente marco de temporales para activar esta respuesta:

	CAMPO	CONTENIDO
	0	Parámetro k tipo FIX

	CODIGO	SIGNIFICADO
1)	1	Push objeto de marco de temporales.
2)	0	Parámetro K (temporal 0).
3)	11	Pop y almacena en campo del receptor.
4)	2	Campo the_key (campo 2).
5)	17	Return receptor.

```
setkey k => /* Asigna un valor al campo the_key */
      instance ! the_key := K;                          ( 1 )..( 4 )
      <- instance;                                            ( 5 )
end=>
```

v) Respuesta setcount-key1. Con esta respuesta se puede asignar un valor al campo count de un objeto tipo key1.
Sin Marco de Literales.

Se necesita el siguiente marco de temporales para activar esta respuesta:

	CAMPO	CONTENIDO
	0	Parámetro c tipo FIX

	CODIGO	SIGNIFICADO
1)	1	Push objeto de marco de temporales.
2)	0	Parámetro c (temporal 0).
3)	11	Pop y almacena en campo del receptor.
4)	3	Campo count (campo 3).
5)	17	Return receptor.

```
setcount c => /* Asigna un valor al campo count */
      instance ! count := c;                            ( 1 )..( 4 )
      <- instance;                                            ( 5 )
end=>
```

vi) Respuesta +. Esta respuesta permite incrementar el valor del campo count de un objeto tipo key1 en las unidades especificadas

por el argumento.
Sin Marco de Literales.

Se necesita el siguiente marco de temporales para activar esta respuesta:

	CAMPO	CONTENIDO
	0	Parámetro c tipo FIX

	CODIGO	SIGNIFICADO
1)	0	Push campo del receptor.
2)	3	Campo count (campo 3).
3)	1	Push objeto de marco de temporales.
4)	0	Parámetro c (temporal 0).
5)	24	Activa primitiva.
6)	0	Primitiva suma de enteros.
7)	11	Pop y almacena en campo del receptor.
8)	3	Campo count (campo 3).
9)	17	Return receptor.

```
+ c => /* Incrementa el campo count en c unidades */
     instance ! count :=                          (7),(8)
                  instance ! count                (1),(2)
                                    <= +          (5),(6)
                                         c;       (3),(4)
     <- instance;                                    (9)
end=>
```

VI.3.2 . Administrador key1

El administrador key1 define la superclase de "Key", por lo cual le hereda a key sus respuestas. Las unicas respuestas de "key1" que se heredan efectivamente son las que tienen nombres diferentes a las de "Key". Dichas respuestas son: "write" "Key" y "eq0".

```
[
administrator .Key
public
     to_itself
          create <key value>.fix <count value>.fix =>
          <- instance;

          readkey =>
          <- instance;

     end_it

     to_instance
          eq <K>.fix =>
          <- <Es contenido de key = k?>.boolean;

          setkey <K>.fix =>
```

```
            <-   instance;

            setcount   <k>.fix   =>
            <-   instance;

            +   <c>.fix   =>
            <-   instance;

            write =>
            <-;

            key =>
            <-   <the_key>.fix;

            eq0 =>
            <-  <Es  la  llave  key  :  0?>.boolean;
      end_inst
end_pub

private
      instance   count   :   fix
                 the_key  :  fix  fields

      to_itself
            create  k  c  =>
                                    .
                                    .
                                    .
            end=>

            readKey   =>
                                    .
                                    .
                                    .
            end=>
      end_it

      to_itself
            eq k =>
                                    .
                                    .
                                    .
            end=>

            setKey  k  =>
                                    .
                                    .
            end=>

            setcount  c  =>
                                    .
                                    .
            end=>

            + c =>
```

```
            end=>

            write =>
                  " Key = " <= write;
                  instance ; the_key <= write;
                  " count = " <= write;
                  instance ; count <= writeln;
                  <-;
            end =>

            key =>
                  <- instance ; the_key;
            end =>

            eq0 =>
                  <- instance ; the_key <= EQ 0;
            end =>

      end_it
end_priv
```

1) Respuesta write-key. Esta respuesta despliega el contenido de un objeto tipo key.

Marco de Literales

CAMPO	CONTENIDO
1	String " key = "
2	String " count = "

Marco de Temporales - No necesita.

	CODIGO	SIGNIFICADO
1)	2	Push de objeto en marco de Literales.
2)	1	Literal " key = " (literal 1).
3)	24	Manda Primitiva.
4)	11	Primitiva Despliega String.
5)	0	Push Campo del Resceptor.
6)	2	Campo the_key (campo 2 del receptor).
7)	24	Manda Primitiva.
8)	7	Primitiva de escritura de entero.
9)	2	Push de objeto en marco de Literales.
10)	2	Literal " count= " (literal 2).
11)	24	Manda Primitiva.
12)	11	Primitiva Despliega String.
13)	0	Push campo del receptor.
14)	3	Campo count (campo 3).
15)	24	Manda Primitiva.
16)	18	Primitiva de escribe entero y sala linea
17)	16	Return tope del stack.

```
            write =>
                  " key = " <= write;                    (1)..(4)
                  instance ; the_key <= write;           (5)..(8)
```

```
            " count = " <= write;                          (9)..(12)
            instance ! count <= writeln;                   (13)..(16)
            <-;                                                (17)
        end =>
```

iii) Respuesta key-key. Esta respuesta regresa el valor del campo the_key de un objeto tipo key.

Marco de Literales - No necesita.

Marco de Temporales - No necesita.

	CODIGO	SIGNIFICADO
1)	0	Push Campo del receptor.
2)	2	Campo the_Key del receptor.
3)	16	Return tope del stack.

```
        key =>
              <-                                               (1)
                  instance ! the_key;                      (2),(3)
        end =>
```

iii) Respuesta eq0-key. Esta respuesta compara el valor del campo the_key de un objeto tipo key con cero.

Marco de Literales - No necesita.

Marco de Temporales - No necesita.

	CODIGO	SIGNIFICADO
1)	0	Push campo del receptor.
2)	2	Campo the_key (campo 2 del receptor).
3)	5	Push 0.
4)	24	Manda Primitiva.
5)	10	Primitiva de igualdad de enteros.
6)	16	Return tope del stack.

```
        eq0 =>
              <-                                               (6)
                  instance ! the_key                       (1),(2)
                                <= EQ 0;                   (3)..(5)
        end =>
```

VI.3.3 Administrador list_of_keys

Este administrador es el que maneja los objetos que representan listas de objetos de tipo key.

```
{
administrator   list_of_keys
public
      to_itself
            create =>
            <-  instance;
```

```
      end_it

      to_instance
            NullList?   =>
            <-   <Es   la   lista   vacia?>.boolean;

            search   <k>.fix   =>
            <-   instance;

            write =>
            <-;
      end_inst
end_pub

private
      instance   l   :   list   fields

      to_itself
            create =>
                  declare   aux   :   list_of_keys   temporary

                  aux   :=   .tm   <-   create   'list_of_keys';
                  aux   :   l   :=   .lprim   <-   create;
                  <- aux;
            end =>
      end_it

      to_instance

            NullList?   =>
                  <-   instance   :   l   <-   ListNull?;
            end =>

            search   k   =>
                  declare
                        la   :   list_of_keys
                        b   :   boolean   temporary

                        .if   <-   (instance   <-   NullList?)   then
                              instance   :   l   :=   instance   :   l
                                    <-   conslist   k
                        else
                        [
                              b   :=   true;
                              la   :=   instance   :   l;
                              .repeat   <-   while   (   la   <-   NullList?
                                                      <- not
                                                      <-   and   b   )
                                    .if   <-   (la   <-   head   <-   eq   k)   then
                                    [
                                          b   :=   false;
                                          la   <-   head   <-   +   1;
                                    ]
                                    else
                                          la   :=   la   <-   tail;
```

```
                    .if  <=  b  then
                         instance  :  1  :=  instance  :  1
                              <=  conslist  k
               ]
               <-   instance;
          end =>

          write =>
               declare  la  :  list_of_keys  temporary

               la  :=  instance  :  1;
               .repeat  <=  while  [  la  <=  NullList?  <=  not  ]
               [
                    la  <=  head  <=  write;
                    la  :=  la  <=  tail;
               ]
               <-;
          end =>

     end_inst
end_priv
```

i) Respuesta create-list_of_keys. Esta respuesta permite crear un elemento de tipo list_of_keys. Un objeto de este tipo contiene un solo campo el cual contiene un objeto del tipo primitivo list.

Marco de Literales

CAMPO	CONTENIDO
1	Apuntador a administrador list_of_keys
2	Tipo primitivo Lista 7

Marco de Temporales

CAMPO	CONTENIDO
0	Variable aux tipo list_of_keys

	CODIGO	SIGNIFICADO
1)	2	Push Objeto del marco de literales.
2)	1	Apuntador a administrador list_of_keys (literal 1).
3)	35	Crea Objeto.
4)	3	Longitud del objeto = 3.
5)	12	Pop y almacena en marco del temporales.
6)	0	Variable aux (temporal 0).
7)	1	Push objeto en marco de temporales.
8)	0	Variable aux (temporal 0).
9)	7	Push 2.
10)	2	Push Objeto del marco de literales.
11)	2	Tipo primitivo 7 (literal 2).
12)	35	Crea Objeto.
13)	2	Longitud 2.
14)	24	Activa Primitiva.
15)	4	Primitiva Guarda.
16)	1	Push objeto en marco de temporales.

```
17)        0          Variable aux (temporal 0).
18)        16         Return tope del Stack.
```

Correspondencia de códigos:

```
create =>
      declare aux : list_of_keys temporary

      aux :=                                                    (5),(6)
            .tm <= allocate 'list';                             (1)..(4)
      aux ; 1 :=                                  (7)..(9),(14),(15)
            .list <= create;                                   (10)..(13)
      <- aux;                                                  (16)..(18)
end =>
```

Un objeto de tipo list_of_keys contiene los siguientes campos:

campo (de 32 bits)	contenido
0	4 - longitud del objeto.
1	apuntador al admini. list_of_keys.
2	objeto tipo primitivo '.list' - campo 1 del objeto.

ii) Respuesta NullList?. Esta respuesta nos permite distinguir al elemento nulo dentro de los objetos de tipo list_of_keys. El contenido de la respuesta es el siguente:

Marco de Literales - No necesita.

Marco de Temporales - No necesita.

	CODIGO	SIGNIFICADO
1)	0	Push campo del receptor.
2)	2	Campo 1 (campo 2 del receptor).
3)	24	Activa Primitiva.
4)	13	Primitiva de listas LisNull?
5)	16	Return Tope del Stack.

```
NullList?  =>
      <-                                                        (5)
         instance ; 1                                        (0),(2)
            <= ListNull?;                                    (3),(4)
      end =>
```

iii) Respuesta search. Esta respuesta realiza la busqueda de un elemento de tipo "key" dentro de un objeto de tipo "list_of_keys". Si la busqueda es exitosa, se incrementa el atributo "count" de la llave ya existente dentro de la lista. Si la busqueda no fue exitosa, se inserta la llave en la lista.

Marco de Literales.

	CAMPO	CONTENIDO
Marco de Temporales		
	CAMPO	CONTENIDO
	CODIGO	SIGNIFICADO
1)	9	Push Receptor.
2)	22	Envia Mensaje.
3)	1	Respuesta NullList? (literal 1)
4)	0	Sin argumentos.
5)	33	Salto si falso.
6)	10	Longitud del salto.
7)	0	Push campo del receptor.
8)	2	Campo 1 (campo 2 del receptor).
9)	1	Push Objeto en marco de temporales.
10)	0	Argumento K (temporal 0).
11)	24	Activa Primitiva.
12)	14	Primitiva de listas ConsList.
13)	11	Pop y almacena en campo del receptor.
14)	2	Campo 1 (campo 2 del receptor).
15)	26	Salto Incondicionado.
16)	63	Longitud del salto.
17)	6	Push 1.
18)	12	Pop y almacena en marco de temporales.
19)	2	Variable b (temporal 2).
20)	0	Push campo del receptor.
21)	2	Campo 1 (campo 2 del receptor).
22)	12	Pop y almacena en marco de temporales.
23)	1	Variable la (temporal 1).
24)	1	Push Objeto en marco de temporales.
25)	1	Variable la (temporal 1).
26)	24	Activa Primitiva.
27)	13	Primitiva de listas ListNull?
28)	24	Activa Primitiva.
30)	1	Push Objeto en marco de temporales.
31)	2	Variable b (temporal 2).
32)	24	Activa Primitiva.
33)	17	Primitiva AND.
34)	33	Salto si falso.
35)	32	Longitud del salto.
36)	1	Push Objeto en marco de temporales.
37)	1	Variable la (temporal 1).
38)	24	Activa Primitiva.
39)	15	Primitiva de listas head.
40)	1	Push Objeto en marco de temporales.
41)	0	Argumento K (temporal 0).
42)	23	Envio de mensaje con busqueda de respuesta.
43)	2	Respuesta EQ.
44)	1	Con 1 parametro.
45)	33	Salto si falso.
46)	13	Longitud del salto.
47)	5	Push 0.
48)	1	Push Objeto en marco de temporales.

```
49)     2       Variable b (temporal 2).
50)     1       Push Objeto en marco de temporales.
51)     1       Variable la (temporal 1).
52)     24      Activa Primitiva.
53)     15      Primitiva de listas head.
54)     6       Push 1.
55)     23      Envio de mensaje con busqueda.
56)     3       Respuesta + (literal 4)
57)     1       Con un argumento.
58)     26      Salto Incondicionado.
59)     6       Longitud del salto.
60)     1       Push Objeto en marco de temporales.
61)     1       Variable la (temporal 1).
62)     24      Activa Primitiva.
63)     16      Primitiva del listas tail.
64)     1       Push Objeto en marco de temporales.
65)     1       Variable la (temporal 1).
66)     26      Salto Incondicionado.
67)     -44     Longitud del salto.
68)     1       Push Objeto en marco de temporales.
69)     2       Variable b (temporal 2).
70)     33      Salto si falso.
71)     8       Longitud del salto.
72)     0       Push campo del receptor.
73)     2       Campo 1 (campo 2 del receptor).
74)     1       Push Objeto en marco de temporales.
75)     0       Argumento k (temporal 0).
76)     24      Activa Primitiva.
77)     14      Primitiva de listas ConsList.
78)     11      Pop y almacena en campo del receptor.
79)     2       Campo 1 (campo 2 del receptor).
80)     17      Return el Receptor.
```

```
search  k  =>
      declare
            la   :  list_of_keys
            b    :  boolean   temporary

            .if <=                                          (5),(6)
                  (instance                                   (1)
                      <= NullList?) then                    (2)..(4)
                instance : 1 :=                            (13)..(14)
                      instance : 1                          (7)..(8)
                      <=    conslist
                       (.key <= create                     (11)..(12)
                                        k 1)                (9)..(10)
            else                                            (15),(16)
            [
                  b := true;                                (17)..(19)
                  la := instance : 1;                       (20)..(23)
                  .repeat <= while [                         (34),(35)
                              la                             (24),(25)
                                 <=     NullList?            (26),(27)
                                 <= not                      (28),(29)
                                 <=    and    b     ]        (30)..(33)
```

```
                                .if <=                          (45),(46)
                                      (la                       (36),(37)
                                           <= head              (38),(39)
                                           <= eq k)            (40)..(44)
                                                          then
                                {
                                     b := false;               (47)..(49)
                                     la                         (50),(51)
                                      <= head                   (52),(53)
                                      <= + 1;                  (54)..(57)
                                }
                                else                            (58),(59)
                                     la :=                      (64),(65)
                                          la                    (60),(61)
                                            <=      tail;        (62),(63)
                           ],                                   (66),(67)
                           .if <= b then                       (68)..(71)
                                instance : 1 :=
                                     instance : 1               (72),(73)
                                     <= conslist                     (77)
                                      (.key <=
                                     create k 1)                (74)..(76)
                           <= instance;                         (78)..(80)
        end =>
```

iv) Respuesta write. Esta respuesta sirve para deplegar un elemento de tipo "list_of_keys". El significado de desplegarlo consiste en desplegar el contenido del mismo, o sea la secuencia de elementos de tipo "key" dentro del el.

Marco de Literales

CAMPO	CONTENIDO
1	Apunt. a respuesta write-key

Marco de Temporales

CAMPO	CONTENIDO
0	Variable "la" tipo list_of_keys

	CODIGO	SIGNIFICADO
1)	0	Push campo del Receptor.
2)	2	Campo 1 (campo 2).
3)	12	Pop y almacena en marco de temporales
4)	0	Variable la (temporal 0).
5)	1	Push Objeto de marco de temporales.
6)	0	Variable la (temporal 0).
7)	24	Activa Primitiva.
8)	13	Primitiva ListNull?
9)	24	Activa Primitiva.
10)	8	Primitiva Not.
11)	33	Salto si falso.
12)	15	Longitude del salto.
13)	1	Push Objeto de marco de temporales.
14)	0	Variable la (temporal 0)
15)	24	Activa Primitiva.

```
16)        15          Primitiva de listas head.
17)        23          Manda (con busqueda)
18)        1           Respuesta Write.
19)        0           Sin argumentos.
20)        1           Push Objeto de marco de temporales.
21)        0           Variable la (temporal 0).
22)        24          Activa Primitiva.
23)        16          Primitiva del listas Tail.
24)        12          Pop y almacena en Marco de Temporales.
25)        0           Variable la (temporal 0).
26)        26          Salto Incondicionado.
27)        -23         Longitud del salto.
28)        16          Return.

          write =>
               declare   la   :   list_of_keys   temporary

               la := instance ; 1;                                  (1)..(4)
               .repeat <= while                                    (11),(12)
                         [ la                                       (5),(6)
                         <= NullList?                               (7),(8)
                         <= not ]                                   (9),(10)
               [
                    la                                            (13),(14)
                         <= head                                  (15),(16)
                         <= write;                                (17)..(19)
                    la :=                                         (24),(25)
                         la                                       (20),(21)
                         <= tail;                                 (22),(23)
               ]                                                  (26),(27)
               <-;                                                     (28)
          end =>
```

VI.3.4 Programa Usuario

Enseguida se muestra un ejemplo de un programa "usuario" que emplea objetos administrados por "list_of_keys" y por "key". El programa crea una lista vacia, que va llenando con llaves construidas a partir de datos leidos. El proceso de llenado termina al leerse el dato 0. Por ultimo se despliega el contenido de la lista resultante.

Programa sencillo que usa los administradores "list_of_keys" y "key":

```
declare  K  :  key
         l  :  list_of_keys  temporary

     "PROGRAMA DE PRUEBA 'LISTA'
      TECLEE UN NUMERO" <= write;
     l  :=  .list_of_keys  <=  create;
     "  K= "  <= write;
     K  :=  .key  <=  readin;
     .repeat <= while [ K  <= eq0  <= not  ]
```

```
    [
        l :: l <= search k;
        " K= " .<= write;
        k :: .key <= readln;
    ];
    l <= write;
<-;
```

Marco de Literales

CAMPO	CONTENIDO
1	Apunt. a Respuesta create-list_of_keys
2	Apunt. a Respuesta search-list_of_keys
3	Apunt. a Respuesta write-list_of_keys
4	String " K= "
5	String "PROGRAMA DE PRUEBA 'LISTA' TECLEE UN NUMERO"
6	Apunt. a Respuesta readkey-key
7	Apunt. a Respuesta eq0-key

Marco de Temporales

CAMPO	CONTENIDO
0	Variable "l" tipo list_of_keys
1	Variable "k" tipo key

	CODIGO	SIGNIFICADO
1)	2	Push objeto de marco de literales.
2)	5	String "PROGRAMA DE PRUEBA 'LISTA' TECLEE UN NUMERO". Literal 5.
3)	24	Activa primitiva.
4)	11	Primitiva de despliegue de string.
5)	22	Manda mensaje.
6)	1	Respuesta create-list_of_keys (literal 1).
7)	-1	Sin parametros ni receptor en stack.
8)	12	Pop y almacena en marco de temporales.
9)	0	Variable temporal l (temporal 0).
10)	2	Push objeto de marco de literales.
11)	4	String " K= ". Literal 4.
12)	24	Activa primitiva.
13)	11	Primitiva de despliegue de string.
14)	22	Manda mensaje.
15)	6	Respuesta readkey-key.
16)	-1	Sin parametro ni receptor.
17)	12	Pop y almacena en marco de temporales.
18)	1	Variable k (temporal 1).
19)	1	Push objeto de marco de temporales.
20)	1	Variable k (temporal 1).
21)	22	Manda mensaje.
22)	7	Respuesta eq0-key.
23)	0	Sin argumentos.
24)	24	Activa primitiva.
25)	8	Primitiva not.
26)	33	Salto si falso.
27)	20	Longitud del salto.
28)	1	Push objeto de marco de temporales.

```
29)         0           Variable 1 (temporal 0).
30)         1           Push objeto de marco de temporales.
31)         1           Variable k (temporal 1).
32)         22          Manda mensaje.
33)         2           Respuesta search (literal 2).
34)         1           Con un parametro.
35)         12          Pop y almacena en marco de temporales.
36)         0           Variable 1 (temporal 0).
37)         2           Push objeto de marco de literales.
38)         4           String " K= " (literal 4).
39)         24          Activa primitiva.
40)         11          Primitiva de despliegue de string.
41)         22          Manda mensaje.
42)         6           Respuesta readkey-key (literal 6).
43)         -1          Sin argumentos ni receptor.
44)         12          Pop y almacena en marco de temporales.
45)         1           Variable k (literal 1).
46)         26          Salto incondicionado.
47)         -29         Longitud del salto.
48)         1           Push objeto de marco de temporales.
49)         0           Variable 1 (temporal 0).
50)         22          Manda mensaje.
51)         3           Respuesta write-list_of_keys (literal 3).
52)         0           Sin argumentos.
53)         34          halt.
```

```
declare  k  :  key
         l  :  list_of_keys  temporary

     "PROGRAMA  DE  PRUEBA  'LISTA'
      TECLEE UN NUMERO"                                          (1),(2)
                              <= write;                          (3),(4)
     l :=                                                        (8),(9)
          .list_of_keys <= create;                              (5)..(7)
     " K= "                                                     (10),(11)
            <= write;                                           (12),(13)
     k :=                                                       (17),(18)
          .key <= readln;                                      (14)..(16)
     .repeat <= while                                           (26),(27)
                          [ k                                   (19),(20)
                             <= eq0                            (21)..(23)
                             <= not ]                           (24),(25)
     [
          l :=                                                  (35),(36)
                l                                               (28),(29)
                     <= search k;                              (30)..(34)
          " K= "                                                (37),(38)
                     <= write;                                  (39),(40)
          k :=                                                  (44),(45)
               .key <= readln;                                 (41)..(43)
     ];                                                         (46),(47)
     !                                                          (48),(49)
               <= write;                                       (50)..(52)
<=;                                                                (53)
```

Muestra de la ejecución del programa:

```
PROGRAMA DE PRUEBA 'LISTA'
TECLEE UN NUMERO
 K: 23
 K: 128
 K: 876
 K: 987
 K: 6789
 K: -678
 K: 987
 K: 23
 K: 472
 K: 23
 K: 0
 KEY = 472                    COUNT = 1
 KEY = -678                   COUNT = 1
 KEY = 6789                   COUNT = 1
 KEY = 987                    COUNT = 2
 KEY = 876                    COUNT = 1
 KEY = 128                    COUNT = 1
 KEY = 23                     COUNT = 3
```

VI.4 Referencias

[Gerzso 83] Gerzso, M. "Report on the Language TM: its design and definition; IIMAS UNAM 1983.

[Wirth 76] Wirth, N "Algorithms + Data Structures = Programs", Prentice Hall.

CAPITULO VII

Discusión y Conclusiones

Las metas a alcanzar con esta tesis consistentieron en diseñar e implantar una máquina virtual para TM. Uno de los criterios que más se enfatizó fue el de maximizar la rapidez de ejecución. Las estrategias que se siguieron para agilizar la ejecución con la máquina virtual fueron las siguientes:

- Tipos Primitivos. Se incluyeron algunos tipos primitivos con el fin de lograr la generación de un código más eficiente (los tipos más complejos se construyen a partir de tipos primitivos, las operaciones sobre los primitivos no están escritas en TM, sino como rutinas en lenguaje C). La existencia de tipos primitivos tambien tiene la ventaja de permitir la optimización de la cantidad de memoria asignada a cada tipo primitivo, existiendo el caso (números enteros) en que el objeto mismo se almacena en el apuntador, o bien casos en los que no se sigue el formato general y no es necesario apartar lugares para la longitud del objeto y la clase del mismo.

- Eliminación de Búsquedas. Uno de los procesos que más lentitud produce en la ejecución de sistemás escritos en lenguajes orientados a objetos consiste en las búsqueda de las respuestas que corresponden a cada mensaje. Esta implantación está diseñada de manera que las búsquedas de respuestas se realizen a tiempo de compilación cuando sea posible. Existen casos en los que es imposible determinar cual es la respuesta apropiada a un mensaje a tiempo de compilación (por ejem. cuando se envian mensajes a variables de tipo anything), sólo en esos casos, el compilador generará código que indique a la máquina virtual que debe realizar la búsqueda de la respuesta.

- Optimización de Estructuras de Control. Otra de las operaciones que puede consumir más tiempo de ejecución aparece en las instrucciones de cambio control (como if-then-else o repeat-until, etc.). Normalmente estas instrucciones se implementan como mensajes y se pasan los bloques de instrucciones como argumentos (por ejemplo una instrucción de tipo if-then-else contiene dos parametros de tipo bloque de instrucciones, uno para el "if" y otro para el "else"). Si se sigue ese método, la transferencia de control a alguno de los bloques es comparable a la invocación de un procedimiento (o al envío de un mensaje) ya que se debe crear un contexto especial para el nuevo bloque. Es por esto que se decidió que en la presente implantación las instrucciones de cambio de control se implantaran simplemente utilizando saltos condicionados dentro del mismo bloque, sin llevarse a cabo el envío de ningun mensaje. Esto tiene la desventaja de que se tendrá un número fijo de instrucciones de cambio de control, pero tiene la ventaja de que se logrará un aumento considerable en la velocidad de ejecución.

-Stack de Contextos. Estudiando el comportamiento de envío de mensajes en TM, se llegó a la conclusión de las cadenas de activación de respuestas se podrían representar mediante la utilización de un stack.

Los contextos no son considerados como objetos y su administración se realiza de manera separada (no ocupan espacio en el heap de la memoria de objetos). Gracias a esto las operaciones de creación y destrucción de contextos no son necesarias ya que acciones equivalentes se realizan moviendo únicamente el apuntador al tope del stack.

- Permanencia de argumentos y receptor en el stack de evaluaciones. En lugar de sacar del stack a los argumentos y el receptor de un mensaje para copiarlos dentro de un contexto (como se realiza en la definición de la máquina virtual de SmallTalk-80 y en la de Fernando F.), se dejan allí y se accesan por un desplazamiento a partir de la posición del tope del stack en el momento de la invocación del mensaje. Al finalizar la ejecución del mensaje se mueve el tope del stack de manera equivalente a la realización de varios pops para sacar argumentos y receptor.

-Prefetch. Los códigos a interpretar se van extrayendo de la respuesta activa de cuatro en cuatro con el fin de agilizar la ejecución, ya que el cada lectura a un campo de un objeto implica pasar por dos niveles de direccionamiento.

No se han llevado a cabo evaluaciones de eficiencia formales y exhaustivas para la máquina virtual diseñada. Se han escrito programas con códigos para la máquina y se ha realizado un gran número de ejecuciones de prueba. La comparación de velocidades entre esta implantación y las anteriores ha sido altamente favorable para la actual, pero no se ha cuantificado de manera precisa.

Otras evaluaciones de eficiencia que se han efectuado han consistido en escribir programas en lenguaje Pascal (Whitesmiths) en la computadora VAX que efectuaran las mismas funciones que realizan los programas de ejemplo (Ver cap. VII) con el fin de comparar las velocidades de ejecución. Se observó que los programas interpretados por la máquina virtual se ejecutaban en un tiempo aproximadamente del doble que los programas en Pascal. Cabe aclarar que el compilador de Pascal genera código para la computadora VAX.

El conjunto de funciones primitivas en TM no se ha definido. Con una definición del conjunto mínimo de funciones primitivas se podría dar al usuario la confianza de que en cualquier instalación de TM se puede disponer de esas herramientas básicas. La definición de ese conjunto mínimo se logrará partiendo de las experiencias que se logren con el uso del sistema.

Por lo anterior, en esta implantación se trató de que la adición y cambio de las rutinas que llevan a cabo las funciones primitivas se pudiera realizar de manera sencilla (Ver apéndice sobre implementación). En el sistema actual se dispone de un conjunto de funciones primitivas que fue elegido de manera arbitraria.

Un objetivo adicional de este trabajo consistió en el diseño de la especificación formal de una de las partes mas importantes del sistema, la memoria de objetos. La realización de esta especificación permitió revisar facilmente el comportamiento de las operaciones sobre la memoria de objetos implantada. Gracias a la especificación se pudieron detectar y corregir algunas fallas de la implantación (se descubrió que algunas operaciones no estaban definidas para ciertos datos de entrada con los que debían funcionar). Las memorias de las computadoras están organizadas como una secuencia de celdas. La formalización permitió descubrir que si se pudiera construir una memoria que se comportara como un conjunto de celdas se eliminaría el problema de la fragmentación y con ello se evitaría la necesidad de realizar compactaciones.

APENDICE

Implantación

A.1 Datos Generales:

Computadora en la que reside el sistema: Minicomputadora VAX-11 730 del Departamento de Sistemas de Cómputo del IIMAS.

Sistema Operativo VAX/VMS versión 3.5

Lenguaje que se empleó para realizar el sistema: Lenguaje C [1] usando el compilador marca Whitesmiths (descrito en el manual [2]). Macroensamblador VAX-11 MACRO Versión 3.00 (descrito en el manual [3]).

Archivos Fuente:

arr.mar - Conjunto de rutinas escritas en macro que realizan las funciones primitivas del manejador de la memoria de objetos.

mem.c - Rutinas del manejador de la memoria escritas en C. Estas rutinas definen la interfaz con el entre la memoria de objetos y el interprete.

lotm.c - Rutinas que realizan la instalación en la memoria de objetos de los objetos de tipo administrador y códigos necesarios para la ejecución del programa en TM. Contiene el cargador ligador.

int.c - Interprete que ejecuta las instrucciones que aparecen en los objetos de tipo código que representran el programa TM que se desea ejecutar. En este archivo se encuentra el módulo principal del sistema.

prim.c - Archivo que contiene las rutinas primitivas del sistema

Actualmente no hay disponible un compilador que genere código para la máquina virtual. Se estima que el compilador estará disponible a principios de 1987.

A.2 Guía de Uso

Es necesario tener en cuenta que la presente guía sirve para el uso de la máquina virtual y no del sistema TM completo. Para poder usar la máquina virtual los códigos en lenguaje TM deberán compilarse manualmente.

Para realizar programas que sean ejecutados por la máquina virtual se siguen los siguientes pasos:

a) Se definen los administradores necesarios para la ejecución

del programa.

b) Se define la expresión o programa usuario que empleará los diversos tipos de datos definidos por los administradores.

c) Se escribe el código de cada administrador con sus respuestas en un archivo separado (siguiendo la convención empleada por el cargador ligador, ver cap. VI). Los nombres de los archivos de administradores con sus repuestas deberan tener la extensión ".lib". El código correspondiente al programa usuario deberá residir en el archivo "interac.txt".

d) Se activa el sistema. Actualmente el sistema reside en el directorio [SYS$USERS]SERGIO/TMMV. Para ejecutar el sistema simplemente es necesario teclear el comando "RUN INT", despues del cual aparece el sig. menú:

1. Cargar Administradores y Respuestas
2. Ejecutar el Programa Usuario
3. Recoleccion
4. FIN

teclee el número correspondiente.

Opción 1. Al seleccionar la opción "1" el programa pedirá los nombres de los archivos donde se encuentran los códigos de los administradores y sus respuestas uno por uno (ver capitulo VI). Los nombres deberán ser tecleados sin incluir la extensión y podrán introducirse en cualquier orden. Despues de cada nombre se presiona la tecla "ENTER" o "RETURN". Al finalizar la lista de archivos se presiona dos veces la tecla "ENTER". No se debe incluir el nombre "interac.txt" en la lista de archivos, ya que se toma por omisión como el último de la lista.

Despues de pedir la lista de archivos, el programa aloja los objetos correspondientes a códigos y administradores en la memoria de objetos. Se tratan de resolver todas las referencias que existan entre los módulos cargados. Se despliega una lista de referencias no resueltas. El hecho de que existan referencias no resueltas no impide la ejecución de los códigos instalados, ya que es posible que dicha referencia no se requiera.

Todas las referencias no resueltas tendran asociado un apuntador nulo "NIL".

Una vez que se han instalado y ligado los archivos en la memoria de objetos, se puede proceder a ejecutar el programa usuario.

Opción 2. Al seleccionar esta opción se tratará de ejecutar el objeto identificado como programa usuario.

Opción 3. Con esta opción se realiza la recolección de basura y compactación por el esquema de marcado y barrido (ver

administrador de la memoria cap IV). Ademas de tener este control "manual" para hacer funcionar el recolector, éste funciona automáticamente cuando no se puede satisfacer algun requerimiento de memoria.

Opción 4. Con esta opción se termina de emplear la máquina virtual. Esta implantación no cuenta con objetos persistentes, por lo que los objetos almacenados en la memoria de objetos se perderán al elegir esta opción.

A.3 Primitivas

Existe un conjunto de rutinas de respuesta a mensajes que no estan construidas por medio de códigos para la máquina virtual. Este tipo de rutinas se refieren a los mensajes básicos que se mandan a objetos primitivos. Las rutinas estan implantadas en lenguaje C y marcan el final de la recursividad en el envío de cualquier mensaje (todas las respuestas se construyen en base a la existencia de este conjunto de primitivas).

En la presente implantación del sistema se cuenta con un conjunto de 24 operaciones primitivas. Para agregar otras operaciones primitivas al sistema se debe modificar el archivo "primi.c", compilarlo y realizar el ligado de los archivos objeto "int", "primi", "lotm", "low" y "arr".

La modificación al archivo "primi.c" es sencilla; basta teclear la definición en C de la nueva operación y agregar su nombre al arreglo de apuntadores a funciones "Primitiva[]". La nueva operación tendrá un número de identificación que corresponde a su posición en el arreglo. Dicho número se utiliza al realizar la instrucción "HandaPr" (ver sección V.3.6).

A.4 Referencias

[1] Kernighan and Ritchie, "The C Programming Language", Prentice-Hall 1978.

[2] C Programmers' Manual Whitesmiths, Ltd. Edition 2.2 March 1983

[3] VAX-11 MACRO User's Guide, Digital Equipment Corporation May 1982