Universidad Nacional Autónoma de México

Facultad de Ciencias

Recolección de basura: recolección precisa en un recolector conservador

T E S I S

QUE PARA OBTENER EL TÍTULO DE

Licenciado en Ciencias de la Computación

P R E S E N T A :

Francisco Lorenzo Solsona Cruz

Directora de tesis: M. en C. Elisa Viso Gurovich

FACULTAD DE CIENCIAS
SECCION ESCOLAR

2001

Universidad Nacional
Autónoma de México

UNAM – Dirección General de Bibliotecas
Tesis Digitales
Restricciones de uso

DERECHOS RESERVADOS ©
PROHIBIDA SU REPRODUCCIÓN TOTAL O PARCIAL

Todo el material contenido en esta tesis esta protegido por la Ley Federal del Derecho de Autor (LFDA) de los Estados Unidos Mexicanos (México).

El uso de imágenes, fragmentos de videos, y demás material que sea objeto de protección de los derechos de autor, será exclusivamente para fines educativos e informativos y deberá citar la fuente donde la obtuvo mencionando el autor o autores. Cualquier uso distinto como el lucro, reproducción, edición o modificación, será perseguido y sancionado por el respectivo titular de los Derechos de Autor.

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

Comunicamos a usted que hemos revisado el trabajo escrito:

"Recolección de basura: recolección precisa en un recolector conservador"

realizado por Francisco Lorenzo Solsona Cruz

con número de cuenta 9561797-8, quién cubrió los créditos de la carrera de Ciencias de la Computación

Dicho trabajo cuenta con nuestro voto aprobatorio.

Atentamente

Director de Tesis Propietario	M en C. Elisa Viso Gurovich	Elisa Viso
Propietario	M. en C. Javier García García	
Propietario	M. en C. José de Jesús Galaviz Casas	
Suplente	Dr Osvaldo Cairó Batistuti	
Suplente	M en C. Virginia Abrín Batule	Virginia Abrín Batule

Consejo Departamental de Matemáticas

M en C. C. Ma Guadalupe E. Ibargüengoitia González

Contenido

Lista de Figuras ix

1 Antecedentes 1
1.1 Jerarquía de memoria 1
1.2 Conjuntos de instrucciones 5
1.2.1 La estructura de compiladores recientes 6
1.2.2 Asignación de memoria en lenguajes de alto nivel 8
1.3 Recolección de basura 9

2 Recolección de basura 13
2.1 Representación de objetos 14
2.2 Técnicas básicas para recolección de basura 14
2.2.1 Conteo de referencias 15
Conteo de referencias pospuesto 17
Variaciones al conteo de referencias 18
2.2.2 Recolección de marcado y barrido 19
2.2.3 Recolección de marcado y compactación 20
2.2.4 Recolección de basura de copiado 21
Un recolector de copiado sencillo: "Detente y copia" usando semi-espacios 22
Eficiencia de la recolección de copiado 25
2.2.5 Recolección implícita sin copiado 25
2.2.6 Técnicas básicas de recorrido, ¿cuál escoger? 27
2.2.7 Problemas con los recolectores de recorrido sencillo 29
2.2.8 Recolectores de basura conservadores 30

3 Recolectores de recorrido incremental 31
3.1 Conservadurismo y coherencia 32
3.2 Marcado tricolor 33
3.2.1 Enfoques incrementales 34
3.3 Algoritmos con barrera de escritura 35
3.3.1 Algoritmos de fotografía instantánea 35
3.3.2 Algoritmos con barrera de escritura de actualización incremental 36
3.4 Algoritmos con barrera de lectura de Baker 38

3.4.1 Copiado incremental 38
3.4.2 Algoritmo incremental sin copiado de Baker –La rueda de ardilla (treadmill) 39
3.4.3 Conservadurismo de la barrera de lectura de Baker 40
3.5 Recolección con copiado de replicación 41
3.6 Conservadurismo y coherencia 41
3.6.1 Coherencia y conservadurismo en recolección sin copiado 42
3.6.2 Coherencia y conservadurismo en recolección con copiado 43
3.6.3 Recolección "radical" y recorrido oportunista 43
3.7 Comparación de técnicas incrementales 44
3.8 Recolección de recorrido en tiempo real 45
3.8.1 Recorrido del conjunto raíz 47
3.8.2 Garantía de progreso suficiente 48
3.8.3 Discusión 51
3.9 ¿Cuál algoritmo incremental escoger? 51

4 Recolección de basura generacional 53
4.1 Múltiples sub-heaps con frecuencias de recolección distintas 54
4.2 Política de avance 57
4.3 Organización del heap 58
4.3.1 Sub-áreas en esquemas de copiado 58
4.3.2 Generaciones en sistemas sin copiado 60
4.3.3 Discusión 60
4.4 Seguimiento de referencias intergeneracionales 61
4.4.1 Tablas de indirección 61
4.4.2 Conjuntos memoria de Ungar 62
4.4.3 Marcado de páginas 63
4.4.4 Marcado de palabras 64
4.4.5 Marcado de cartas 64
4.4.6 Listas de almacenamiento 65
4.4.7 ¿Qué estrategia de barrera de escritura escoger? 65
4.5 Revisión del principio generacional 67
4.6 Trampas de la recolección generacional 67
4.6.1 El problema del "cerdo en la serpiente" 68
4.6.2 Objetos pequeños alojados en el heap 68
4.6.3 Conjuntos raíz muy grandes 69
4.7 Recolección generacional de tiempo real 70

5 Consideraciones de localidad 73
5.1 Variedades de localidad de referencia 73
5.2 Localidad y objetos de vida corta 76
5.3 Localidad de recorrido de memoria 76
5.4 Agrupamiento de objetos de larga vida 77
5.4.1 Agrupamiento estático 77
5.4.2 Reorganización dinámica 78

5.4.3 Coordinación con la paginación 78

6 Detalles de implementación de bajo nivel **81**

6.1 Referencias etiquetadas y encabezados de objetos 81
6.2 Localización conservadora de referencias 83
6.3 Soporte lingüístico y referencias inteligentes 85
6.4 Cooperación del compilador y optimizaciones 86
6.4.1 Recolección en cualquier momento Vs. recolección en puntos seguros . 86
6.4.2 Conjuntos de registros particionados Vs. almacenamiento de representaciones variables 87
6.4.3 Optimizaciones sobre la recolección de basura misma 88
6.5 Manejo de almacenamiento libre 88
6.6 Representación compacta de datos en el heap 89

7 Características del lenguaje relacionadas a la recolección de basura **91**

7.1 Referencias débiles 91
7.2 Finalización 92
7.3 Múltiples heaps manejados de manera distinta 92

8 El recolector conservador por excelencia **95**

8.1 Propuesta de modificación al recolector conservador de Boehm 97
8.2 Qué modificar para hacerlo explícito y menos conservador 98
8.3 Conclusiones 101

Bibliografía **103**

Lista de Figuras

1.1 Rendimiento comparativo de **memoria** y **procesador** 2
1.2 Niveles en una jerarquía de memoria 3
1.3 Pasadas o fases de un compilador moderno. 7

2.1 Conteo de referencias 15
2.2 Conteo de referencias, con un ciclo irrecuperable 16
2.3 Un recolector simple con semi-espacios 22
2.4 Algoritmo de Cheney 24
2.5 Uso de memoria en un recolector con semi-espacios 26

3.1 Violación a la invariante tricolor 34
3.2 Recolector *treadmill* durante la recolección 40

4.1 Un recolector generacional de copiado simple 54
4.2 Uso de memoria en recolectores generacionales 55

Agradecimientos

Antes que nada, les agradezco a mis padres: Silvia y Francisco, a mi hermano Martín y a mis hermanas: Santa del Carmen, Silvia, Cynthia y Claudia por su apoyo incondicional y por siempre estar ahí, aunque yo no sea modelo de hijo o hermano. También le agradezco a todos mis sobrinas y sobrinos, que han venido a llenar de luz a la familia.

Mi directora de tesis, mi ejemplo a seguir y una de mis mejores amigas: Elisa Viso Gurovich, no sólo se merece mi agradecimiento sino mi más grande respeto y admiración: ¡ojalá tuvieramos más mujeres como ella en México!. Le agradezco también a Virgnia Abrín Batule, gran mujer, gran líder, gran amiga, mi jefa en muchos proyectos y la persona con mejor don de mando y organización que he conocido, en palabras robadas a Elisa: "una mujerona". En la misma línea, también le agradezco a Mauricio Aguilar González, mi jefe y amigo en la biblioteca de la Facultad, a quien le debo muchas charlas interesantes y un montón de conocimientos que emanan de él como si fuera fuente.

El español no tiene las palabras suficientes para que yo pueda siquiera comenzar a agraderle a mi novia Karla Ramírez Pulido, quien nunca cesó de empujarme y estuvo conmigo –allí en las trincheras– en las buenas y en las malas, a quién, la Liga de la Justicia, a tenido a bien asignarle la medalla "a la mujer más aguantadora (y consentidora)" y con quien quiero, espero, deseo, ansío, ruego pasar el resto de mis días.

Agradezco al resto de mis sinodales (un par están arriba): con especial cariño a Javier García García, excelente profesor, un gran amigo y maestro Además de que fue justo bajo su atenta visión, que yo comencé (¿y aprendí?) a dar ayudantías y lo mejor es que nunca se quejó, creo. A Osvaldo Cairó Battistutti, un par de clases muy divertidas durante su sabático y después una larga y muy fructifera relación personal y de trábajo que nos llevó desde Acapulco, hasta a Argentina, pasando por Alemania. Por último en esta clasificación, le agradezco también a José de Jesús Galaviz Casas, de quien nunca tuve el gusto de ser alumno (a eso atribuyo mi casi nula base de datos de chistes), pero con quien he tenido el gusto de participar en uno que otro seminario y con quien he discutido también cosas muy interesantes.

También le agradezco a todos mis amigos y compañeros.

¿Verdad que tuvieron miedo? ¡Lo sabía!, aquí va el resto de la lista y antes de que se me pase decirles a todos: **¡Muchas gracias!**.

Luis E. González Galicia, el más viejo y entrañable de mis amigos, un café más y un reventón más y entonces si dejamos sin café y alcohol al resto de la población, muchos alucines: tantas veces arreglamos el mundo que perdimos la cuenta y por eso tuvimos que volverlo a descomponer, en el inter surgieron miles de temas de tesis (aunque algunos los dejamos inconclusos), mucha risa y alegría. Manuel Sugawara Muro, excelente amigo y compañero, sin él la carrera hubiese sido más pesada y tediosa, además de ser un trabajador excelso, una relación con altos y bajos, como muchas, pero tan valiosa como todas las demás. Julio Barreiro Guerrero, otro exclente amigo, muchísimas experiencias compartidas, y gran parte de la clase de LaTeX2e usada para editar este trabajo fue realizada con su grata compañia y experiencia, además de congresos, trabajo, playa, fiesta, $\cdots$. Ivan Hernández Serrano, gran amigo, más fidelidad y compañerismo difícilmente alcanzable, buena platica y mucha diversión, mucha. Lo mismo se puede aplicar al buen Jorge Hernández Serrano, hermano del anterior. Arturo Vázquez Corona, super *brother*, horas y horas de sana diversión programando cosas que nadie nos pidió y que a nadie más le interesan, rompecabezas, café, etc. Paola Ramírez Pulido, sí, así es: cuñada favorita, pero no sólo por ese hecho fortuito, sino por ser una gran amiga y por muchas y placenteras charlas llenas de risa, drama y emoción.

A continuación otros de amigos, con quienes compartí muchos momentos gratos a lo largo de toda la carrera, antes e incluso después (también algunos momentos ingratos como la fatídica huelga de 1999 en la UNAM): Mauricio Plaza Villegas (Mok), un buen amigo, super divertido, también tomé una curso de verano con él. Carlos A. Sánchez M. (el buen Charly), muchas pláticas interesantes, una que otra fiesta, un chorro de cursos juntos. Todos los cálculos, todas las fiestas de fin de año desde hace ya varios años y muchas, muchas horas de pláticas divertidas: Sandra Macotela B., Armando Ricalde (Doc), Ricardo Pérez (Rick), Erick Morales, Alejandra, Edmundo. Patricia Pellicer C. primero ayudante y luego *roomie* desde hace ya un rato, con quien también he platicado muchísimo de política, lugares, matemáticas, etc. A algunos de mis maestros de la carrera, de quienes me llevo el mejor concepto y unas cuantas cosas que espero aplicar el resto de mi vida, a continuación menciono a algunos que recordaré siempre con especial cariño: Margarita Chávez Cano, muy tierna, mi jefa por poco tiempo en la división, y una clase de probabilidad y estadística muy padre. José Alfredo Amor Montaño, verdugo de mi promedio y al mismo tiempo uno de mis mejores maestros: gracias. Lourdes Velasco Arregui, por cuatro cálculos que no se olvidan fácilmente y menos con tanto apapacho de la maestra-casi-mamá de grupos enormes: ¡Qué paciencia! —tengo que aclarar que el "cálculo" si lo pienso olvidar, de hecho ya empecé a hacerlo hace un rato, lo que no voy a olvidar son esos dos años "de cálculo". Hugo Rincón, el álgebra moderna más veloz en el continente americano, excelente profesor.

A todos los que olvidé, le pido una disculpa y les ruego acepten este: "gracias".

Capítulo 1

Antecedentes

1.1 Jerarquía de memoria

Los pioneros de la computación predijeron correctamente que los programadores desearían cantidades ilimitadas de memoria. Una solución económica para satisfacer ese deseo es una memoria jerárquica, que tome ventaja de la *localidad* y la relación costo–desempeño de la tecnología moderna de los sistemas de memoria. El principio de localidad dice que los programas no acceden a todo el código o los datos de manera uniforme. En estudios realizados en los últimos años se muestra que los programas tienden a reutilizar datos e instrucciones recientemente utilizadas. Como regla general, un programa consume el 90% de su tiempo total de ejecución en solamente el 10% de su código. Una implicación del principio de localidad es que podemos predecir con relativa exactitud qué instrucciones y datos usará un programa en el futuro cercano basándonos en los accesos realizados en el pasado cercano.

Este principio y la tendencia moderna en las arquitecturas de las computadoras –el hardware más pequeño es más rápido– han empujado la tecnología a una jerarquía basada en memorias de diferentes velocidades y tamaños. Dado que la memoria rápida es cara, la jerarquía de memoria está organizada en varios niveles –cada nivel, más pequeño, rápido y más caro por unidad de almacenamiento (*byte*) que el siguiente nivel. La meta final es tener un sistema de memoria con un costo casi tan barato como el costo de la memoria más barata y tan rápido como el sistema más rápido. Los niveles de la jerarquía usualmente son subconjuntos unos de otros; toda la información en un nivel se encuentra también en el nivel anterior, y la información en ese nivel más bajo está en el que está debajo de él, y así sucesivamente hasta llegar al fondo de la jerarquía. Cada nivel transforma direcciones de una memoria más grande a una memoria más pequeña pero más rápida en la jerarquía. Como parte de la transformación de direcciones la jerarquía de memoria tiene la responsabilidad de revisar las direcciones; por lo que una parte fundamental de la jerarquía de memoria es un esquema de protección para escrutinar direcciones.

La importancia de la jerarquía de memoria se ha incrementado con los avances en la tecnología de los procesadores[1], los cuales han mejorado su desempeño 55% por año desde

[1] Utilizaremos el término procesador o CPU para referirnos a la Unidad Central de Procesamiento de un sistema de cómputo

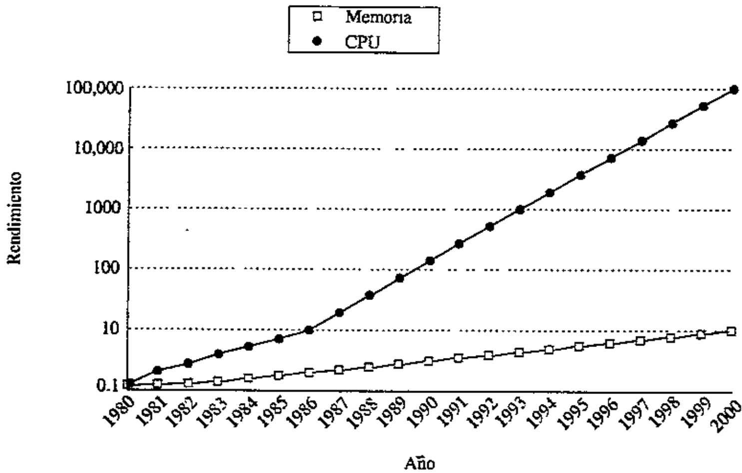


Figura 1.1: Rendimiento comparativo de **memoria** y **procesador** (tomando el año 1980 como base). La memoria base contemplada es de 64KB (DRAM), con tres años para la siguiente generación y una mejora del 7% por año. La línea del procesador supone una mejora de 1.35 por año hasta 1996, y de 1.55 después. Nótese que el eje vertical tiene una escala logarítmica para hacer visible la brecha entre el rendimiento del procesador y la memoria.

1987, y 35% por año hasta antes de 1986. Desgraciadamente, no podemos decir lo mismo del aumento en el desempeño de la memoria en los sistemas de cómputo[2]; por lo cual, la brecha entre el avance de los procesadores y la memoria es más grande año con año, como se puede ver en la figura 1.1. En un sistema de cómputo, la velocidad final está dada por el componente más lento del sistema[Patterson and Hennessy, 1996].

El *principio de localidad de referencia* dice que la información más usada recientemente muy probablemente será utilizada en un futuro cercano. Hacer el caso común rápidamente sugiere que favorecer el acceso a tal información mejorará el rendimiento. Por lo tanto, debemos tratar de mantener los datos recientemente accedidos en la memoria más rápida. En la figura 1.2 se muestra una jerarquía de memoria típica, con tamaños y velocidades de acceso.

Los niveles en los que se descompone el sistema de memoria de la mayoría de los sistemas modernos de computación es:

Cache. Este es el nombre que recibe generalmente el primer nivel en la jerarquía de la memoria, que es alcanzado en cuanto una dirección sale del procesador. Su capacidad de almacenamiento es reducida y opera a velocidades de procesador. La memoria cache es usada por el procesador para instrucciones y datos durante la ejecución de un programa. Es controlada por la computadora misma y es la más costosa.

[2]Utilizaremos los términos: sistema de cómputo, computadora y máquina como sinónimos.

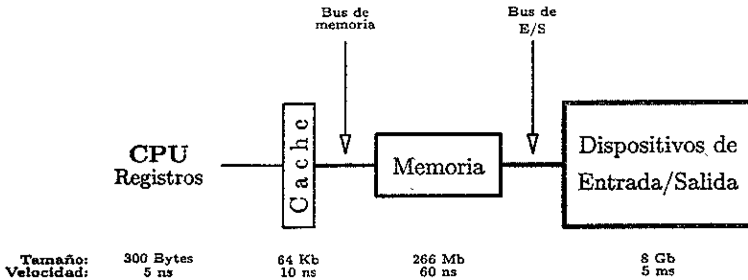


Figura 1.2: **niveles en una jerarquía de memoria** típica. Entre más nos alejamos del procesador, la capacidad de la memoria crece, su velocidad de acceso disminuye lo mismo que su costo. Los tamaños y velocidades de acceso hacen referencia a una computadora de escritorio de mediana escala.

Memoria principal. Esta memoria constituye el siguiente nivel en la jerarquía. Es el destino de las entradas y salidas del sistema. Por lo general, esta memoria se encuentra fuera del procesador, conectada a éste por medio de un conjunto de líneas que son usados para transmitir datos, llamado *bus*. Es conocida como memoria de acceso directo; es decir, el acceso de una celda es independiente del acceso inmediato anterior y no tiene que ser a celdas contiguas (*RAM*, Random Access Memory). Es controlada por el sistema operativo

Memoria virtual. En cualquier momento en el tiempo, los sistemas de cómputo pueden ejecutar varios procesos, cada uno con su propio espacio de direcciones Sin embargo, sería muy costoso dedicar un espacio considerable de memoria a cada proceso, especialmente porque muchos procesos sólo utilizan una pequeña parte de dicho espacio. Por lo tanto, debe existir alguna forma de *compartir* un tamaño menor de memoria física entre varios procesos. Una manera de lograr esto, es usando *memoria virtual*, que divide la memoria física en bloques y los asigna a distintos procesos. La mayoría de los esquemas para implementar memoria virtual también reducen el tiempo de arranque de un programa, ya que no todo el código y datos necesitan estar presentes en la memoria principal para que el programa pueda iniciar su ejecución

A pesar de que la memoria virtual es esencial en todos los sistemas de cómputo modernos, compartir no es la razón por la que se inventa la memoria virtual. Si un programa era muy grande para entrar en la memoria física, era responsabilidad del programador lograr que cupiera, dividiendo el programa en varias partes para después identificar aquéllas que no tienen dependencias o referencias entre sí, y por último cargando o liberando la memoria que éstas ocupaban durante la ejecución del programa según fuera posible. Era responsabilidad del programador asegurar que el programa no intentara acceder más memoria de la que físicamente existía en el sistema y que la sección correcta del programa estuviera en memoria cuando era requerida. Es claro que esta responsabilidad disminuye la productividad de un programador. La memoria virtual

fue inventada para liberar al programador de esta carga, manejando automáticamente dos niveles de la jerarquía de memoria, la memoria principal y la memoria secundaria. La memoria virtual es manejada por el sistema operativo.

El sistema de memoria virtual, generalmente mantiene un bit por página que indica si la página ha sido "ensuciada" (cambiada de alguna forma) desde la última vez que fue escrita al disco, éste conjunto de bits es conocido como *dirty bits*.

Memoria secundaria o almacenamiento en disco. Esta memoria constituye, en la gran mayoría de los sistemas modernos, el último nivel en la jerarquía. Es de gran tamaño y el acceso a la información guardada en ella es varios *órdenes de magnitud* más lento comparado con el acceso a la información en la memoria principal. El dispositivo en que se implementa este tipo de memoria es, por lo general, un disco magnético. Este tipo de memoria es manejada por el sistema operativo y por el usuario de la computadora. La sección de almacenamiento en disco que es manejada por el sistema operativo recibe el nombre de área de intercambio (*swap*), el resto es usado como área de almacenamiento permanente.

Cada uno de los niveles mencionados se subdivide a su vez en varias partes que varían de una arquitectura a otra, pero un análisis más detallado sobre la jerarquía de memoria en un sistema de cómputo escapa del ámbito del presente trabajo.

Dos niveles de suma importancia en la jerarquía de memoria son el cache y la memoria virtual. El cache, como hemos explicado es una memoria muy rápida que se encuentra muy cerca del CPU y contiene los datos o el código accedido más recientemente. Cuando el CPU encuentra un dato solicitado en el cache, se dice que hubo éxito (*cache hit*). Cuando el CPU no encuentra un dato en el cache, es una falla de cache (*cache miss*). Un bloque de datos de tamaño fijo, llamado *bloque*, que contiene la palabra solicitada es extraído de la memoria principal y colocado en el cache. Es una buena idea ponerlo en el cache, porque la probabilidad de usar esta palabra otra vez en el futuro es muy alta. Además, otro principio, conocido como el *principio de localidad espacial*, nos dice que hay una probabilidad alta de que datos cercanos sean usados en tiempos cercanos también. Por lo tanto, las demás palabras en el bloque recién insertado en el cache tienen una probabilidad alta de ser usadas pronto.

No todos los objetos referenciados por un programa deben residir en la memoria principal. Si la computadora tiene memoria virtual, algunos objetos pueden residir en disco. El espacio de direcciones, en este caso, es dividido en bloques de tamaño fijo, llamados *páginas*. En cualquier momento, cada página puede residir ya sea en la memoria principal o en disco. Cuando el CPU hace referencia a un objeto de una página que no está presente en el cache o en la memoria principal, ocurre una falta de página (*page fault*), y la página completa se mueve del disco a la memoria principal. Dado que las faltas de páginas tardan mucho tiempo, son solucionadas por medio de software y el CPU no es detenido; mientras que, cuando sucede una falta de cache, estas son manejas por medio de hardware y –generalmente– el procesador es detenido mientras el bloque es movido al cache.

El cache y la memoria principal guardan la misma relación que la memoria principal y el disco.

1.2 Conjuntos de instrucciones

La interacción entre compiladores y lenguajes de alto nivel afecta significativamente cómo usan los programas el conjunto de instrucciones de una arquitectura de computadora dada. Existen dos preguntas importantes: ¿Cómo se asigna memoria a una variable y cómo es referida dicha variable? ¿Cuántos registros se necesitan para colocar en memoria un cierto número de variables? Para poder contestar a estas preguntas, analicemos primero algunos de los principios que siguen los conjuntos de instrucciones de un procesador, y cuál es su relación con los compiladores de lenguajes de alto nivel.

Casi todas las máquinas diseñadas después de 1980 utilizan arquitecturas con registros de propósito general para cargar y almacenar. Anteriormente, existían otros dos tipos de arquitecturas, donde las operaciones se ejecutaban en un stack o en un acumulador. Esta distinción entre arquitecturas es muy importante, ya que nos sirve para clasificar los conjuntos de instrucciones y el tipo de almacenamiento interno que tiene un determinado CPU. Por ejemplo, en las máquinas de stack, los operandos para las instrucciones que acepta el procesador se encuentran *implícitos* en el tope del stack, mientras que en las arquitecturas con un acumulador, un operando implícito es el acumulador mismo. Por otro lado, en las máquinas con registros, los operandos siempre son *explícitos* –ya sean registros o localidades de memoria. A partir de este punto consideraremos únicamente arquitecturas con registros de propósito general, ya que son –prácticamente– el único tipo de máquinas usado hoy en día.

Lo primero que debemos notar es que los registros son más rápidos que la memoria (porque están dentro del procesador). En segundo lugar, es más fácil para un compilador utilizar los registros y de manera más efectiva que otro tipo de almacenamiento interno, como el cache que es –prácticamente– imposible controlar desde un compilador. Por ejemplo, la expresión $(A \times B) - (C \times D) - (E \times F)$ puede evaluarse haciendo las operaciones en cualquier orden, lo que puede ser más eficiente debido a la localización de los operandos y cuestiones relacionadas con *pipelining*[3].

Lo más importante acerca de los registros es que éstos pueden ser utilizados para guardar variables. Cuando se asignan variables a los registros, el tráfico en la memoria disminuye, el programa se ejecuta más rápido (porque los registros son más rápidos que la memoria) y la densidad del código se reduce también (dado que un registro puede ser nombrado con menos bits que una localidad de memoria).

¿Cuántos registros son suficientes? La respuesta depende de cómo los usa el compilador. La mayoría de los compiladores reservan algunos registros para evaluación de expresiones, algunos más para paso de parámetros y los demás son utilizados para guardar variables. Es común en arquitecturas modernas tener 32 registros.

---

[3]Pipelining es una técnica de implementación que permite realizar varias operaciones en paralelo en tiempo de ejecución Hoy en día, la técnica clave para hacer procesadores más rápidos es, justamente, pipelining

Un pipeline es como una línea de ensamblaje En una línea de ensamblado de automóviles, hay varios pasos, cada uno contribuye su parte para la construcción de un automóvil. Cada paso opera en paralelo con respecto a los demás, pero en diferentes automóviles En un pipeline de una computadora, cada paso en el pipeline lleva a cabo una parte de una instrucción Al igual que en una línea de ensamblado, los distintos pasos hacen partes distintas de operaciones distintas .Cada uno de estos pasos es conocido como *pipe stage* o *pipe segment*

¿Cómo se interpreta una dirección de memoria? Es decir, ¿qué objeto se accede como función de la dirección y la longitud? Independientemente de cuantos registros tenga una arquitectura dada y cómo son utilizados, debe definir cómo se interpreta y especifica una dirección de memoria. La mayoría de las arquitecturas modernas, utilizan dirección en bytes y proveen acceso para bytes (8 bits), medias palabras (half words, 16 bits) y palabras (words, 32 bits). La mayoría también proveen una forma para acceder palabras dobles (double words, 64 bits).

Casi toda la programación que se hace hoy en día es en lenguajes de alto nivel. Esto significa que la mayoría de instrucciones ejecutadas por un procesador son la salida de un compilador, por lo tanto, el conjunto de instrucciones de la arquitectura es en esencia el blanco de un compilador. Esto crea una dependencia directa entre el compilador y la eficiencia de la arquitectura, por lo que podemos afirmar que la implementación de los conjuntos de instrucciones en arquitecturas recientes, está basado en un buen entendimiento de la tecnología usada en compiladores. Por tal motivo, en el resto del presente trabajo supondremos que trabajamos con un conjunto de instrucciones de máquina, cuyos tres componentes principales son ortogonales: las operaciones, los tipos de datos y los tipos de direccionamiento. También supondremos que el número de registros es estándar y que la forma de interpretar las direcciones de memoria es justamente como hemos explicado en esta sección. Esto nos provee de una base sólida para enfocarnos en el manejo de la asignación de memoria desde el punto de vista de los lenguajes de alto nivel.

1.2.1 La estructura de compiladores recientes

La meta principal del creador de un compilador es la corrección –todo programa válido debe ser compilado correctamente. La segunda meta es usualmente la velocidad del código compilado. Típicamente, sigue un conjunto de metas, como rapidez en la compilación, soporte para depuración, etc. Normalmente, las pasadas que efectúa el compilador transforman representaciones de alto nivel, abstractas, en representaciones progresivas de más bajo nivel, hasta que el conjunto de instrucciones de la máquina se alcanza. Esta estructura sirve para manejar la complejidad de las transformaciones y permite escribir un compilador libre de errores fácilmente.

La complejidad de escribir un compilador es la mayor limitación para el número de optimizaciones que pueden hacerse. La estructura ayuda a reducir la complejidad del compilador, pero esto significa que el compilador debe ordenar y realizar algunas transformaciones antes que otras. En el diagrama del compilador con optimización en la Figura 1.3, podemos apreciar que algunas optimizaciones son realizadas mucho antes de saber cómo lucirá el código en detalle. Una vez que dicha transformación se ha llevado a cabo, el compilador no se puede dar el lujo de regresar y revisar los pasos e incluso, deshacer transformaciones previas. Por lo tanto, los compiladores hacen suposiciones acerca de la habilidad de los pasos posteriores para lidiar con algunos problemas. Por ejemplo, usualmente deben decidir cuáles llamadas a procedimientos harán *en línea*, antes de siquiera saber el tamaño del procedimiento que se está llamando. Los escritores de compiladores llaman a este problema el *problema de ordenamiento de fases*, (phase-ordering problem).

¿Cómo interactúan estas transformaciones ordenadas con el conjunto de instrucciones de la arquitectura? Un buen ejemplo ocurre con la optimización llamada eliminación

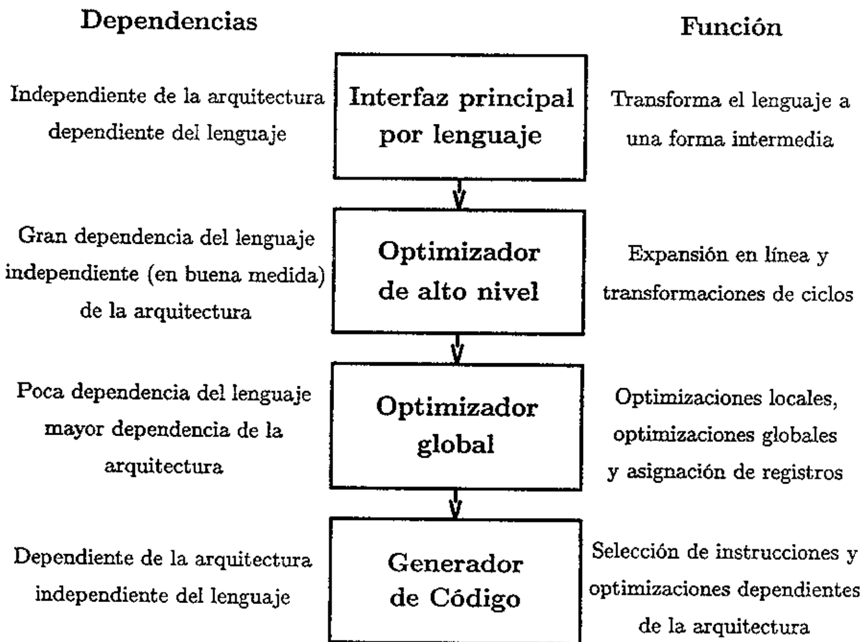


Figura 1.3: Los compiladores modernos consisten, típicamente, de dos a cuatro pasadas. Entre mayor optimización provea el compilador, se requiere un mayor número de pasadas. Una *pasada* es simplemente una fase en la cual el compilador lee y transforma el programa completo. (El término *fase* se usa como sinónimo de *pasada*.) Las fases de optimización son opcionales y pueden ser evitadas cuando se requiere una compilación muy rápida y es aceptable código de baja calidad. Esta estructura maximiza la probabilidad de que un programa compilado a distintos niveles de optimización produzca la misma salida dada la misma entrada Varios lenguajes pueden utilizar las mismas fases de optimización y generación de código, ya que las fases de optimización están separadas, (sólo se requiere una nueva presentación o capa externa para soportar un nuevo lenguaje). La *expansión en línea de procedimientos*, mencionada en las optimizaciones de alto nivel, también es llamada *integración de procedimientos*

de subexpresiones globales comunes (*global common subexpression elimination*). Esta optimización encuentra dos instancias de una expresión que calculan el mismo valor y salva el valor del primer cálculo en un temporal Después utiliza el valor almacenado en el temporal, eliminando el segundo cálculo de la expresión Para que esta optimización sea significativa, el temporal debe ser guardado en un registro De otra forma, el costo de guardar el temporal en la memoria y después cargarlo puede aminorar la ganancia obtenida al no tener que recalcular el valor Hay, de hecho, casos en los que esta optimizacion hace mas lenta la ejecución del código cuando el temporal no se almacena en un registro El orden de las fases

complica aún más esta situación, ya que la asignación a los registros se hace típicamente cerca del final de la pasada de optimización global, justo antes de la generación de código. Por lo tanto, un compilador que realice esta optimización *debe* suponer que el asignador de registros en verdad asignará este temporal a un registro.

Las optimizaciones que llevan a cabo los compiladores modernos pueden ser clasificadas por el *estilo* de la transformación como sigue:

Optimizaciones de alto nivel: Son realizadas sobre el código fuente y el resultado es pasado a otras fases de optimización.

Optimizaciones locales: Éstas optimizan un fragmento de código en un línea directa, (llamado *bloque básico* en el argot de compiladores).

Optimizaciones globales: Extienden las optimizaciones locales para trabajar entre varias ramas del código e introduce un conjunto de transformaciones para optimizar ciclos.

Asignación a registros: La asignación a los registros se considera la optimización más importante, debido a que muchas optimizaciones (locales y globales) dependen de estas asignaciones. El ideal es asignar el 100% de las variables vivas a los registros.

Optimizaciones dependientes de la arquitectura: Aquí se trata de obtener la máxima ventaja del conocimiento que tiene el diseñador del compilador sobre la arquitectura específica.

Debido al papel central que juega la asignación a los registros, tanto en acelerar la ejecución del código como en hacer útiles algunas optimizaciones, es una de las optimizaciones más importantes (si no es que la más importante). En la actualidad la asignación a los registros está basada en una técnica conocida como *coloración de gráficas*. La idea central detrás de la coloración de gráficas es construir una gráfica que representa a los posibles candidatos para asignar a los registros y entonces utilizar la gráfica para asignar los registros. El problema de colorear una gráfica es *NP-completo*; sin embargo, hay heurísticas que funcionan bien en la práctica.

La coloración de gráficas funciona mejor cuando hay al menos 16 registros de propósito general (y preferente más) disponibles para asignación global de variables enteras y registros adicionales para variables de punto flotante. Desafortunadamente, la coloración de gráficas no funciona muy bien cuando el número de registros es pequeño, porque la heurística utilizada en la mayoría de los algoritmos depende de ese número mínimo de registros, y con una muy alta probabilidad fallarán. El énfasis en este enfoque está puesto en asignar a los registros el 100% de las variables activas.

1.2.2 Asignación de memoria en lenguajes de alto nivel

Las distintas áreas en las cuales los lenguajes de alto nivel modernos asignan memoria a sus datos, se muestran a continuación:

Pila (stack). Éste es usado para asignar las variables locales. El stack se agranda cada vez que existe una *llamada* a una función y se recorta cuando una función *regresa*.

Los objetos en el stack son referenciados por medio del apuntador del stack y son principalmente variables escalares (variables simples) y no arreglos. El stack se utiliza para registros de activación y no como un stack para evaluar expresiones.

Área global de datos. Se usa para alojar objetos declarados estáticamente, tales como variables y constantes. Un gran porcentaje de estos objetos son arreglos y otro tipo de estructuras de datos complejas.

Espacio de memoria dinámica (heap). Es usado para alojar objetos dinámicos que no se adhieren a una disciplina de stack. Los objetos en el heap son accedidos por medio de referencias (en algunos lenguajes llamadas apuntadores) y por lo general estos objetos no son escalares, sino estructurados.

La asignación en los registros de una computadora es mucho más eficiente para objetos que se pueden almacenar en el stack que para variables globales. Los objetos que son alojados en el heap es imposible que puedan ser asignados a los registros, por su estructura y dado que son accedidos por medio de referencias. Las variables globales y algunas variables del stack es imposible introducirlas a los registros, ya que pueden ser *seudónimos*, lo que significa que hay varias formas de hacer referencia a la dirección en memoria de una misma variable, y esto hace ilegal colocarla en un registro (en la tecnología de compiladores de hoy en día, prácticamente todas las variables heap son seudónimos). Por ejemplo, considere la siguiente secuencia de código, donde & regresa la dirección de una variable y * regresa el valor almacenado en una referencia:

p = & a	– Almacena la dirección de a en p
a = ...	– Asigna directamente algo a la variable a
*p = ...	– Utiliza p para asignar nuevamente a a
...a...	– Accede la variable a

La variable a no podría ser asignada a un registro y sobrevivir la asignación a *p sin generar código incorrecto. El hecho de que un lenguaje de alto nivel soporte seudónimos para variables, introduce un problema substancial, ya que es muy difícil, y en algunos casos imposible, decidir para cuáles objetos hay referencias vivas. Un compilador debe ser conservador; muchos compiladores modernos no asignarían ninguna variable local a un registro, cuando hay una referencia que puede apuntar a *una* variable local. Por este motivo, el uso del stack y del heap es muy importante para mantener la ejecución del programa tan eficiente como sea posible.

1.3 Recolección de basura

El término *recolección de basura*[4] se refiere a la recuperación automática del espacio en memoria de un sistema de cómputo. Mientras que en muchos sistemas, es directamente

[4] *Recolección de basura* (*garbage collection*) a lo largo de este trabajo será utilizado en el sentido amplio: esto es, incluyendo a todos los métodos conocidos de recolección de basura, antiguos y modernos. En la literatura, los métodos antiguos sí se conocen como recolectores de basura, pero a la gran mayoría de los métodos modernos les llaman *recuperación automática de almacenamiento* (*automatic storage reclamation*)

el programador quien debe explícitamente recuperar memoria del *heap* en algún momento del programa –por medio de comandos del lenguaje mismo como free o dispose– en los sistemas que cuentan con un recolector de basura, se libera al programador de esta pesada carga. La función del recolector de basura es encontrar los objetos[5] de datos que ya no se están utilizando y hacer su espacio disponible para reutilización del programa en ejecución. Un objeto es considerado basura (y sujeto entonces a recuperación) si no es alcanzable por medio de recorrer alguna sucesión de ligas por el programa en ejecución. Los objetos *vivos* (potencialmente alcanzables) son preservados por el recolector, asegurando que el programa nunca recorrerá una liga que lo lleve a un espacio que ya no corresponde al objeto solicitado (*dangling pointer*).

La recolección de basura es necesaria para llevar a cabo una programación completa modular y evitar introducir dependencias innecesarias entre módulos. Una rutina que opera sobre una estructura de datos no necesita saber si otras rutinas están operando sobre la misma estructura, a menos que exista una buena razón para coordinar sus actividades. En los sistemas en los que los objetos tienen que ser liberados explícitamente, algún módulo debe ser responsable de conocer cuando *otros* módulos no están interesados en un objeto particular, rompiendo con los esquemas de encapsulamiento modular.

Dado que *estar vivo* es una propiedad global, esto introduce la necesidad de incluir un registro sistemático no local, en rutinas que de otra forma podrían ser ortogonales, y evita el que puedan usarse para componer varios sistemas simultáneamente, ya que contienen particularidades del sistema para el que fueron creadas. El mantenimiento de este registro por parte de la rutina puede reducir la extensibilidad, ya que cada vez que se agregan nuevas funcionalidades al sistema, también debe modificarse el código que mantiene el registro en el sistema en cada una de las rutinas.

Las complicaciones innecesarias creadas por la asignación explícita de almacenamiento son especialmente problemáticas, ya que los errores en la programación frecuentemente introducen un comportamiento erróneo que rompe las abstracciones básicas del lenguaje de programación, haciendo que los errores sean difíciles de diagnosticar.

Si no se recupera la memoria en un momento preciso puede provocar pequeñas fugas de memoria, con memoria no recuperada acumulándose gradualmente hasta que el proceso termina o el espacio de intercambio en disco (*swap*) se agota. Si se recupera memoria de manera apresurada, esto puede llevar al sistema a tener un comportamiento muy extraño, porque el espacio que ocupa un objeto puede utilizarse para almacenar un objeto totalmente distinto mientras un apuntador viejo aún existe. La misma localidad de memoria sería entonces interpretada como dos objetos distintos simultáneamente, con modificaciones a uno que introducirían mutaciones impredecibles en el otro.

Estos errores en un sistema son muy peligrosos ya que por lo general no se repiten, haciendo el proceso de depuración muy difícil; incluso, este tipo de errores no se presentan sino hasta que el programa es usado de manera muy intensa. Si el programa asignador de espacio en memoria (*allocator*) no reutiliza el espacio liberado por un objeto, un apuntador colgante (*dangling pointer*) puede no causar ningún problema. Más tarde, cuando ya se

[5]Utilizo el término *objeto* de manera amplia, para incluir cualquier clase de estructura de datos, tales como los creados por instrucciones record de Pascal o struct de C, pero también para incluir objetos en un sentido más estricto, objetos con encapsulamiento y herencia; esto es, en el sentido de la programación orientada a objetos.

encuentra el programa corriendo dentro de un programa de aplicación, el programa de aplicación completo puede fallar mientras hace un conjunto distinto de solicitudes de memoria, o es ligado con otro programa de asignación de espacio en memoria. Una fuga de memoria puede pasar desapercibida en condiciones normales de trabajo —tal vez, por varios años— ya que la ejecución del programa termina antes de que se haya utilizado mucho espacio extra. Pero si el código es incorporado en un programa que tenga corridas muy largas, el programa eventualmente agotará su espacio de swap y se colapsará.

La asignación y recuperación explícita de memoria lleva a errores de programación en formas más sutiles. Es común para los programadores asignar estáticamente un número moderado de objetos, para que sea innecesario asignarles memoria en el heap y decidir cuándo y dónde recuperarlos. Esto conlleva a limitaciones fijas en el software, haciéndolos fallar cuando esas limitaciones son excedidas, posiblemente años más tarde cuando la memoria (y conjuntos de datos) sean más grandes. Esta fragilidad hace que el código sea poco reutilizable, ya que las limitaciones no documentadas lo harán fallar, aún cuando el programa sea utilizado en una forma consistente con su abstracción.

Estos problemas provocan que muchos programadores (de programas de aplicación) implementen alguna forma de recolector de basura específico para su aplicación, para evitar muchos de los dolores de cabeza provocados por los manejadores de asignación explícita de memoria. Desafortunadamente, este tipo de recolectores son frecuentemente incompletos y están llenos de errores, ya que fueron diseñados para una aplicación específica. Esto hace que sean poco confiables y difíciles de utilizar ya que no están integrados en el lenguaje de programación. Sin embargo, el hecho de que existan a pesar de los problemas que introducen es testimonio de lo valioso de un recolector de basura, y sugiere que la recolección de basura debe ser parte de las implementaciones de lenguajes de programación.

Este trabajo es justamente acerca de este último tipo de recolectores de basura, los que están integrados en la implementación del lenguaje de programación. La forma acordada para su uso es que las rutinas de asignación del lenguaje (o importadas de una biblioteca) realicen acciones especiales para recuperar memoria, conforme sea necesario; es decir, cuando una solicitud de memoria no pueda ser satisfecha fácilmente. Esto hace que llamadas explícitas a un recuperador de memoria sean innecesarias, ya que son implícitas en cada llamada al asignador de memoria.

La gran mayoría de los recolectores de basura, como ya mencionamos, requieren de mucha cooperación por parte del compilador[6](o intérprete), así como reconocer las formas de los objetos de datos. Dependiendo de los detalles del recolector de basura, se pueden requerir cambios al generador de código, para que emita cierta información en tiempo de compilación, y tal vez para que ejecute distintos conjuntos de instrucciones en tiempo de ejecución.

En el siguiente capítulo describiremos las fases principales de un recolector de basura, la representación de objetos y haremos un recorrido de las técnicas básicas para recolectar basura.

[6]Contrario a la muy esparcida creencia, no hay ningún conflicto entre usar un lenguaje compilado y un recolector de basura, implementaciones de vanguardia (*state-of-the-art*) de lenguajes de programación con recolección de basura utilizan sofisticados compiladores con optimización.

Los objetos en el stack son referenciados por medio del apuntador del stack y son principalmente variables escalares (variables simples) y no arreglos. El stack se utiliza para registros de activación y no como un stack para evaluar expresiones.

Área global de datos. Se usa para alojar objetos declarados estáticamente, tales como variables y constantes. Un gran porcentaje de estos objetos son arreglos y otro tipo de estructuras de datos complejas.

Espacio de memoria dinámica (heap). Es usado para alojar objetos dinámicos que no se adhieren a una disciplina de stack. Los objetos en el heap son accedidos por medio de referencias (en algunos lenguajes llamadas apuntadores) y por lo general estos objetos no son escalares, sino estructurados.

La asignación en los registros de una computadora es mucho más eficiente para objetos que se pueden almacenar en el stack que para variables globales. Los objetos que son alojados en el heap es imposible que puedan ser asignados a los registros, por su estructura y dado que son accedidos por medio de referencias. Las variables globales y algunas variables del stack es imposible introducirlas a los registros, ya que pueden ser *seudónimos*, lo que significa que hay varias formas de hacer referencia a la dirección en memoria de una misma variable, y esto hace ilegal colocarla en un registro (en la tecnología de compiladores de hoy en día, prácticamente todas las variables heap son seudónimos). Por ejemplo, considere la siguiente secuencia de código, donde & regresa la dirección de una variable y * regresa el valor almacenado en una referencia:

p = & a	– Almacena la dirección de a en p
a = ...	– Asigna directamente algo a la variable a
*p = ...	– Utiliza p para asignar nuevamente a a
...a...	– Accede la variable a

La variable a no podría ser asignada a un registro y sobrevivir la asignación a *p sin generar código incorrecto. El hecho de que un lenguaje de alto nivel soporte seudónimos para variables, introduce un problema substancial, ya que es muy difícil, y en algunos casos imposible, decidir para cuáles objetos hay referencias vivas. Un compilador debe ser conservador; muchos compiladores modernos no asignarían ninguna variable local a un registro, cuando hay una referencia que puede apuntar a *una* variable local. Por este motivo, el uso del stack y del heap es muy importante para mantener la ejecución del programa tan eficiente como sea posible.

1.3 Recolección de basura

El término *recolección de basura*[4] se refiere a la recuperación automática del espacio en memoria de un sistema de cómputo. Mientras que en muchos sistemas, es directamente

[4] *Recolección de basura* (*garbage collection*) a lo largo de este trabajo será utilizado en el sentido amplio; esto es, incluyendo a todos los métodos conocidos de recolección de basura, antiguos y modernos En la literatura, los métodos antiguos sí se conocen como recolectores de basura, pero a la gran mayoría de los métodos modernos les llaman *recuperación automática de almacenamiento* (*automatic storage reclamation*)

el programador quien debe explícitamente recuperar memoria del *heap* en algún momento del programa –por medio de comandos del lenguaje mismo como `free` o `dispose`– en los sistemas que cuentan con un recolector de basura, se libera al programador de esta pesada carga. La función del recolector de basura es encontrar los objetos[5] de datos que ya no se están utilizando y hacer su espacio disponible para reutilización del programa en ejecución. Un objeto es considerado basura (y sujeto entonces a recuperación) si no es alcanzable por medio de recorrer alguna sucesión de ligas por el programa en ejecución. Los objetos *vivos* (potencialmente alcanzables) son preservados por el recolector, asegurando que el programa nunca recorrerá una liga que lo lleve a un espacio que ya no corresponde al objeto solicitado (*dangling pointer*).

La recolección de basura es necesaria para llevar a cabo una programación completa modular y evitar introducir dependencias innecesarias entre módulos. Una rutina que opera sobre una estructura de datos no necesita saber si otras rutinas están operando sobre la misma estructura, a menos que exista una buena razón para coordinar sus actividades. En los sistemas en los que los objetos tienen que ser liberados explícitamente, algún módulo debe ser responsable de conocer cuando *otros* módulos no están interesados en un objeto particular, rompiendo con los esquemas de encapsulamiento modular.

Dado que *estar vivo* es una propiedad global, esto introduce la necesidad de incluir un registro sistemático no local, en rutinas que de otra forma podrían ser ortogonales, y evita el que puedan usarse para componer varios sistemas simultáneamente, ya que contienen particularidades del sistema para el que fueron creadas. El mantenimiento de este registro por parte de la rutina puede reducir la extensibilidad, ya que cada vez que se agregan nuevas funcionalidades al sistema, también debe modificarse el código que mantiene el registro en el sistema en cada una de las rutinas.

Las complicaciones innecesarias creadas por la asignación explícita de almacenamiento son especialmente problemáticas, ya que los errores en la programación frecuentemente introducen un comportamiento erróneo que rompe las abstracciones básicas del lenguaje de programación, haciendo que los errores sean difíciles de diagnosticar.

Si no se recupera la memoria en un momento preciso puede provocar pequeñas fugas de memoria, con memoria no recuperada acumulándose gradualmente hasta que el proceso termina o el espacio de intercambio en disco (*swap*) se agota. Si se recupera memoria de manera apresurada, esto puede llevar al sistema a tener un comportamiento muy extraño, porque el espacio que ocupa un objeto puede utilizarse para almacenar un objeto totalmente distinto mientras un apuntador viejo aún existe. La misma localidad de memoria sería entonces interpretada como dos objetos distintos simultáneamente, con modificaciones a uno que introducirían mutaciones impredecibles en el otro.

Estos errores en un sistema son muy peligrosos ya que por lo general no se repiten, haciendo el proceso de depuración muy difícil; incluso, este tipo de errores no se presentan sino hasta que el programa es usado de manera muy intensa. Si el programa asignador de espacio en memoria (*allocator*) no reutiliza el espacio liberado por un objeto, un apuntador colgante (*dangling pointer*) puede no causar ningún problema. Más tarde, cuando ya se

[5]Utilizo el término *objeto* de manera amplia, para incluir cualquier clase de estructura de datos, tales como los creados por instrucciones `record` de Pascal o `struct` de C, pero también para incluir objetos en un sentido más estricto, objetos con encapsulamiento y herencia; esto es, en el sentido de la programación orientada a objetos.

encuentra el programa corriendo dentro de un programa de aplicación, el programa de aplicación completo puede fallar mientras hace un conjunto distinto de solicitudes de memoria, o es ligado con otro programa de asignación de espacio en memoria. Una fuga de memoria puede pasar desapercibida en condiciones normales de trabajo —tal vez, por varios años— ya que la ejecución del programa termina antes de que se haya utilizado mucho espacio extra. Pero si el código es incorporado en un programa que tenga corridas muy largas, el programa eventualmente agotará su espacio de swap y se colapsará.

La asignación y recuperación explícita de memoria lleva a errores de programación en formas más sutiles. Es común para los programadores asignar estáticamente un número moderado de objetos, para que sea innecesario asignarles memoria en el heap y decidir cuándo y dónde recuperarlos. Esto conlleva a limitaciones fijas en el software, haciéndolos fallar cuando esas limitaciones son excedidas, posiblemente años más tarde cuando la memoria (y conjuntos de datos) sean más grandes. Esta fragilidad hace que el código sea poco reutilizable, ya que las limitaciones no documentadas lo harán fallar, aún cuando el programa sea utilizado en una forma consistente con su abstracción.

Estos problemas provocan que muchos programadores (de programas de aplicación) implementen alguna forma de recolector de basura específico para su aplicación, para evitar muchos de los dolores de cabeza provocados por los manejadores de asignación explícita de memoria. Desafortunadamente, este tipo de recolectores son frecuentemente incompletos y están llenos de errores, ya que fueron diseñados para una aplicación específica. Esto hace que sean poco confiables y difíciles de utilizar ya que no están integrados en el lenguaje de programación. Sin embargo, el hecho de que existan a pesar de los problemas que introducen es testimonio de lo valioso de un recolector de basura, y sugiere que la recolección de basura debe ser parte de las implementaciones de lenguajes de programación.

Este trabajo es justamente acerca de este último tipo de recolectores de basura, los que están integrados en la implementación del lenguaje de programación. La forma acordada para su uso es que las rutinas de asignación del lenguaje (o importadas de una biblioteca) realicen acciones especiales para recuperar memoria, conforme sea necesario; es decir, cuando una solicitud de memoria no pueda ser satisfecha fácilmente. Esto hace que llamadas explícitas a un recuperador de memoria sean innecesarias, ya que son implícitas en cada llamada al asignador de memoria.

La gran mayoría de los recolectores de basura, como ya mencionamos, requieren de mucha cooperación por parte del compilador[6](o intérprete), así como reconocer las formas de los objetos de datos. Dependiendo de los detalles del recolector de basura, se pueden requerir cambios al generador de código, para que emita cierta información en tiempo de compilación, y tal vez para que ejecute distintos conjuntos de instrucciones en tiempo de ejecución.

En el siguiente capítulo describiremos las fases principales de un recolector de basura, la representación de objetos y haremos un recorrido de las técnicas básicas para recolectar basura

[6]Contrario a la muy esparcida creencia, no hay ningún conflicto entre usar un lenguaje compilado y un recolector de basura, implementaciones de vanguardia (*state-of-the-art*) de lenguajes de programación con recolección de basura utilizan sofisticados compiladores con optimización

Capítulo 2

Recolección de basura

Los recolectores de basura automáticamente recuperan el espacio ocupado por objetos de datos para los que el programa en ejecución ha perdido el acceso. Tales objetos reciben el nombre de *basura*. La labor básica de un recolector de basura consiste, en breve, de dos partes:

(i) Distinguir entre los objetos vivos y los que corresponden a basura de alguna manera. Esta etapa es conocida como *detección de basura*, y

(ii) Recuperar las áreas de almacenamiento usada por los objetos basura, para que el programa en ejecución pueda reutilizarlas. Esta etapa es la que corresponde a la *recolección de basura*.

En la práctica estas dos fases pueden ser funcional o temporalmente intercaladas; además, la técnica utilizada para la recuperación es dependiente, en gran medida, de la técnica utilizada para detección de basura

El criterio para determinar si un objeto está vivo en un determinado momento es, en la mayoría de los sistemas recolectores de basura, más conservador que aquel usado en, por ejemplo, algunos compiladores con optimización, en los cuales un objeto puede ser considerado muerto si se encuentra en un punto en el cual nunca puede volver a ser usado, determinado por los análisis del flujo de control y flujo de datos del programa. Generalmente, un recolector de basura utiliza un criterio más simple, menos dinámico, definido en términos de un *conjunto raíz* y de si un objeto es *alcanzable* desde estas raíces.

La recolección de basura se lleva a cabo cuando el programa en ejecución intenta asignar espacio de almacenamiento a algún objeto, pero no hay suficiente memoria disponible para satisfacer la solicitud. La rutina de asignación de memoria llama a la rutina de recolección de basura para liberar espacio y después asignar la memoria solicitada para un objeto

Cuando la recolección de basura se inicia, todas las variables globales de procedimientos activos se consideran vivas, y también las variables locales de cualquier procedimiento activo. El conjunto raíz consiste de las variables globales, las variables locales en la pila de activación y los registros usados por cualquier procedimiento activo. Todos los objetos alcanzables en el heap desde alguna de estas variables, también son alcanzables y por tanto deben conservarse. Además, como el programa podría seguir ligas desde esos objetos

para alcanzar a otros objetos, todo objeto alcanzable desde un objeto vivo es alcanzable. Por lo tanto, el conjunto de objetos vivos es simplemente el conjunto de los objetos que se encuentran en una trayectoria dirigida (de apuntadores), teniendo como vértice inicial alguno de los objetos en el conjunto raíz.

Cualquier objeto que no sea alcanzable desde el conjunto raíz es basura; es decir, inútil, ya que no hay una secuencia legal de acciones que pudiera tomar el programa y que le permitiera alcanzar al objeto. Por tal motivo, los objetos considerados basura no afectan el curso futuro del cálculo, y su espacio puede ser recuperado.

2.1 Representación de objetos

A lo largo de este capítulo, supondremos por simplicidad que los objetos en el heap son auto–identificables; es decir, que es fácil determinar el tipo de objeto del que se trata en tiempo de ejecución. Algunas implementaciones de lenguajes estáticamente tipificados con recolectores de basura, generalmente tienen campos ocultos (a manera de encabezados), en los cuales se guarda la información del tipo de objeto, que puede ser utilizada para decodificar el formato del objeto mismo. Tal información puede ser generada fácilmente por el compilador, ya que éste necesita dicha información para generar código correcto para hacer referencia a los campos de los objetos.

Los lenguajes dinámicamente tipificados, como *Lisp* y *Smalltalk*, usan referencias *etiquetadas*, que son versiones reducidas de la representación de la dirección de hardware. Lo que hace el compilador es eliminar los bits que representan la dirección y pone en su lugar un campo que sirve para identificar el tipo. Esto permite que objetos pequeños, inmutables (en particular, enteros pequeños) puedan ser representados como secuencia única de bits, que se guarda directamente en la parte de la dirección, en lugar de que los enteros pequeños sean referidos por medio de una dirección (como el resto de los objetos). Esta representación etiquetada soporta tipos polimórficos que pueden contener directamente uno de estos objetos *inmediatos* o una referencia a un objeto en el heap.

Para lenguajes estáticamente tipificados puros, no se necesita guardar ninguna información de tipo en tiempo de ejecución, excepto para los tipos de las variables del conjunto raíz, (esto será discutido en la Sección 6.1). A pesar de esto, en lenguajes estáticamente tipificados, se utilizan encabezados para simplificar la implementación y hacer más directo el acceso a variables en el heap, sin que eso involucre un aumento considerable en el costo —Los sistemas con manejo explícito del heap también utilizan encabezados para los objetos por la misma razón.

Los recolectores de basura con *búsqueda conservadora de referencias* se pueden utilizar con poco o incluso sin cooperación alguna del compilador –ni siquiera requieren saber el tipo de variables nombradas– pero postergaremos la discusión de este tipo de recolectores hasta la Sección 2.2.8.

2.2 Técnicas básicas para recolección de basura

Dadas las dos operaciones básicas de un recolector de basura, podemos dar muchas variantes que cumplan esas funciones. La primera parte, *distinguir* objetos vivos de la

basura, puede hacerse de varias maneras: *contando referencias*, *marcando* o *copiando*[1]. Dado que cada esquema tiene una gran influencia en la segunda parte (*recuperación*) y en técnicas de reutilización, introduciremos las técnicas de recuperación de la memoria conforme se vayan explicando las técnicas para distinguir la basura.

2.2.1 Conteo de referencias

En un sistema con conteo de referencias, cada objeto tiene un contador asociado que indica el número de referencias (ligas) a él. Siempre que se crea una liga al objeto –v.gr. cuando se copia una liga de un lugar a otro por medio de una asignación– el contador del objeto se aumenta en uno. Cuando se elimina una referencia a un objeto, el contador es decrementado (véase la Figura 2.1). La memoria ocupada por un objeto puede ser recuperada cuando la cuenta del objeto sea igual a cero, ya que esto significa que no existen referencias al objeto y que el programa en ejecución no puede alcanzarlo.

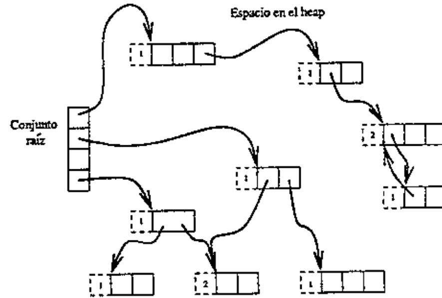


Figura 2.1: Conteo de referencias

En un sistema de conteo de referencias típico, cada objeto tiene un encabezado con la información que describe al objeto, que incluye como dato agregado un contador de referencias. Al igual que otro tipo de información en forma de encabezados, el contador de referencias no es –por lo general– accesible desde el lenguaje.

Cuando un objeto es recuperado por el sistema, se examinan sus campos de referencias, son examinados todos los objetos a los cuales hace referencia y se decrementan sus respectivos contadores, ya que las referencias de objetos basura no tienen injerencia alguna en la determinación de que tan vivo está un objeto. Por ejemplo, si la única referencia a una estructura muy grande se convierte en basura, todos los contadores de referencias de los objetos en la estructura se vuelven cero, por lo que todos los objetos son recuperados.

En términos de las dos fases abstractas en la recolección de basura, el ajuste y chequeo de contadores de referencias implementa la primera fase, y la fase de recuperación cuando un contador se vuelve cero. Ambas operaciones son intercaladas durante la ejecución del programa, ya que pueden ocurrir cuando una referencia es creada o destruida.

Una de las ventajas del conteo de referencias es la naturaleza *incremental* de sus operaciones –el trabajo de recolectar la basura (actualizar los contadores de referencias) está íntimamente relacionado con la propia ejecución del programa. Puede implementarse incremental y en *tiempo real*; esto es, realizando a lo más una pequeña cantidad delimitada de trabajo por unidad de tiempo de ejecución.

Claramente, los ajustes de contadores "regulares" son intrínsecamente incrementales, ya que a lo más generan unas cuantas operaciones por cada operación que ejecute el programa. La recuperación transitiva de estructuras de datos completas puede ser pospuesta, e incluso se puede hacer poco a poco, manteniendo una lista de objetos por liberar –aquellos

[1]Algunos autores utilizan el término "recolección de basura" en un sentido más exclusivo que no incluye a los sistemas de conteo de referencias o copiado; aquí lo utilizaremos en lugar del término más rebuscado y de poco uso "recuperación automática de almacenamiento"

cuya cuenta de referencias es cero– pero que aún no han sido procesados.

Esta recolección incremental puede satisfacer los requerimientos de un sistema de tiempo real, garantizando que las operaciones del recolector de basura nunca detendrán la ejecución del programa más allá de una pequeña fracción de tiempo. Esto nos permitiría soportar aplicaciones de tiempo real en las cuales garantizar el tiempo de respuesta es un parámetro crítico; la recolección incremental asegura que el programa podrá realizar una cantidad significativa de trabajo, notoriamente reducida probablemente por unidad de tiempo (un posible objetivo puede ser el no gastar más de un milisegundo de cada periodo de dos-milisegundos en operaciones propias de la recolección, dejando así un milisegundo para "trabajo útil" que pueda satisfacer las necesidades de tiempo real del programa.)

Hay dos problemas con los recolectores que utilizan conteo de referencias: son difíciles de hacer eficientes y no siempre son efectivos.

El problema con los ciclos. El problema es que con conteo de referencias no podemos recuperar estructuras circulares. Si las referencias en un conjunto de objetos crea un ciclo dirigido, los contadores de referencias de los objetos nunca serán reducidos a cero, *aún cuando no exista ninguna trayectoria a ellos desde el conjunto raíz.*

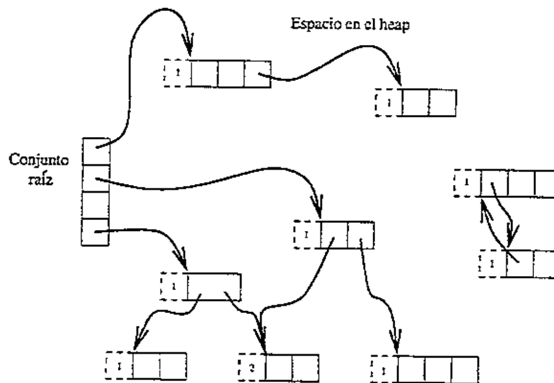


Figura 2.2: Conteo de referencias, con un ciclo irrecuperable

En la Figura 2.2 se ilustra este problema. Considere el par de objetos aislados en la parte derecha de la figura. Cada uno tiene una referencia al otro, y por lo tanto sus contadores de referencias valen uno. Y no hay ninguna ruta del conjunto raíz a ellos; sin embargo, el recolector no puede recuperar sus espacios.

Conceptualmente hablando, el problema es que el conteo de referencias sólo determina una *aproximación conservadora* de estar vivo. Si un objeto no está siendo referido por cualquier otro objeto, claramente es basura, lo contrario no siempre es cierto.

Parecería que las estructuras circulares no son muy usuales, pero no es así. Mientras que muchas estructuras de datos son acíclicas, es frecuente –en programas normales– crear algunos ciclos, y algunos programas crean muchos. Por ejemplo, los nodos en

árboles pueden tener ligas que apuntan al padre, para facilitar algunas operaciones. Otros ciclos más complejos se forman al combinar estructuras complejas que combinan las ventajas de varias estructuras más simples, etc.

Por este motivo, los sistemas que utilizan recolectores de basura con conteo de referencias, usualmente incluyen alguna otra forma de recolección y periódicamente el recolector de basura utiliza el segundo esquema de recolección, para evitar que se acumule mucha basura cíclica.

Muchos programadores que utilizan sistemas con conteo de referencias (tales como *Interlisp* y versiones antiguas de *Smalltalk*) han modificado su estilo de programación para evitar crear estructuras *cíclicas* que se conviertan en basura, o bien para romper tales ciclos antes de que sean basura y puedan ser detectados por el recolector.

El problema de la eficiencia. Otro problema con el conteo de referencias es que el costo es proporcional a la cantidad de trabajo realizado por el programa en ejecución, con una gran constante de proporcionalidad.

Un costo es que cada vez que una referencia es creada o destruida, su cuenta de referencias debe ser ajustada. Si el valor de una variable cambia de una referencia a otra, la cuenta de dos *objetos* debe ajustarse –la de uno de ellos debe ser incrementada y la del otro reducida y además, en éste último revisar si la cuenta ha llegado a cero o no.

Las variables de stack de corta vida introducen una sobrecarga muy pesada en un sistema simple de conteo de referencias. Cuando se pasa un argumento, por ejemplo, aparece un nuevo apuntador en el stack, que usualmente desaparece muy poco tiempo después ya que la mayoría de las activaciones de procedimientos (cerca de las hojas de la gráfica de llamadas) regresan muy poco tiempo después de haber sido llamadas. En estos casos, se incrementan contadores de referencias, y poco después son decrementados a su valor original. Es deseable optimizar estos aumentos y decrementos que se cancelan mutuamente, eliminándolos.

Conteo de referencias pospuesto

Gran parte del costo que representan las variables de corta vida, puede ser eliminado manejando las variables locales de manera especial. En lugar de ajustar los contadores de referencias y recuperar el espacio ocupado por objetos cuya cuenta sea cero, ignoramos las referencias provenientes de variables locales. Usualmente, los contadores de referencias sólo serán ajustados para reflejar ligas de un objeto en el heap a otro. Esto implica que los contadores de referencias pueden no ser muy precisos, ya que se pueden crear o destruir objetos en el stack sin que sean registrados por el sistema de conteo; lo cual, a su vez, significa que algunos objetos cuya cuenta se reduzca a cero **no** pueden ser recuperados. La recolección de basura sólo puede hacerse cuando se toman en cuenta las referencias provenientes del stack

De vez en cuando, los contadores de referencias son actualizados revisando el stack y contabilizando las referencias (si las hay) a objetos en el heap. Sólo entonces, aquellos objetos cuyos contadores de referencias sean cero pueden ser recuperados de manera segura. El

intervalo entre estas fases es generalmente pequeño para asegurar que la basura se recolecte continua y rápidamente; sin embargo, es lo suficientemente grande para que el costo de actualizar los contadores periódicamente (buscando referencias provenientes del stack) no sea alto.

El posponer el conteo de referencias evita que se actualicen los contadores para la gran mayoría de variables de corta vida del stack, y también reduce considerablemente la sobrecarga que impone esta técnica. Cuando se crean o destruyen ligas de un objeto a otro en el heap, los contadores de referencias respectivos deben ser ajustados. El costo sigue siendo –en casi todos los sistemas– proporcional a la cantidad de trabajo realizada por el programa en ejecución, pero con una constante de proporcionalidad más pequeña.

Variaciones al conteo de referencias

Otra optimización que podemos hacer en el conteo de referencias es usar un campo de tamaño muy pequeño para el contador, un solo bit, por ejemplo. Esto evita el tener que mantener un campo muy grande por cada objeto. Dado que posponer el conteo de referencias nos evita la necesidad de representar la cuenta de referencias provenientes del stack, un solo bit es suficiente para la mayoría de los objetos.

Existe otro costo más en la técnica de conteo de referencias que es más difícil de evitar. Cuando la cuenta de algunos objetos se vuelve cero y son reclamados, debe hacerse cierto trabajo administrativo para ponerlos a disposición del programa en ejecución. Típicamente esto involucra ligar los objetos recién liberados a una o más *listas libres* de objetos reusables, de las cuales se satisfacen las solicitudes de memoria del programa (más adelante en esta misma parte del trabajo se discutirán otras estrategias, en el contexto de recolección de marcado y barrido, en la sección 2.2.2). Los campos de referencias de los objetos también deben examinarse para que los objetos a los cuales apuntan puedan ser liberados.

Es difícil que estas operaciones de recuperación nos tomen menos de un par de docenas de instrucciones por objeto, y por lo tanto el costo es proporcional al número de objetos colocados en memoria por el programa en ejecución.

Los costos asociados con la técnica de conteo de referencias para recuperar memoria, aunados a su imposibilidad para reclamar estructuras circulares han provocado que esta técnica se haya vuelto poco atractiva para los implementadores en los últimos años. Como se verá más adelante, existen técnicas más eficientes y confiables. Aún así, el conteo de referencias tiene algunos puntos a su favor. La facultad de poder recuperar memoria inmediatamente contribuye a un mejor manejo, en promedio, de la memoria y favorece ampliamente la propiedad de *localidad de referencia*; los sistemas que utilizan conteo de referencias tienen un desempeño con muy poca degradación, aún cuando todo el heap este ocupado por objetos vivos, mientras otros recolectores se basan en intercambiar un mayor espacio por eficiencia.

La inhabilidad para reclamar estructuras cíclicas no es un problema en algunos lenguajes de programación que no permiten construcciones de estructuras cíclicas (v.gr., lenguajes funcionales puros). El conteo de las referencias puede ser valioso en algunos sistemas. Por ejemplo, en algunas implementaciones de lenguajes funcionales, estos contadores pueden ser utilizados para realizar optimizaciones, permitiendo hacer modificaciones

structivas en objetos que tengan una sola referencia[2]. La recolección distribuida de basura mbién puede beneficiarse del conteo de referencias, en lugar de usar rastreos globales.

La mayoría de las implementaciones de alto rendimiento de los lenguajes de progra- ación de propósito general modernos no utilizan conteo de referencias; pero no sucede lo ismo en otro tipo de aplicaciones, donde son comunes las estructuras acíclicas. Una buena arte de los sistemas de archivos utilizan conteo de referencias para manejar archivos o blo- ues de discos. Además, debido a su simplicidad, el conteo de referencias se usa en paquetes e software tales como lenguajes de programación sencillos –generalmente interpretados– y aquetes de diseño gráfico.

.2.2 Recolección de marcado y barrido

Los recolectores de basura de marcado y barrido (*mark-sweep*) se llaman así por as dos fases que implementan el algoritmo abstracto de recolección de basura descrito an- eriormente:

1. *Distinguir a los objetos vivos de la basura.* Esto se hace durante el recorrido – comenzando en la conjunto raíz y recorriendo la gráfica de relaciones de referencias– usualmente, utilizando una búsqueda a profundidad o búsqueda en amplitud. Los ob- jetos que se alcanzan se *marcan* de alguna forma, ya sea alterando algunos bits dentro del objeto o tal vez llevando una bitácora de ellos en un mapa de bits o algún otro tipo de tabla.
2. *Recuperar la basura.* Una vez que los objetos vivos son distinguibles de los objetos ba- sura, la memoria se *barre*, esto es, se examina exhaustivamente, para encontrar a todos los objetos sin marcar (basura) y recuperar el espacio que éstos ocupan. Tradicional- mente, al igual que con el conteo de referencias, estos objetos recuperados se ligan en una o más listas de objetos libres para que puedan ser accedidos por las rutinas de asignación de espacio.

Hay tres grandes problemas con los recolectores tradicionales de marcado y ba- rrido. Primero, es difícil manejar objetos de tamaños variados sin fragmentar la memoria disponible. El espacio recuperado de la basura está intercalado junto con los objetos vivos, por lo que la asignación posterior de espacio a objetos grandes puede ser problemática; es decir, el espacio libre que representa toda la basura recuperada, puede en su *totalidad* ser suficiente para almacenar a un objeto grande; sin embargo, como los espacios libres no son –necesariamente– contiguos, puede resultar imposible asignarle memoria al nuevo objeto. Esto puede ser mitigado en cierta medida manteniendo listas de objetos libres de tamaños variados, y fusionando espacios libres contiguos en uno sólo. Aún así algunas dificultades no pueden ser resueltas: el sistema debe escoger si asignar más memoria conforme se vaya necesitando para crear objetos pequeños o dividir pedazos grandes contiguos de memoria con el riesgo de fragmentarlos permanentemente. Este problema de fragmentación no es

[2]En la gran mayoría de los lenguajes funcionales, hay una marcada diferencia entre operaciones destruc- tivas y no destructivas (o de copia) Las primeras reutilizan la memoria ocupada por los objetos sobre los cuales actúa la operación; mientras que las operaciones no destructivas, hacen una copia de los objetos y trabajan con las copias.

único al marcado y barrido –también ocurre en conteo de referencias y en la mayoría de los esquemas que manejan explícitamente el heap.

El segundo problema con la recolección de marcado y barrido es que el costo de la recolección es proporcional al tamaño del heap, incluyendo tanto a los objetos vivos como a la basura. Todos los objetos vivos deben ser marcados y todos los objetos basura deben ser recolectados, lo que impone una limitación en cualquier posible mejora en la eficiencia.

El tercer problema involucra a la localidad de referencia. Ya que los objetos nunca son movidos, los objetos vivos se mantienen en su lugar original aún después de la recolección, intercalados con espacio libre. Entonces los nuevos objetos son asignados a esos espacios; el resultado es que objetos de muy diferentes edades terminan intercalados en la memoria. Esto tiene implicaciones negativas para la localidad de referencia, y los recolectores de marcado y barrido simples (no generacionales) son considerados *inadecuados* para la mayoría de las aplicaciones que manejan memoria virtual —es posible que el "conjunto de trabajo" de objetos activos esté esparcido en varias páginas virtuales de memoria, por lo que esas páginas son intercambiadas frecuentemente entre la memoria y el área de swap. La fragmentación y la localidad de referencia son —en el caso general— inevitables; sin embargo, representan un peligro potencial para algunos programas.

Cabe notar que estos problemas no son insuperables con técnicas lo suficientemente ingeniosas. Por ejemplo, si se utiliza un mapa de bits para bits de marcaje, se pueden marcar 32 bits de un sólo golpe utilizando una operación ALU con enteros de 32 bits y un brinco condicional. Si objetos vivos tienden a sobrevivir en secciones de memoria pequeñas, como aparentemente lo hacen, esto puede disminuir considerablemente la constante de proporcionalidad de la fase de barrido; la dependencia lineal en el tamaño de la memoria puede no ser tan problemática como parece a primera vista. Los objetos vivos acumulados en pequeñas secciones de memoria también mitigan los problema de localidad y reutilización de espacio entre los objetos vivos; si los objetos tienden a sobrevivir o morir en grupo en la memoria, la mezcla de objetos usados por distintas fases del programa no tiene gran relevancia —véase [Hayes, 1991].

2.2.3 Recolección de marcado y compactación

Los recolectores de marcado y compactación solucionan los problemas de fragmentación y asignación de memoria de los recolectores de marcado y barrido. Al igual que los recolectores de marcado y barrido, una fase de marcado recorre y marca a los objetos alcanzables. A continuación los objetos vivos se *compactan*, moviendo todos los objetos vivos, hasta que éstos se encuentran en posiciones contiguas de memoria. Esto deja el resto de la memoria como un solo espacio contiguo libre. Esto se hace generalmente haciendo un recorrido lineal de la memoria, encontrando los objetos vivos y *recorriéndolos* hasta que sean adyacentes al objeto vivo previo. Eventualmente, todos los objetos vivos serán recorridos hasta ser adyacentes a un vecino vivo. Esto deja un área contigua de memoria ocupada en el heap y todos los "hoyos" se mueven implícitamente al otro lado del heap (a lo que ahora es un espacio contiguo de memoria libre en el otro extremo del heap.)

Esta compactación por corrimientos tiene varias propiedades interesantes. El área libre contigua elimina los problemas de fragmentación, de manera que asignar memoria a objetos de distintos tamaños es muy simple. La asignación puede llevarse a cabo incremen-

ando un apuntador en un área contigua de memoria, análogo a la forma en que podemos signar espacio en el stack a objetos de distintos tamaños. Además, la basura se *saca*, sin nodificar el orden original de los objetos en la memoria. Esto aminora el problema de ocalidad, porque el orden de asignación de memoria es más similar al orden de accesos ubsecuentes que un nuevo orden arbitrario como el que impone un recolector de basura de opia.

Si bien la localidad que resulta al compactar es una gran ventaja, el proceso de ecolección mismo comparte la desafortunada propiedad de necesitar varias pasadas sobre el onjunto de objetos al igual que los recolectores de marcado y barrido. Después de la fase le marcaje, los compactadores hacen dos o tres pasadas más sobre los objetos vivos. Una pasada calcula la nueva localización a la que serán movidos los objetos. Por lo tanto, estos algoritmos pueden ser significativamente más lentos que los de marcado y barrido si un gran porcentaje de datos sobreviven y tienen que ser compactados.

Un enfoque alternativo es usar un algoritmo con dos apuntadores[3], que buscan hacia adentro partiendo de ambos extremos de un área en el heap para encontrar mayores oportunidades de compactación. Un apuntador recorre hacia abajo el heap partiendo de la parte más alta, buscando objetos vivos y el otro busca hacia arriba partiendo de la parte más baja, buscando hoyos en los cuales colocarlos, (muchas variantes de este algoritmo son posibles, para lidiar con múltiples áreas que contengan objetos de distintos tamaños y evitar así intercalar objetos provenientes de áreas muy distintas).

2.2.4 Recolección de basura de copiado

Al igual que la recolección de marcado y compactación (pero a diferencia de la recolección de marcado y barrido) la recolección de copiado[4]no "recolecta" la basura. En su lugar, mueve todos los objetos *vivos* a una cierta área, y el resto del heap es tomado como memoria disponible, puesto que sólo contiene basura. Por lo tanto, en estos sistemas el mecanismo de "recolección de basura" es implícito, incluso mucho autores evitan aplicar el término para referirse a ellos.

Los recolectores de copiado, al igual que la recolección de marcado y compactación, mueven los objetos que son alcanzados mientras se recorren las referencias en el heap a un área contigua de éste último Sin embargo, los recolectores de copiado no tienen una fase de marcado, integran el recorrido de los datos con la copia de los mismo, por lo que la mayoría de los datos sólo son revisados una sola vez. El trabajo necesario es, por lo tanto, proporcional a la cantidad de datos vivos (ya que todos ellos deben ser copiados).

El recorrido junto con la copia de los objetos se conoce como *localización* (*scavenging*), porque consiste en escoger a los objetos que aún se usarán de entre la basura y los lleva a una nueva posición.

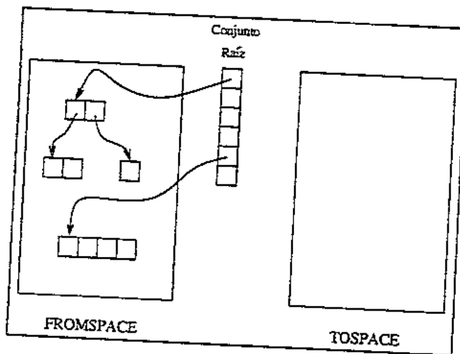
[3]Este algoritmo fue propuesto por *Daniel J Edwards* y es descrito en la página 421 de [Knuth, 1969].

[4]Nota histórica: el primer recolector de basura de copiado fue el de *Minsky* usado en Lisp 1 5 [Minsky, 1963]. En lugar de copiar los datos de una parte de la memoria a otra, utilizaba un sólo espacio en el heap Los datos vivos se copian en un disco y luego se leían nuevamente en un área contigua de memoria en el heap En máquinas modernas, esto sería intolerablemente lento, porque las operaciones con archivos –leer y escribir objetos vivos– es varios órdenes de magnitud más lento que hacer operaciones en la memoria

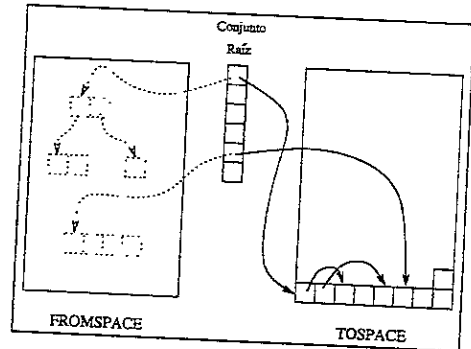
Un recolector de copiado sencillo: "Detente y copia" usando semi-espacios

En este tipo de recolector, el espacio dedicado al heap se subdivide en dos semi-espacios contiguos. Durante la ejecución normal del programa, sólo uno de estos semi-espacios está en uso, como se muestra en la Figura 2.3.1. La memoria es asignada linealmente en el semi-espacio *actual* conforme el programa en ejecución la va solicitando. Al igual que con los recolectores de marcado y compactación, el uso de un espacio de memoria libre contiguo hace que la asignación sea simple y rápida; no hay fragmentación, aún cuando se asigne memoria a objetos de diversos tamaños.

Cuando el programa en ejecución solicita una cantidad de memoria que no alcanza con la memoria libre en el semi-espacio actual, el programa se detiene y se llama al recolector de basura de copiado para recuperar el espacio (de aquí surge el termino "detente y copia").



2.3.1: Antes de la recolección



2.3.2: Después de la recolección

Figura 2.3: Un recolector con semi-espacios simple

Todos los datos vivos se copian del semi-espacio actual (*fromspace*) al semi-espacio vacío (*tospace*). Una vez que el copiado se ha completado, el nuevo semi-espacio se convierte en el semi-espacio actual, y se continúa la ejecución del programa. Por lo tanto, los roles de ambos sub-espacios se intercambian cada vez que se invoca al recolector de basura (véase la figura 2.3.2).

El esquema de copiado más simple que conocemos es el algoritmo de Cheney [Cheney, 1970]. En este algoritmo, los objetos alcanzables inmediatamente forman la cola inicial de objetos para aplicar un recorrido en amplitud (BFS). Un apuntador de "búsqueda" avanza a partir del primer objeto, localidad por localidad. Cada vez que se encuentra con una referencia hacia el semi-espacio *fromspace*, el objeto referido es movido al final de la cola, y la referencia al objeto es actualizada para reflejar este hecho. Se avanza el apuntador libre y la búsqueda continúa. Esto se conoce, en la búsqueda en amplitud, como la *expansión del nodo*, ya que todos los descendientes del nodo son alcanzados –y copiados–, (véase la Figura 2.4). Todas las estructuras de datos en el sub-espacio *fromspace* se muestran en la parte superior de la figura, a continuación se muestran los primeros estados por los que el sub-

pacio *tospace* pasa durante el proceso de la recolección –el sub-espacio *tospace* se muestra
omo una sección de memoria linealmente ordenada para enfatizar la búsqueda lineal y el
opiado.

El trabajo de recolección no termina al llegar al final del primer objeto, sino que
l proceso de búsqueda continúa ahora sobre objetos subsecuentes, encontrando a sus de-
cendientes y copiándolos. Un recorrido continuo desde el inicio de la cola tiene el efecto
e eliminar nodos consecutivos y encuentra a todos sus descendientes. Los descendientes se
opian al final de la cola. Eventualmente, la búsqueda llega al final de la cola, lo que significa
ue todos los objetos han sido alcanzados por la búsqueda (y copiados) y también han sido
ecorridos en busca de sus descendientes. Por lo tanto, ya no hay objetos alcanzables que
opiar y el proceso de localización termina.

En la práctica se requiere un proceso un tanto más complicado, para evitar que
objetos con múltiples referencias no sean copiados al espacio *tospace* varias veces. Cuando se
ransporta un objeto al *tospace*, una *referencia de actualización* se instala en la vieja versión
del objeto. Esta referencia de actualización indica que el objeto viejo es obsoleto y tiene
además una referencia a la nueva localización del objeto (a la copia del objeto en el sub-
espacio *tospace*). Cuando el proceso de localización encuentra una referencia hacia un objeto
en el *fromspace*, se checa si el objeto referido tiene una referencia de actualización. Si la
tiene, dicho objeto ya había sido movido al *tospace*, por lo que sólo se actualiza la referencia
por la cual llegamos a él para apuntar a la nueva posición del objeto. Esto asegura que cada
objeto vivo es "movido" sólo una vez y que todas las referencias a él son actualizadas para
referirse a la copia nueva.

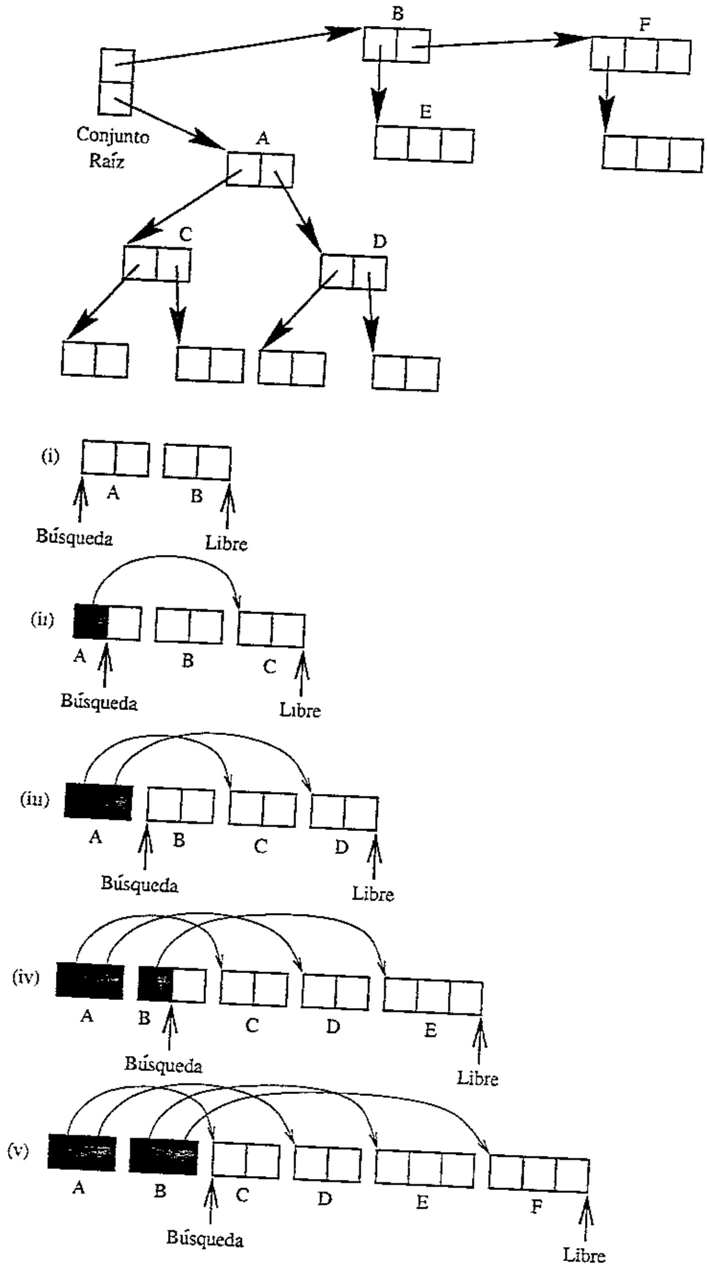


Figura 2.4: Algoritmo de Cheney. En la parte superior se muestran las estructuras de datos en el *fromspace* y a continuación los primeros pasos durante el proceso de localización, utilizando búsqueda en amplitud.

Eficiencia de la recolección de copiado

Un recolector de basura de copiado se puede hacer tan eficiente como se desee si se cuenta con una cantidad de memoria adecuada. El trabajo realizado durante cada recolección es proporcional a la cantidad de datos vivos en ese momento. Si se supone que aproximadamente está viva la misma cantidad de información en cualquier momento durante la ejecución del programa, al decrementar la frecuencia de la recolección de basura se decrementa el esfuerzo total invertido en la misma.

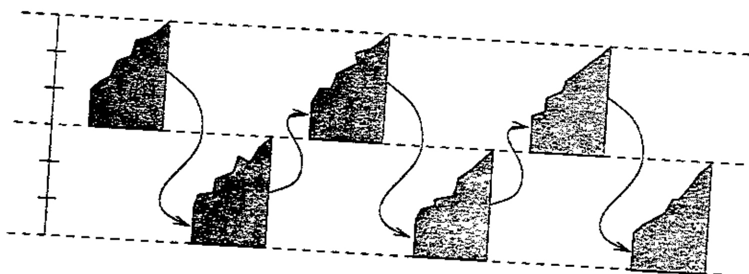
Una forma sencilla de decrementar la frecuencia de la recolección de basura es aumentar la cantidad de memoria en el heap. Si cada semi-espacio es más grande, el programa correrá más tiempo antes de llenarlo. Otra ventaja de esta situación es que decrementando la frecuencia de la recolección de basura, aumentamos la vida promedio de los objetos. Los objetos que se convierten en basura antes de que empiece la recolección no son copiados, por lo que la probabilidad de que no tengamos que copiar un objeto crece.

Supongamos, por ejemplo, que durante la ejecución de un programa se asignan 20 mega-bytes de memoria, pero sólo un mega-byte está vivo en cualquier momento. Si tenemos dos semi-espacios de tres mega-bytes cada uno, la basura será recolectada diez veces (dado que el semi-espacio actual está lleno en una tercera parte después de la recolección, lo que deja sólo dos mega-bytes para asignar antes de la siguiente recolección). Esto significa que el sistema copiará la mitad de toda la información que asigne, tal y como se muestra en la Figura 2.5.1. (Las flechas representan el copiado de los objetos vivos entre semi-espacios durante la recolección.)

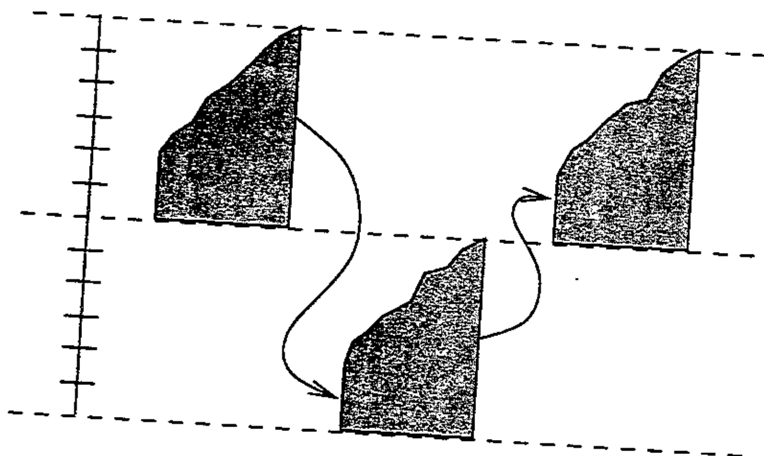
Por otro lado, si el tamaño de los semi-espacios se duplica, después de hacer la recolección quedarán 5 mega-bytes de espacio libre en el semi-espacio actual. Esto reducirá la necesidad de recolección a un tercio de la anterior, o bien, 3 o 4 veces durante la ejecución del programa. Esto reduce de manera directa el costo de la recolección a menos de la mitad, tal como se muestra en la Figura 2.5.2 (por el momento, ignoramos el costo generado por la paginación cuando se usa memoria virtual, suponiendo que el área más grande de heap puede ser mantenida en memoria principal y no hay necesidad de pasarla al área de swap, en el disco. Más adelante, en la sección 2.2.7, explicaremos cómo el uso de un área grande de heap puede ser poco práctica si la cantidad de memoria RAM es insuficiente, debido al costo de la paginación).

2.2.5 Recolección implícita sin copiado

Recientemente se han propuesto nuevas formas de recolección sin utilizar copiado, pero con algunas de las ventajas en eficiencia que los esquemas de copiado ofrecen. La afirmación central en estos recolectores es que en los recolectores de copiado, los "espacios" del recolector son, en realidad, una implementación particular de conjuntos. El proceso de recorrido elimina objetos del conjunto sujeto a la recolección; cuando se completa el recorrido, cualquier cosa que permanezca aún en el conjunto se dice que es basura, por lo que el conjunto puede ser recuperado en su totalidad. Otra implementación de conjuntos podría funcionar también, sólo tendríamos que asegurarnos de que tuviera características de desempeño similares. En particular, dada una referencia a un objeto, debe ser fácil determinar a qué conjunto pertenece; además, debe ser fácil cambiar el rol de los conjuntos.



2.5.1: Con 3MB por semi-espacio



2.5.2: Con 6MB por semi-espacio

Figura 2.5: Uso de memoria en un recolector de basura con semi-espacios

de la misma forma que los semi-espacios *fromspace* y *tospace* eran intercambiados en un recolector de copiado. En un recolector de copiado, el conjunto es un área de memoria, pero en un recolector sin copiado, puede ser cualquier tipo de conjunto de fragmentos de memoria, en los cuales se hayan almacenado objetos vivos.

El nuevo sistema sin copiado añade dos campos de referencia y un campo "color" a cada objeto. Estos campos son invisibles al programador de aplicaciones, y sirven para ligar cada *segmento* de memoria en una lista doblemente ligada que será nuestro conjunto. El campo de color indica a qué conjunto pertenece el objeto.

La forma de operar de este recolector es simple, e isomorfa a la operación del recolector de copiado[5]. Los segmentos de memoria libre se ligan para formar una lista doblemente ligada, mientras que los segmentos de memoria que contienen objetos asignados

por el programa en ejecución se ligan en otra lista.

Cuando la lista de segmentos libres se agota, el recolector recorre los objetos vivos y los "mueve" del conjunto en el que fueron asignados (que llamamos *fromset*) a otro conjunto (el *toset*). Esto es implementado, desligando al objeto de la lista doblemente ligada, *fromset*, invirtiendo su campo de color, y ligándolo en la lista doblemente ligada *toset*.

Al igual que en la recolección de copiado, la recuperación de espacio es implícita. Cuando todos los objetos alcanzables han sido recorridos y movidos de *fromset* a *toset*, el conjunto *fromset* sólo contiene basura. Es, por lo tanto, un conjunto de espacio libre, que puede ser utilizado inmediatamente para satisfacer lor requerimientos de memoria del programa que se ejecuta. El costo de la recolección es proporcional al número de objetos vivos, mientras que la basura es recuperada en un tiempo constante muy pequeño.

Este esquema puede ser optimizado de varias formas análogas a las que usamos en los recolectores de copiado –la asignación puede ser más rápida porque las listas de segmentos libres y utilizados pueden ser contiguas y separadas únicamente por una referencia. En lugar de desligar objetos de una lista y ligarlos en otra, el asignador de memoria puede simplemente avanzar un apuntador que sirve como "marca" para separar el espacio libre del usado. De manera similar, el recorrido en amplitud que hacemos en el algoritmo de Cheney (Sección 2.2.4), puede ser implementado con sólo dos apuntadores, y las listas de espacios libres y ocupados pueden ser contiguas, de forma tal que cuando avanzamos el apuntador de "búsqueda" sólo requerimos avanzar el apuntador que separa las dos listas.

Este esquema tiene tanto ventajas como desventajas comparado con los recolectores de copiado. En el lado negativo, las constantes por objeto son un poco más grandes y existe la posibilidad de fragmentación. En el lado positivo, el costo de recorrer objetos muy grandes no es tan alto como en los recolectores de copiado. Al igual que los recolectores de marcado y barrido, no tenemos necesidad de copiar todo el objeto; si no puede almacenar referencias, entonces tampoco necesitamos revisarlo. Lo que es más importante acerca de este esquema, para muchas aplicaciones, es que no requerimos que referencias entre objetos –al nivel del lenguaje– sean modificadas, lo cual impone menos restricciones sobre los compiladores. Como explicaremos más adelante, esto es particularmente importante para recolectores paralelos y recolectores incrementales en tiempo real.

Los costos de espacio de esta técnica son difícilmente equiparables a los de los recolectores de copiado. Requerimos dos apuntadores por objeto, pero los objetos vivos que son recorridos *no* requieren espacio en ambos semi-espacios (*fromspace* y *tospace*) simultáneamente. En la mayoría de los casos, esto hace que el costo en espacio sea más pequeño comparado con el de los recolectores de copiado, pero los costos de la fragmentación, cuando ésta ocurre, (debido a la imposibilidad de este esquema para compactar datos) puede sobrepasar el ahorro en espacio.

2.2.6 Técnicas básicas de recorrido, ¿cuál escoger?

El tratamiento que se da a los algoritmos de recolección de basura en libros de texto, generalmente, supone una complejidad asintótica, pero todos los algoritmos básicos

[5] Por este motivo, *Wang*, quien propuso este esquema, se refiere a este tipo de recolección como "de falso copiado". *Tomas Wang* y *Henry G. Baker* propusieron este esquema de recolección aproximadamente al mismo tiempo –finales de la década de los ochenta–, de manera independiente

tienen costos similares, especialmente cuando vemos a la recolección de basura como parte del esquema de manejo del área de almacenamiento libre. Asignación y recolección de memoria son dos lados de misma moneda (reutilización básica de memoria), y cualquier algoritmo implica un gasto extra en la asignación de memoria. Un criterio muy usado para recolección de basura de "alto rendimiento" es comparar el costo de la recolección de basura, en promedio, con el costo de la asignación de memoria a los objetos.

Debido a lo anterior, cualquier recolector de memoria de recorrido tiene tres componentes que se suman para calcular su costo:

1. El trabajo inicial requerido cada vez que se inicia la recolección, como la revisión del conjunto raíz,

2. El trabajo realizado durante la asignación (proporcional a la cantidad de memoria asignada, o el número de objetos a los cuales se asigna memoria), y

3. El trabajo realizado durante la localización de basura (por ejemplo, el recorrido).

El trabajo inicial está –usualmente– determinado para un programa particular, por el tamaño del conjunto raíz. El trabajo realizado durante la asignación es proporcional al número de objetos asignados a la memoria, más un costo de inicialización proporcional al tamaño de los objetos. El costo de la localización de basura es proporcional a la cantidad de objetos vivos que debed ser recorridos.

Los últimos dos costos son muy similares, ya que la cantidad de objetos vivos recorridos es usualmente un porcentaje significativo de la cantidad de memoria asignada por el programa. Por lo tanto, aquellos algoritmos cuyo costo es proporcional a la cantidad de memoria asignada (v.gr. marcado y barrido) pueden competir con los costos de aquéllos cuyo costo es proporcional a la cantidad de datos vivos recorridos (v.gr. copiado).

A manera de ejemplo, supongamos que 10% de los objetos asignados sobreviven a la recolección y que el 90% nunca necesitan ser recorridos. Al decidir cuál algoritmo es más eficiente, la complejidad asintótica es menos importante que las constantes asociadas. Si el costo de *barrer* un objeto es 10 veces menor que el costo de *copiarlo*, entonces un recolector de marcado y barrido nos cuesta aproximadamente lo mismo que uno de copiado. Si el costo del barrido es cargado[6]a la rutina de asignación de memoria, y si además es relativamente pequeño comparado con el costo de inicializar los objetos, entonces es claro, que el costo de la etapa de barrido no es extremadamente alto. En implementaciones de vanguardia, la diferencia entre los recolectores de marcado y barrido y recolectores de copiado no es muy grande; sin embargo, los segundos "aparentan" ser mucho más eficientes.

En sistemas donde la cantidad de memoria no es mucho mayor que la cantidad de información viva, los recolectores que no mueven nada tienen una considerable ventaja sobre los recolectores de copiado, ya que no necesitan espacio para almacenar dos versiones de cada objeto vivo (las versiones "from" y "to"). Cuando el espacio es muy "justo", los recolectores con conteo de referencias son particularmente atractivos, ya que su rendimiento es independiente de la relación entre los objetos vivos y la cantidad de memoria.

[6]Este tipo de "movimiento" de costos entre distintas etapas de un programa, es conocido como *amortización de costos*. Es una técnica muy utilizada en análisis de algoritmos para compensar los gastos excesivos en los que alguna función muy utilizada en nuestro algoritmo, pueda incurrir.

Más aún, en sistemas de alto rendimiento es muy usual encontrar técnicas híbridas para ajustarse a los cambios provocados por las distintas categorías de objetos. Algunos recolectores de copiado, utilizan un área separada para almacenar *objetos grandes* y evitar así copiarlos de un semi-espacio a otro. Los objetos grandes son "pasados por alto" y usualmente se manejan por separado, sin moverlos de su lugar, con la ayuda de otras técnicas, como una variante de marcado y barrido junto con técnicas de listas de objetos libres. Otros esquemas híbridos pueden utilizar técnicas sin copiado la mayor parte del tiempo, y ocasionalmente compactar parte de los datos utilizando técnicas de copiado para evitar la fragmentación permanente de la memoria.

Uno de los puntos más importantes a favor de las técnicas en-sitio (sin copiado), es la habilidad de hacerlos *conservadores* con respecto a los datos que pueden o no ser referencias. Esto nos permite utilizarlos para lenguajes tales como C, o para compiladores con optimización, en los cuales es difícil o imposible identificar sin ambigüedad todas las referencias en tiempo de ejecución. Un recolector sin copiado puede ser conservador, ya que cualquier cosa que "luzca" como un apuntador puede ser dejado en donde está, y el objeto (probablemente) referenciado no tiene porqué ser cambiado. En contraste, un recolector de copiado debe saber si el valor corresponde a una referencia o no –y si debe mover el objeto y actualizar la referencia a él o no. Las técnicas conservadoras para encontrar referencias serán discutidas con mayor detalle en la sección 2.2.8.

2.2.7 Problemas con los recolectores de recorrido sencillo

Es ampliamente conocido el hecho de que la complejidad asintótica de los recolectores de copiado es excelente –el costo de la recolección se acerca a cero conforme va creciendo la memoria. Algunos otros recolectores pueden ser igual de eficientes si las constantes asociadas con la recuperación y reasignación de memoria son lo suficientemente pequeñas. En ese caso, la recolección de basura representa el costo más importante

Desafortunadamente, es muy difícil en la práctica alcanzar grandes niveles de eficiencia con un recolector de basura sencillo, ya que obtener grandes cantidades de memoria es costoso. Si se utiliza memoria virtual, una pobre localidad durante el ciclo de asignación y recuperación de memoria causará una paginación excesiva. En procesadores de alta velocidad, el simple hecho de bajar al área de swap la página que contiene datos recientemente almacenados, puede resultar muy costoso, y la paginación provocada por un recolector de copiado puede resultar excesiva, ya que todos los datos vivos deben "tocarse" en el proceso.

Por lo tanto, no es conveniente hacer el área de heap mayor a la cantidad de memoria disponible. Aún cuando la memoria principal es cada día más barata, la localidad dentro de la memoria cache es cada vez más importante, por lo que el problema no desaparece, simplemente se mueve a otro nivel en la jerarquía de memoria.

En general, es imposible alcanzar la eficiencia potencial de los esquemas de recolección sencillos; incrementar el tamaño de la memoria para posponer o evitar las recolecciones sólo puede realizarse mientras el costo de la paginación sea más pequeño que la ganancia obtenida al posponer la recolección

Es importante aclarar que este problema no es único de los recolectores de copiado *Todas* las estrategias eficientes de recolección involucran intercambios de espacio-tiempo similares –la recolección de basura es pospuesta de forma tal que el trabajo de detección se hace

menos frecuentemente, lo que significa que el espacio no se puede recuperar rápidamente. En promedio, esto incrementa la cantidad de memoria desperdiciada en basura no recuperada.

2.2.8 Recolectores de basura conservadores

Un recolector de basura ideal es capaz de recuperar el espacio de cualquier objeto justo después del último uso de dicho objeto. Por supuesto que tal recolector *no es implementable* en la práctica, ya que no se puede determinar, en general, cuando ocurre el último uso de un objeto. Los recolectores de basura reales proveen solamente una aproximación razonable de este comportamiento, utilizando aproximaciones conservadoras de esta omnisciencia. El arte de diseñar recolectores de basura eficientes consiste en introducir acciones conservadoras que reducen significativamente el trabajo que hacemos para detectar la basura. La noción *conservador* en este contexto es muy general y no debe confundirse con las técnicas de identificación de referencias usadas por los recolectores de basura "conservadores". Todos los recolectores de basura son conservadores en una o más formas.

La primera suposición conservadora que hacen la mayoría de los recolectores es que cualquier variable en el stack, global o en los registros está viva, a pesar de que dicha variable puede nunca volver a ser referenciada.

Los recolectores de recorrido introducen una forma *temporal* de conservadurismo, simplemente permitiendo que la basura no sea recolectada en varios ciclos de recolección. Los recolectores con conteo de referencias son conservadores *topológicamente*, ya que no distinguen entre trayectorias distintas que comparten una arista en la gráfica de relaciones de referencias.

En el siguiente capítulo, mostraremos que hay varias formas posibles y grados de conservadurismo.

Capítulo 3

Recolectores de recorrido incremental

Para aplicaciones de *tiempo real*, necesitamos recolectar la basura de manera incremental. La recolección de basura no se puede llevar a cabo como una sola acción atómica mientras el programa está suspendido, por lo que necesitamos intercalar pequeñas unidades de recolección de basura con pequeñas unidades de ejecución del programa. Como dijimos anteriormente, es relativamente fácil hacer incrementales los recolectores de basura con conteo de referencias. Sin embargo, los problemas de eficiencia y efectividad de los recolectores con conteo de referencias son desalentadores, por lo que es deseable hacer incrementales los recolectores de recorrido (de copiado o marcado).

En gran parte de la discusión que presentamos a continuación, la diferencia entre los recolectores de copiado y los de marcado y barrido no es de particular importancia. El recorrido incremental para la recolección de basura es más interesante que la recuperación de la basura detectada.

La dificultad con los recolectores de recorrido incremental es que mientras el recolector está recorriendo la gráfica de estructuras de datos alcanzables, la gráfica puede cambiar –el programa en ejecución puede *mutar* la gráfica mientras el recolector no *está viendo*. Por esta razón, en la discusión de recolectores incrementales nos referimos al programa en ejecución como *mutante*. Desde el punto de vista del recolector de basura, la aplicación ejecutándose es una corrutina o proceso concurrente con una tendencia desafortunada a modificar las estructuras de datos que el recolector está intentando recorrer. Un esquema incremental debe tener alguna forma de mantener la pista de los cambios a la gráfica de objetos alcanzables; incluso recalcular partes de su recorrido en la presencia de tales cambios.

Una característica importante de las técnicas incrementales es su grado de conservadurismo con respecto a los cambios hecho por el mutante durante la recolección de basura Si el mutante cambia la gráfica de objetos alcanzables, los objetos liberados pueden ser o no recuperados por el recolector. Algunos *objetos flotantes* pueden no ser recuperados porque el recolector ya los había marcado como vivos antes de que el mutante los libere Debe garantizar que esta basura será recolectada al siguiente ciclo, ya que será basura al *inicio* de la siguiente recolección

3.1 Conservadurismo y coherencia

Los recorridos de marcado incremental debe tomar en cuenta los cambios a la gráfica de objetos alcanzables, hechos por el mutante durante el recorrido del recolector. El copiado incremental impone problemas más severos de *coordinación* –el mutante también debe protegerse de cambios realizados por el recolector.

Debe resultar muy ilustrativo ver estas situaciones como una variedad de problemas de *coherencia* -tener varios procesos intentando compartir información, mientras mantienen una especie de vista consistente. Estos conceptos vienen de problemas en sistemas paralelos; sin embargo, no es esencial entender totalmente la terminología de estos sistemas, ya que se irán aclarando conforme vayamos adentrándonos en el tema.

Un recorrido incremental de marcado y barrido es un problema de coherencia con *varios lectores* y *un solo escritor* –el recorrido del recolector debe responder a los cambios, pero el mutante es el único capaz de cambiar la gráfica de objetos. De manera similar, sólo el recorrido puede cambiar los bits de marcaje; cada proceso puede actualizar valores, pero cada campo puede ser escrito por un único proceso. El mutante es el único que escribe en los campos de referencias, mientras que el recolector tiene acceso exclusivo a modificar los campos de marcaje.

Los recolectores de copiado imponen un problema mucho más difícil –un problema con *múltiples lectores y múltiples escritores.* Ambos, el mutante y el recolector pueden modificar campos de referencias y cada uno debe estar protegido de inconsistencias introducidas por el otro.

Los recolectores resuelven eficientemente estos problemas, tomando ventaja de la semántica de la recolección de basura, y utilizando formas de *consistencia relajada* –el proceso en ejecución no siempre *necesita* tener una vista consistente de las estructuras de datos, siempre y cuando esas diferencias entre vistas "no importen" para la correctez del algoritmo.

En particular, la vista del recolector de basura de la gráfica de objetos alcanzables es típicamente *diferente* a la gráfica que ve el mutante. Es simplemente una aproximación *conservadora* de la gráfica real de objetos alcanzables –el recolector de basura puede ver algunos objetos inalcanzables como alcanzables, siempre y cuando no vea objetos alcanzables como inalcanzables, y *erróneamente* recupere el espacio que éstos ocupan. Algunos objetos basura pueden ser pasados como objetos vivos por algún tiempo; éstos son objetos que se convierten en basura justo después de que el recorrido del recolector los ha alcanzado. Tal basura flotante es reclamada, sin duda alguna, en el siguiente ciclo de recolección; ya que se convierten en basura al *inicio* de la recolección, esto elimina la posibilidad de que el recolector (conservadoramente) los vea como objetos vivos. La imposibilidad para recolectar basura flotante inmediatamente representa una desventaja considerable; sin embargo, es esencial para evitar un gasto excesivo de coordinación entre el mutante y el recolector.

El tipo de consistencia relajada usada –y las características de coherencia de este esquema de recolección– están íntimamente relacionados con el concepto de conservadurismo. En general, entre más relajemos la consistencia entre las vistas del mutante y el recolector de la gráfica de objetos alcanzables, nuestra recolección se hace más conservadora, y tendremos que aceptar más basura flotante. En el lado positivo, entre más relajada sea nuestra definición de consistencia, obtenemos mayor flexibilidad en los detalles del algoritmo de recorrido. En la recolección de basura distribuida y paralela, tener consistencia relajada permite un

mayor grado de paralelismo y/o menos sincronización, pero esto es un tema que escapa el alcance de este trabajo.

3.2 Marcado tricolor

Los algoritmos de recolección de basura pueden describirse como el proceso de recorrer la gráfica de objetos alcanzables y colorearlos. La abstracción del *marcado tricolor* es muy útil para entender la recolección incremental de basura. Los objetos sujetos a la recolección son conceptualmente pintados de *blanco*. Al final de la recolección, aquellos que serán conservados deben ser de color *negro*. Cuando ya no hay nodos para pintar de negro, terminamos el recorrido de las estructuras de datos libres.

En un recolector simple de marcado y barrido, este coloreado se implementa asignando directamente bits de marcaje –los objetos cuyos bits se marcan corresponden a objetos pintados de negro. En un recolector de copiado, el proceso de moverlos del *fromspace* al *tospace*, es equivalente a colorearlos –los objetos no alcanzables en el *fromspace* son considerados blancos y los objetos movidos al *tospace* son negros. La abstracción de coloreado es ortogonal a la distinción entre recolectores de marcado y copiado, y es importante para entender las diferencias básicas entre recolectores incrementales.

En los recolectores incrementales, los estados intermedios del recorrido de coloreado son importantes, debido a la acción irrefrenable del mutante –no le podemos permitir al mutante cambiar cosas "a espaldas del recolector", pues el recolector fallaría al tratar de encontrar todos los objetos alcanzables.

Para entender y prevenir tales interacciones entre el mutante y el recolector, es útil introducir un tercer color, *gris*, para indicar que un objeto ha sido alcanzado por el recorrido, pero que *sus descendientes* probablemente no. Esto es, conforme el recorrido avanza desde las raíces, se colorea –inicialmente– a los objetos de gris. Cuando son revisados y se recorren las referencias a sus hijos, son pintados de negro y los hijos son pintados de gris.

En un recolector de copiado, los objetos grises son los objetos en el área sin revisar del *tospace* –si se utiliza un recorrido en amplitud como el de Cheney, Figura 2.4, entonces los objetos grises son los que están entre la búsqueda y las referencias libres. En un recolector de marcado y barrido, los objetos grises corresponden al stack o cola de objetos usados para controlar el recorrido de marcaje y los objetos negros son los que se quitan de la cola. En ambos casos, los objetos que no han sido alcanzados son blancos.

Intuitivamente, el recorrido avanza como una ola con objetos grises al frente, que separa a los objetos blancos (no alcanzados aún) de los objetos negros que han sido pasados por la ola –esto es, no hay referencias directas desde objetos negros hacia objetos blancos Esto nos permite abstraernos de las particularidades del algoritmo de recorrido –puede ser búsqueda en amplitud (BFS), en profundidad (DFS), o cualquier búsqueda exhaustiva Sólo nos importa poder identificar una franja gris bien definida y que el mutante preserve el invariante de que ningún objeto negro tenga una referencia a un objeto blanco

La importancia de esta invariante es que el recolector debe poder suponer que ha "terminado" con los objetos negros, y puede continuar el recorrido de los objetos grises y avanzar la ola Si el mutante crea una referencia de un objeto negro a uno blanco, debe –de alguna manera– notificar al recolector que ha violado su suposición. Esto asegura que

el registro del recolector está al día.

La Figura 3.1 demuestra la necesidad de coordinación. Supongamos que el objeto A ha sido completamente recorrido (y por lo tanto, pintado de negro); sus descendientes han sido alcanzados y pintados de gris. Ahora supongamos que el mutante intercambia la referencia de A a C con la referencia de B a D. La única referencia a D proviene de un campo en A, a quién el recolector ya había recorrido. Si el recorrido continúa sin ninguna coordinación, B será coloreado de negro, C será alcanzado nuevamente por el recorrido (desde B), y D nunca será alcanzado, por lo que será –erróneamente– considerado basura y su espacio será recuperado.

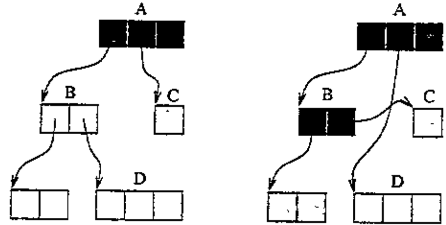


Figura 3.1: Una violación del invariante tricolor

3.2.1 Enfoques incrementales

Hay dos enfoques básicos para coordinar las acciones entre el recolector y el mutante. Uno es usar una *barrera de lectura*, que detecta cuando el mutante intenta acceder una referencia a un objeto blanco, e inmediatamente colorea al objeto de gris; de esta forma, como el mutante no puede leer referencias a objetos blancos, no puede instalar referencias a ellos en objetos negros. El otro enfoque es más directo, e involucra una *barrera de escritura* –cuando el programa intenta escribir una referencia a un objeto, la escritura es atrapada o grabada.

Los enfoques de barrera de escritura, a su vez, se subdividen en dos categorías, dependiendo de qué aspecto del problema resuelven. Para que el recorrido de marcado del recolector tenga un problema como el de la Figura 3.1, es necesario que el mutante:

1. Escriba una referencia a un objeto blanco en un objeto negro, y
2. Destruya la referencia original antes de que el recolector la vea.

Si la primera condición (escribir una referencia en un objeto negro) no se cumple, no se requiere ninguna acción especial –si hay otras referencias al objeto blanco provenientes de objetos grises, el objeto será retenido, en otro caso es basura y no necesitamos retenerlo. Si la segunda condición (borrar la trayectoria original al objeto) no se cumple, el objeto será alcanzado vía la referencia original y por ende, será retenido. Los dos enfoques de barrera de escritura atacan estos dos aspectos del problema.

Los recolectores de *fotografía instantánea a la entrada* aseguran que la segunda condición no pasará –en lugar de permitir que las referencias sean reescritas, primero son salvadas en una estructura de datos "separada" para que el recolector pueda encontrarlas (al entrar sacan una fotografía de la estructura de datos). Por lo tanto, no se puede romper una ruta a un objeto blanco sin proveer otra al recolector de basura.

Los recolectores de *actualización incremental* son más directos al manejar estas referencias problemáticas. Se desea asegurar que si una referencia a un objeto blanco es

copiada a un objeto negro, esa nueva copia de la referencia sea encontrada. Conceptualmente, el objeto negro (o parte de él) se regresa al color gris cuando el mutante "deshace" el recorrido del recolector para indicar que acaba de surgir un hijo que no ha sido revisado. Una forma alternativa es pintar de gris inmediatamente al objeto referenciado. Esto asegura que el recorrido es actualizado en presencia de cambios realizados por el mutante.

Las barreras de lectura y escritura son conceptualmente operaciones de sincronización -antes de que el mutante pueda realizar ciertas operaciones, debe activar al recolector para que éste lleve a cabo alguna acción. En la práctica, esta llamada al recolector de basura sólo requiere una acción relativamente sencilla e incluso el compilador puede emitir las instrucciones adicionales necesarias como parte del código de máquina del mutante. Cada referencia leída o escrita (dependiendo de la estrategia incremental) está acompañada de unas cuantas instrucciones extra que realizan las operaciones del recolector. Dependiendo de la complejidad de la barrera de lectura o escritura, la acción completa de la barrera puede ser ocultada en una llamada a un procedimiento cada vez que se intenta leer o escribir una referencia. Otras estrategias posibles se basan menos en la generación de instrucciones adicionales en el código compilado, y más en la asistencia de operaciones especializadas del hardware o de la memoria virtual.

3.3 Algoritmos con barrera de escritura

Si estamos usando un recolector sin copiado, el uso de una barrera de lectura es un gasto innecesario, ya que no hay necesidad de evitar que el mutante vea una versión inválida de una referencia. Las técnicas de barrera de escritura son más baratas, ya que las escrituras en el heap son menos frecuentes que las lecturas.

3.3.1 Algoritmos de fotografía instantánea

Los algoritmos de *fotografía instantánea* utilizan una barrera de escritura para asegurar que ningún objeto sea inaccesible al recolector de basura cuando la recolección está en proceso. Conceptualmente, al inicio de la recolección de basura, se hace una copia virtual de la gráfica de objetos alcanzables. Esto es, la gráfica de objetos alcanzables se fija al momento en que la recolección de basura comienza, aún cuando el recorrido real procede en forma incremental.

El primer algoritmo de fotografía instantánea se debe a *Abrahamson* y *Patel*, y utilizaba técnicas de memoria virtual *copy-on-write*[1], pero podemos obtener el mismo efecto de manera directa (y de manera eficiente) con una simple barrera de escritura en software.

El algoritmo más simple y conocido de fotografía instantánea es el algoritmo de *Yuasa*. Si una localización es reescrita, el valor sobreescrito se salva y se empuja en un stack de marcaje para su revisión posterior. Esto garantiza que ningún objeto será inalcanzable al recorrido del recolector de basura -todos los objetos que están vivos al inicio de la recolección de basura serán alcanzados, aún cuando las referencias a ellos sean reescritas. En el ejemplo

[1]La técnica de *copy-on-write* permite que no nos tengamos que preocupar por referencias o copias de objetos grandes, ya que tales copias *no* se llevan a cabo sino hasta que son necesarias.

mostrado en la Figura 3.1, la referencia de B hacia D se salva en un stack cuando es reescrita con la referencia a C.

Los esquemas de fotografía instantánea son muy conservadores, porque, de hecho, permiten que la invariante tricolor sea rota, temporalmente, durante el recorrido incremental. En lugar de prevenir la creación de referencias desde objetos negros a objetos blancos, una estrategia global más conservadora previene la pérdida de tales objetos blancos: la *trayectoria original* al objeto no se puede perder, porque todos las referencias reescritas son salvadas y recorridas.

Esto implica que ningún objeto pueda ser liberado durante el recorrido, porque una referencia a un objeto blanco pudo ser insertada en un objeto alcanzable. Esto incluye a los objetos que son creados mientras la recolección está en progreso. Los objetos recién asignados a la memoria son considerados negros, como si ya hubiesen sido recorridos. Esto hace un *corto-circuito* en el recorrido de los objetos nuevos, y de cualquier forma el recorrido no hubiese liberado a ninguno de ellos.

La vista del recolector de la gráfica de objetos alcanzables es la unión de la gráfica al inicio de la recolección de basura con todos aquéllos que fueron puestos en memoria durante el recorrido.

Una característica importante que vale la pena notar acerca de este tipo de algoritmos es que dado que no preservan el invariante tricolor, los objetos grises juegan un papel muy sutil. En lugar de garantizar que *toda* trayectoria de un objeto negro a un objeto blanco pase por un objeto gris, sólo garantizan que para todo objeto alcanzable existirá *al menos una* trayectoria al objeto que pase por otro de color gris.

3.3.2 Algoritmos con barrera de escritura de actualización incremental

Los algoritmos de *fotografía instantánea* y *actualización incremental* son ambos algoritmos con barrera de escritura; sin embargo son muy distintos. Desafortunadamente, los algoritmos de actualización incremental son tratados en la literatura (casi exclusivamente) en términos de sistemas paralelos, en lugar de esquemas incrementales para procesamiento secuencial.

El más conocido de estos algoritmos se debe a *Dijkstra et al.[Dijkstra et al., 1978]*; en éste, en lugar de retener todo lo que se encuentra en una instantánea tomada al inicio de la recolección, heurísticamente (y de alguna forma, conservadoramente) intentamos retener los objetos que están vivos al final de la recolección de basura. Los objetos que mueren durante la recolección –y antes de ser alcanzados por el recorrido de marcaje– no son ni revisados ni marcados. De manera más precisa, un objeto no será alcanzado por el recolector si todas las rutas a él están rotas en puntos que el recolector no ha visto aún. Si una referencia es borrada *después* de ser alcanzada por el recolector, es demasiado tarde, v.gr. si la cabeza de la lista ha sido alcanzada y pintada de gris, y en ese momento se convierte en basura, el resto de la lista seguirá siendo recorrida.

Para evitar el problema de referencias escondidas en objetos alcanzables que ya fueron revisados, cachamos tales referencias cuando son guardadas en los objetos revisados. En lugar de notar cuando una referencia *escapa* de una localidad que no hemos revisado, nos fijamos cuando una referencia es *guardada* en un objeto que ya ha sido recorrido. Si una referencia es reescrita sin ser copiada en otro lugar, mucho mejor –el objeto es basura, por

lo que tal vez no será marcado.

Si la referencia se instala en un objeto que ya hemos determinado vivirá, dicha referencia debe ser tomada en cuenta –en ese momento se incorpora a la gráfica de estructuras alcanzables. Por lo que objetos que ya eran negros con anterioridad serán revisados nuevamente antes de terminar la recolección, para encontrar objetos vivos que de otra forma escaparían. Este proceso puede iterar, ya que más objetos negros pueden ser regresados a sus estados iniciales durante el recorrido. Sin embargo, el recorrido terminará y el recolector eventualmente alcanzará al mutante.[2]

Son posibles muchas variaciones de este algoritmo de actualización incremental, que varían en la implementación de la barrera de escritura y le dan un trato distinto a objetos asignados durante la recolección.

En el esquema incremental presentado en [Dijkstra et al., 1978], suponemos que los objetos son –optimísticamente– inalcanzables cuando son asignados a la memoria. En términos de nuestro marcado tricolor, los objetos son asignados a memoria con color *blanco*, en lugar de negro. En algún punto, el stack debe ser recorrido y los objetos que son alcanzables *en ese momento* son marcados y por lo tanto preservados. En contraste, los esquemas con instantánea deben suponer que tales objetos recién creados están vivos, ya que se pueden instalar referencias a ellos en objetos que ya han sido alcanzados por el recorrido del recolector sin ser detectadas.

Dijkstra decidió, también, que los objetos recién asignados a memoria fueran blancos, bajo la suposición de que los objetos nuevos probablemente vivirán poco y serán reclamados.

Este esquema tiene ventajas significativas potenciales sobre los que asignan nuevos objetos de color negro. La mayoría de los objetos viven muy poco tiempo, por lo que si el recolector no alcanza a dichos objetos al inicio de su recorrido, muy probablemente ya nunca los alcance y al mismo tiempo serán recuperados prontamente. Comparado con el esquema de fotografía instantánea (o el algoritmo de Baker, que describiremos más adelante) hay un costo computacional extra –los objetos recién creados que si viven al final de la recolección deben ser recorridos. Más adelante analizaremos si esta situación vale la pena o no, basándonos en varios factores, tales como la importancia relativa de la eficiencia en el caso promedio y la respuesta en tiempo real *Steele* propuso en [Guy Lewis, 1975] una heurística para asignar algunos objetos con color blanco y otros con color negro, intentando así recuperar rápidamente el espacio ocupado por los objetos de vida muy corta mientras evitamos recorrer a la mayoría de los demás objetos. Sin embargo; la efectividad de esta heurística no ha sido probada, y es difícil de implementar eficientemente en hardware estándar.

El algoritmo de actualización incremental de Dijkstra preserva el invariante tricolor pintando de negro al objeto referido en lugar de regresar el objeto negro al color gris. Intuitivamente, esto empuja la ola gris hacia afuera para mantener el invariante, en lugar de mover la ola hacia atrás. Esto es más conservador que la heurística propuesta por Steele, ya que la referencia puede ser reescrita más adelante, liberando al objeto (por lo que se convertirá en basura flotante). Por otro lado, es más fácil de implementar y mucho más rápido en la práctica; también hace más simple probar la correctez del algoritmo, ya que

---

[2]El algoritmo en [Dijkstra et al , 1978] utiliza una técnica más conservadora que explicaremos más adelante

hay una garantía obvia de progreso en el recorrido.

3.4 Algoritmos con barrera de lectura de Baker

El recolector de basura de tiempo real mejor conocido es el esquema de copiado incremental de *Baker*. Es una adaptación de un esquema de copia simple, como el descrito en la Sección 2.2.4, pero utiliza una *barrera de lectura* para coordinar las acciones del recolector y el mutante. Recientemente, Baker propuso una versión de este algoritmo sin copiado.

3.4.1 Copiado incremental

En el algoritmo de *Baker*, un ciclo de recolección comienza con un *jalón* (*flip*), que conceptualmente invalida a todos los objetos en el *fromspace*, y copia al *tospace* todos los objetos alcanzables desde el conjunto raíz. En este momento se continúa la ejecución del mutante. Cualquier objeto accedido por el mutante en el fromspace es copiado al tospace, y este copiado sobre pedido es obligado por la barrera de lectura. Usualmente la barrera de lectura es implementada con una cuantas instrucciones emitidas por el compilador, que forman una *envoltura* alrededor de las instrucciones de lectura de referencias. El proceso de *localización* se intercala con la ejecución normal del programa, para asegurar que todos los datos alcanzables son copiados al tospace y que el ciclo de recolección se completa antes de que la memoria se agote.

Una característica importante en el esquema de Baker es su manejo de los objetos que son asignados a memoria por el mutante durante la recolección incremental. Estos objetos son asignados al tospace y son tratados como si ya hubiesen sido recorridos –i.e. supone que están vivos. En términos del marcado tricolor, los nuevos objetos son *negros* y ninguno de ellos puede ser recuperado; no son recuperados sino hasta el siguiente ciclo de recolección[3].

Para asegurar que el recolector encuentra todos los datos vivos y los copia al tospace antes de que el área libre se agote, el ritmo de trabajo de recolección está amarrado al ritmo de asignación. Cada vez que se le asigna memoria a un objeto, se incrementa el recorrido y copiado.

En términos de marcado tricolor, el área revisada de tospace contiene objetos negros, y los objetos revisados pero no copiados (que se encuentran entre el apuntador de *búsqueda* y el apuntador *libre*) son grises. Como los objetos inalcanzables en el fromspace son blancos, el recorrido de los objetos (y copiado de sus descendientes) avanza la ola hacia adelante.

El enfoque de Baker es aparear el recorrido de copiado del recolector con el recorrido del mutante de la estructura de datos. Nunca le permitimos al mutante ver referencias dentro del fromspace, i.e., referencias a objetos blancos. Cuando el mutante lee una referencia (potencial) del heap, inmediatamente checa si es una referencia hacia el fromspace; si lo es, el objeto referido es copiado al tospace, i.e., cambiamos su color de blanco a gris. Esto

[3]Baker sugiere copiar objetos vivos viejos en un extremo del tospace, y asignar a los nuevos objetos en el otro extremo. Cada área ocupada en el tospace crece hacia la otra, y los objetos viejos no se mezclan con los nuevos.

avanza, en efecto, la ola gris y mantiene al mutante dentro de ella[4]. La preservación de la invariante tricolor es indirecta —en lugar de revisar si una referencia a un objeto blanco se instala en un objeto negro, la barrera de lectura se asegura de que el mutante no puede ni siquiera ver tales referencias.

Cabe notar que el recolector de Baker cambia las referencias de la gráfica de objetos alcanzables durante el proceso de copiado. La barrera de lectura no sólo le informa al recolector sobre los cambios que hace el mutante, para asegurar que no se pierdan objetos; también previene que el mutante vea inconsistencias temporales creadas por el recolector. Si no hiciera esto, el mutante podría encontrar dos referencias distintas a versiones del mismo objeto, una de ellas obsoleta.

La barrera de lectura puede ser implementada en software, precediendo cada lectura (de una referencia potencial en el heap) con un chequeo y una llamada condicional a la rutina *copia-y-actualiza*. Por lo tanto, el código compilado contiene instrucciones extras para implementar la barrera de lectura. De manera alternativa, puede ser implementado con chequeos de hardware especializado o rutinas en microcódigo.

La barrera de lectura es costosa en hardware común, ya que en el caso general, cualquier cargado de una referencia debe revisar si la referencia apunta a un objeto en el fromspace (blanco); si es así, debe ejecutarse el código extra para mover el objeto al tospace y actualizar la referencia. El costo de estos chequeos es alto en el hardware convencional, porque ocurren muy frecuentemente[5].

Frederick P. Brooks propuso una variante más al esquema de Baker, en la cual los objetos siempre *están* referidos por medio de un campo de indirección embebido en el objeto mismo. Si un objeto es válido, su campo de indirección apunta a sí mismo. Si es un objeto obsoleto, su campo de indirección apunta a la nueva versión. La indirección incondicional es más barata que revisar las indirecciones; sin embargo en estudios prácticos realizados se ha mostrado que, en hardware normal, un sistema se ve afectado seriamente en su rendimiento por cualquiera de estos esquemas.

3.4.2 Algoritmo incremental sin copiado de Baker –La rueda de ardilla (treadmill)

Esta versión sin copiado del esquema de Baker, utiliza listas doblemente ligadas (y campos de color por objeto) para implementar conjuntos de objetos de cada color, en lugar de mantener áreas de memoria separadas. Este esquema ofrece menos restricciones en la implementación del lenguaje al no mover objetos o actualizar referencias[6].

La asignación de nuevos objetos a memoria se realiza en la lista *new* –es contigua a la lista *free*, y la asignación se lleva a cabo recorriendo el apuntador que las separa. Al

[4]Una variante de este algoritmo actualiza las referencias sin copiar los objetos —el copiado es *perezoso*— y simplemente se reserva el espacio en el tospace antes de actualizar la referencia. Esto hace más simple la tarea de ofrecer límites de tiempo más pequeños para todas las operaciones de lista

[5]En arquitecturas especializadas, como las máquinas *Lisp* –que utilizan una variante de Lisp como su conjunto de instrucciones de máquina– hay hardware especializado para detectar referencias que entran al fromspace, las atrapan y mandan a un manejador

[6]En particular, hace posible que utilicemos este esquema de recolección en lenguajes que son incapaces de identificar sin ambiguedad referencias en el stack, lo que imposibilita el uso de un esquema de recolección de copiado simple

inicio de la recolección, el segmento ocupado por *new* está vacío.

La lista *from* mantiene objetos que fueron asignados antes de que la recolección empezara y que están sujetos a la recolección. Conforme el recolector y el mutante recorren la estructura de datos, los objetos son movidos de la lista *from* a la lista *to*. La lista *to* inicialmente también está vacía, pero crece conforme se desligan objetos de la lista *from* durante la recolección.

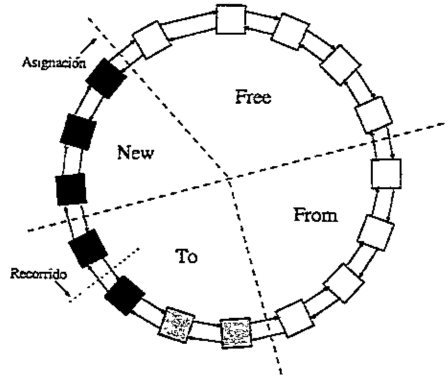


Figura 3.2: Recolector treadmill durante la recolección

La lista *new* contiene objetos nuevos, que son asignados con color negro. La lista *to* contiene tanto objetos negros (que han sido completamente revisados) como objetos grises (que han sido alcanzados por el recorrido pero no revisados completamente). Nótese el isomorfismo que existe con el algoritmo de copiado. Sólo necesitamos un apuntador de búsqueda en la lista *to* que avance sobre los objetos grises.

Eventualmente, todos los objetos alcanzables en la lista *from* son movidos a la lista *to* y sus descendientes son revisados. Cuando ya no hay descendientes alcanzables, todos los objetos en la lista *to* son negros y los objetos restantes son basura. En este punto, la recolección de basura termina. La lista *from* puede ser reutilizada y simplemente la fusionamos con la lista *free*. Las listas *new* y *to* contienen objetos que deben ser preservados y pueden ser fusionadas para formar una nueva lista *to* en el siguiente ciclo de recolección[7].

El estado actual es muy similar al del inicio del ciclo anterior, pero los segmentos se han "movido" sobre el círculo –de aquí surge el nombre de la rueda de la ardilla (treadmill).

3.4.3 Conservadurismo de la barrera de lectura de Baker

Los recolectores de basura de Baker utilizan dos formas de aproximaciones conservadoras para determinar que tan vivo está un objeto. La más obvia es que los objetos recién asignados a la memoria son marcados como vivos, incluso si mueren antes de que la recolección termine. La segunda es que los objetos pre-existentes pueden convertirse en basura después de ser alcanzados por el recorrido del recolector, y no serán recuperados –una vez que un objeto ha sido pintado de gris se considera vivo hasta el siguiente ciclo de recolección. Por otro lado, si los objetos se convierten en basura durante la recolección y todas las trayectorias a esos objetos son destruidas *antes* de ser alcanzados, serán recuperados. Por lo tanto, el esquema incremental de Baker actualiza la gráfica de objetos pre-existentes alcanzables incrementalmente, pero sólo cuando las referencias de objetos grises son modificadas por el mutante. Modificaciones a referencias provenientes de objetos negros no tiene ninguna relevancia para el conservadurismo ya que todos los objetos referidos son grises. El

[7]La discusión del algoritmo de Baker ha sido simplificada. En el algoritmo original, Baker utiliza cuatro colores y permite cambiar el color a todos los miembros de una lista instantáneamente cambiando el sentido de los patrones de bits, en lugar de cambiar los patrones directamente.

grado de conservadurismo (y de basura flotante) depende de los detalles del recorrido del recolector y de las acciones del mutante.

3.5 Recolección con copiado de replicación

Recientemente, una nueva clase de recolector incremental de copiado fue desarrollada, el ***copiado de replicación***, que es diferente al esquema de copiado incremental de Baker. Recordemos que en el recolector de Baker, la recolección comienza con un *flip*, que copia los datos inmediatamente alcanzables al *tospace*, e invalida el *fromspace*; a partir de ese momento, el mutante sólo puede ver las versiones *nuevas* de los objetos, nunca las versiones viejas en el *fromspace*.

El copiado de replicación es casi lo contrario. Mientras el copiado se lleva a cabo, el mutante sigue viendo las versiones de los objetos en el *fromspace*, en lugar de las "réplicas" en el *tospace*. Cuando se completa el proceso de copiado, se realiza un flip, y el mutante comienza a ver las réplicas.

Los detalles de consistencia en el copiado de replicación son muy distintos a los que teníamos en el copiado de Baker. El mutante sigue accediendo a las mismas versiones de objetos durante el recorrido de copiado, así que no necesita revisar referencias de actualización. Esto elimina la necesidad de la barrera de lectura –conceptualmente, todos los objetos son *actualizados* a sus nuevas versiones de un solo golpe, cuando se ejecuta el flip.

Por otro lado, esta estrategia requiere una barrera de escritura, la cual debe lidiar con más problemas que simples actualizaciones de referencias. En el recolector de Baker, el mutante sólo ve las nuevas versiones de objetos, por lo que una escritura en un objeto actualiza la versión actual (tospace). En el copiado de replicación, el mutante siempre ve las versiones viejas en el fromspace; si un objeto ha sido copiado al tospace, y después el mutante modifica la versión vieja en el fromspace, la réplica tendrá valores incorrectos (viejos) –queda fuera de sincronía con la versión que el mutante está viendo.

Para evitar esto, la barrera de escritura debe interceptar *todas* las actualizaciones, y el recolector debe asegurarse que dichas actualizaciones sean propagadas a las réplicas adecuadas cuando se haga el flip. Esto es, todas las modificaciones a las versiones viejas de los objetos se deben hacer a los objetos nuevos, para que el mutante vea los valores correctos después del flip.

Esta barrera de escritura resulta muy costosa para la mayoría de los lenguajes de programación, pero no para los lenguajes funcionales, o "casi funcionales" (tales como ML) donde los efectos secundarios se permiten pero no son usados frecuentemente.

3.6 Conservadurismo y coherencia

Como hemos mencionado en la sección 3.1, los recolectores incrementales pueden tomar distintas alternativas para coordinar el mutante con el recorrido del recolector. Si estos procesos *quasi-paralelos* se coordinan muy estrechamente, sus vistas de las estructura de datos pueden ser muy precisas, pero los costos de coordinación pueden ser inaceptables. Si no se coordinan muy estrechamente, pueden utilizar información no actualizada, y retener objetos que se han convertido en basura durante la recolección

3.6.1 Coherencia y conservadurismo en recolección sin copiado

Los algoritmos sin copiado con barrera de escritura que hemos descrito caen en puntos distintos del espectro de efectividad y conservadurismo. Los algoritmos de fotografía instantánea tratan todo conservadoramente, reduciendo su efectividad. El algoritmo incremental de Dijkstra et al. es menos conservador que los de fotografía instantánea, pero más conservador que el algoritmo de Steele.

En el algoritmo de Steele, si una referencia a un objeto blanco se guarda en un objeto negro, el objeto blanco *no* es pintado de gris inmediatamente –en su lugar, el objeto negro en el cual se insertó la referencia es regresado al color gris, "deshaciendo" lo que el recolector había hecho. Esto significa que si se vuelve a modificar el campo de referencia en el objeto gris (que era negro) probablemente el objeto blanco referido se convierta en basura y el recolector podrá recuperarlo en el ciclo actual de recolección. En contraste, el algoritmo de Dijkstra hubiera pintado de gris al objeto blanco, por lo que no sería recuperado en este ciclo.

La diferencia entre ambos algoritmos puede parecer trivial, pero es fácil imaginar un escenario en el cual se convierte en algo muy importante. Consideremos un programa que guarda la mayoría de sus datos en stacks, implementados como listas ligadas. Si un objeto stack es alcanzado por el recolector y pintado de negro, y después muchos objetos son empujados y luego sacados del stack, el algoritmo de Dijkstra no recuperará *ninguno* de los objetos que se sacaron inmediatamente del stack –dado que el campo de referencia a la lista de objetos en el stack se va modificando cada vez que un nuevo elemento entra al stack, por lo que cada elemento será pintado de gris cuando el anterior sea sacado. El algoritmo de Steele, por otro lado, puede recuperar casi todos los objetos sacados, ya que el campo de referencia puede ser modificado muchas veces antes de que el recorrido del recolector lo vuelva a revisar.

Notemos que este espectro de conservadurismo (algoritmos de fotografía instantánea, Dijkstra y Steele) es un orden lineal si los algoritmos utilizan el mismo algoritmo de recorrido, organizado de la misma forma con respecto al comportamiento del programa –y esto es muy difícil de verificar en la práctica. Los detalles del orden de las acciones del mutante y el recolector determina cuanta basura flotante será retenida. Cualquiera de estos recolectores retendrá cualquier dato alcanzable vía las trayectorias que ya han sido recorridas por el recolector antes de ser rotas por el mutante.

Esto sugiere que la gráfica de objetos alcanzables puede ser recorrida *oportunísticamente*, i.e. el costo total puede ser reducido si ordenamos cuidadosamente el recorrido de objetos grises. Por ejemplo, puede ser deseable evitar tanto tiempo como sea posible el revisar partes que cambian muy rápido de la gráfica, para evitar alcanzar objetos que se convertirán en basura muy pronto.

Si consideramos que todo lo expuesto es idéntico para los distintos esquemas (i.e. en presencia de oportunismo y suerte), el algoritmo de fotografía instantánea es más conservador (por lo tanto, menos efectivo) que el algoritmo de actualización incremental, y el algoritmo incremental de Dijkstra es más conservador que el de Steele.

3.6.2 Coherencia y conservadurismo en recolección con copiado

El algoritmo con barrera de lectura de Baker no cae dentro del espectro anterior. Es menos conservador que el algoritmo de fotografía instantánea, ya que una referencia en un objeto gris puede ser modificada y aún así no será recorrida; es más conservador que los algoritmos de actualización incremental; sin embargo, dado que cualquier cosa alcanzada por el mutante es pintada de gris –los objetos no se pueden convertir en basura, desde el punto de vista del recolector, después de que el mutante los ha *tocado* mientras el ciclo actual de recolección se está llevando a cabo.

El algoritmo de copiado de replicación (al igual que un algoritmo de actualización incremental), es capaz de recuperar objetos que se vuelvan inalcanzables, porque una referencia puede ser modificada antes de ser alcanzada por el recolector. Este recolector es menos conservador que el de Baker, en parte porque puede utilizar una noción más débil de consistencia. El mutante no opera sobre el tospace hasta que se completa la fase de copiado, por lo que las réplicas en el tospace no necesitan ser *totalmente* consistentes durante el copiado incremental. Los cambios realizados en el fromspace por el mutante deben reflejarse en las réplicas del tospace eventualmente, pero requerimos consistencia total sólo al final de la recolección, cuando el flip atómico es realizado. Al igual que otros algoritmos de barrera de escritura, el copiado de replicación puede beneficiarse significativamente de un ordenamiento oportunista del recorrido.

3.6.3 Recolección "radical" y recorrido oportunista

Los algoritmos de recorrido que hemos descrito caen en un espectro de conservadurismo decreciente, de la siguiente forma:

- Fotografía instantánea con barrera de escritura
- De actualización incremental con barrera de lectura
- Barrera de lectura de Baker
- Barrera de escritura de Dijkstra
- Barrera de escritura de Steele

Considerando este espectro, es interesante preguntarnos ¿hay algo menos conservador que el algoritmo de Steele? Esto es, ¿podemos tener un recolector mejor informado que el de Steele, uno que responda más agresivamente a cambios en la gráfica de objetos alcanzables? La respuesta es sí Tal recolector de basura estaría dispuesto a re-hacer parte del recorrido ya hecho, quitar marcas de objetos que fueron alcanzados previamente, en pos de evitar el conservadurismo. Nos referimos a esto como una estrategia *radical* de recolección de basura. A primera vista, un recolector así puede parecer poco práctico, pero bajo ciertas circunstancias, aproximaciones a esta estrategia pueden ser muy atractivas.

Para ser menos conservador debe responder a cualquier cambio en la gráfica de objetos alcanzables, desmarcando objetos previamente alcanzados, para que toda la basura sea detectada Podemos llamar a esto un *recolector totalmente radical.*

Una forma de hacer esto es realizar un recorrido completo de la gráfica de objetos alcanzables cada vez que una referencia es escrita por el mutante. Naturalmente, esta opción es excesivamente costosa. Un recolector normal no-incremental puede ser visto como una aproximación de esto; la gráfica es recorrida "instantáneamente" deteniendo al mutante mientras el recorrido se lleva a cabo, pero sólo se hace ocasionalmente.

Otra forma de alcanzar una recolección totalmente radical es guardar todas las dependencias en la gráfica de objetos alcanzables, y actualizar la base de datos de dependencias cada vez que se hace una actualización de referencia. Cuando todas las trayectorias que mantienen vivo a un objeto son rotas, el objeto es basura. Nuevamente, una implementación de esta estrategia sería poco práctica para la mayoría de los recolectores de propósito general, ya que la base de datos de dependencias puede ser muy grande y por ende las actualizaciones de referencias serían muy costosas.

Notemos que algunas aproximaciones a esta información de dependencias puede ser relativamente barata, de hecho, el conteo de referencias es justamente eso: una aproximación a esta información de dependencia. Un conteo de referencias es una aproximación conservadora del número de trayectorias a un objeto, donde el conteo de referencias puede llegar a cero y permitir que el objeto sea recuperado inmediatamente. Algunos algoritmos distribuidos de recolección de basura también realizan una especie de recolección radical, recalculando muy frecuentemente algunas partes del recorrido del recolector.

3.7 Comparación de técnicas incrementales

Al comparar distintos recolectores, debemos tomar en cuenta que la abstracción del marcado tricolor es distinta a los mecanismos concretos para recorrido tales como marcado y barrido o recolección de copiado. La selección de una barrera de lectura o una de escritura (y la estrategia para asegurar la correctez) es en buena medida independiente de la selección de los mecanismos de recorrido y recuperación.

La selección de un esquema con barrera de lectura o de escritura debe tomarse considerando el hardware disponible. Sin soporte de hardware especializado, una barrera de lectura es más fácil de implementar eficientemente, ya que escribir referencias en el heap es menos frecuente que recorrer referencias. Si contamos con soporte para memoria virtual y no requerimos soporte para aplicaciones en tiempo real, una barrera de lectura que utilice inteligentemente la paginación es nuestra mejor opción.

De los esquemas de barrera de escritura, los algoritmos de fotografía instantánea son significativamente más conservadores que los algoritmos de actualización incremental. Esta ventaja de los algoritmos de actualización incremental puede ser acrecentada escogiendo cuidadosamente el orden del recorrido sobre el conjunto raíz, recorriendo primero las estructuras más estables para evitar que el mutante *deshaga* el trabajo del recolector.

Los esquemas de actualización incremental incrementan la efectividad, pero también pueden incrementar los costos. En el peor caso, toda la basura que se convierte en basura durante la recolección "flota", i.e. se convierte en inalcanzable muy tarde, y el recolector la recorre y la mantiene hasta el siguiente ciclo de recolección. Si los objetos nuevos son asignados a memoria con color blanco (sujetos a ser recuperados), los algoritmos de actualización incremental pueden ser considerablemente más costosos que los algoritmos de fotografía ins-

tantánea, ya que en el peor caso es posible que todos los objetos nuevos floten y requieran ser recorridos, sin ningún incremento en la cantidad de memoria recuperada —discutiremos esto más adelante, en la sección 3.8.2.

Debemos prestar especial atención a la implementación de las barreras de escritura. Algunas implementaciones utilizan los *dirty bits* de la memoria virtual como parte central de la barrera de escritura. Todos los objetos negros en una página deben ser revisados nuevamente si la página está *sucia* antes de que termine la recolección. Esto sacrifica garantías de tiempo real, pero soporta paralelismo.

Se puede mantener una lista de localidades de guardado, o bien dirty bits (en software) para pequeñas áreas de memoria y con esto lograr reducir los costos de búsqueda y limitar el tiempo que toma la actualización del recorrido de marcado. Esto se ha hecho en recolectores de basura generacionales por otros motivos, como lo discutiremos en el siguiente capítulo.

3.8 Recolección de recorrido en tiempo real

Los recolectores incrementales se diseñan -generalmente- para ser *tiempo real*, i.e. para imponer retardos estrictamente limitados en la ejecución de un programa, para que los programadores puedan garantizar que sus programas con recolección de basura cumplirán los tiempos planeados. Las aplicaciones en tiempo real son muy variadas, incluyendo controladores para procesos industriales, probar y monitorear equipo, procesamiento audiovisual, controladores de vuelo automáticos y equipo de telefonía. Las aplicaciones de tiempo real se dividen en dos clases:

Tiempo real duro: Los cálculos se deben completar en límites de tiempo estrictamente establecidos.

Tiempo real blando: Es aceptable que algunas tareas no "lleguen" a tiempo algunas veces, siempre y cuando esto no ocurra frecuentemente[8]

En esta sección, atacaremos únicamente los problemas relacionados con tiempo real duro.

El criterio para recolectar basura en tiempo real se puede enunciar como sigue: *imponer solamente retardos pequeños y acotados a cualquier operación particular de un programa.* Por ejemplo, recorrer una referencia nunca debe tardar más de un microsegundo, asignar un objeto pequeño al heap no debe tardar más de cuatro microsegundos, etc.

Hay dos problemas con este criterio. Un problema es que la noción apropiada de "pequeño" es dependiente de la naturaleza de una aplicación. Para algunas aplicaciones, es aceptable tener respuestas que sean retardadas por una fracción significativa de un segundo, o incluso por varios segundos. Para otra aplicación, un retardo de uno o dos milisegundos puede ser tolerable, mientras que para otras, un retardo de unos cuantos microsegundos puede ser fatal. Por un lado, considere un controlador de un sintetizador musical, en el cual

[8] Por ejemplo, en una sistema de telefonía digital, hacer una conexión puede ser una tarea de tiempo real blando, pero una vez que la conexión se ha establecido, entregar audio continuo puede ser una tarea de tiempo real duro

la apreciación humana no notará un retardo de un milisegundo; por el otro, considere el sistema de guía de alta precisión para misiles anti-misiles.

Otro problema con este tipo de criterio es que enfatiza, de manera poco realista, el papel de las operaciones "pequeñas" de un programa. Cuando presionamos una tecla del teclado de un sintetizador, el controlador puede requerir ejecutar miles de enunciados de un programa, v.gr. decidir qué nota debe tocar, qué nivel debe asignar a los distintos tonos, qué componentes sonoros mezclar y en qué proporciones, etc.

Por lo tanto, para la mayoría de las aplicaciones, un requerimiento para alcanzar el desempeño deseado en tiempo real es que la aplicación siempre pueda utilizar el CPU por una fracción de tiempo en una *escala* relevante para la aplicación. Naturalmente, la fracción relevante dependerá tanto de la aplicación como de la velocidad del procesador.

Para la computadora de control de procesos en una fábrica de químicos, puede ser suficiente que el proceso de control pueda ser ejecutado uno de cada dos segundos, ya que el controlador debe responder a cambios (v.gr. en la temperatura de los contenedores) en un tiempo no mayor a dos segundos, y un segundo es suficiente para calcular la respuesta apropiada. Por otro lado, un controlador para un sintetizador musical puede requerir que el CPU ejecute su programa de control por espacio de medio milisegundo cada dos milisegundos, para mantener los retardos dentro del umbral de lo imperceptible para el oído humano.

Notemos que cualquiera de estas aplicaciones puede funcionar correctamente si el recolector la detiene por un cuarto de milisegundo de vez en cuando. Siempre y cuando estas pausas no sean muy frecuentes, son tan pequeñas que no provocan ningún impacto negativo en el desempeño de la aplicación y ésta podrá cumplir sus plazos especificados.

Pero supongamos que las pausas están agrupadas en el tiempo; si ocurren frecuentemente destrozarán la habilidad de la aplicación para cumplir sus plazos, simplemente porque consumirán una fracción grande del tiempo del CPU. Si la aplicación sólo se ejecuta por espacios de un decimosexto de milisegundo entre cada pausa de un cuarto de milisegundo, no puede obtener más de un quinto del tiempo de CPU. En este caso, cualquiera de los dos programas no podrá cumplir sus requerimientos de tiempo real, ni siquiera aquel que requiere responder en un plazo de dos segundos.

Como describimos en secciones anteriores, algunos recolectores de copiado utilizan protecciones de memoria virtual para disparar una revisión inteligente de páginas de memoria virtual, y esta relación tan estrecha puede provocar que los requerimientos de una aplicación de tiempo real no se cumplan. En el peor de los casos, recorrer una lista de miles de elementos puede provocar que mil páginas sean revisadas, lo que provocará un trabajo considerable por parte del recolector de basura, incluyendo la sobrecarga que representa la paginación. De esta forma, un recorrido que ejecutaría normalmente unos cuantos miles de instrucciones puede, inesperadamente, ejecutar millones, incrementando el tiempo para recorrer la lista varios ordenes de magnitud. La localidad de referencia puede hacer tal situación improbable, pero la probabilidad de casos malos, como éste, no es despreciable.

Desafortunadamente, utilizar un recolector incremental más fino no soluciona el problema tampoco. Consideremos la técnica de copiado de Baker. El tiempo para recorrer la lista depende de si los elementos deben ser re-localizados en el tospace o no. Recorrer una simple referencia, puede requerir que el objeto sea copiado; esto incrementa el costo de esa referencia a memoria en un orden de magnitud *aún si los objetos son pequeños y*

tenemos soporte directo en hardware. Consideremos la copia de una celda CONS de Lisp, que consiste de un encabezado, un campo CAR, y un campo CDR. Al menos necesitamos tres lecturas y tres escrituras de memoria, más instrucciones extra para instalar un apuntador de expedición, ajustar el apuntador libre y –muy probablemente– para brincar hacia la subrutina que hace este trabajo y de regreso al recolector. En tales casos, es posible que la sobrecarga del recolector consuma hasta un 90% del tiempo total del CPU, reduciendo el poder computacional –esto es, el poder disponible garantizado para alcanzar los plazos de tiempo real– en un orden de magnitud.

Por lo tanto, para decidirnos por una estrategia de recorrido en tiempo real, es importante decidir qué tipo de garantía(s) necesitamos, y en qué escala. El algoritmo de Baker es el más conocido de entre los de actualización incremental; sin embargo, puede ser el menos apropiado para la gran mayoría de aplicaciones en tiempo real, porque su rendimiento es muy impredecible en escalas de tiempo pequeñas. Los algoritmos con barrera de escritura son más adecuados, ya que tienen una relación menos estrecha entre el mutante y el recolector. Es más fácil para un programador razonar acerca de garantías en tiempo real si sabe que el recorrer una referencia siempre tarda un tiempo constante, independientemente de si la referencia ha sido alcanzada o no por el recolector. Los algoritmos de barrera de escritura requieren de más trabajo por escritura de referencia, pero el trabajo por operación de programa es menos variable, y en su mayor parte, dicho trabajo no tiene que ser ejecutado inmediatamente para mantener la correctez.

Desafortunadamente, a pesar de que los algoritmos sin copiado tienen la conveniente propiedad de que su sobrecarga en tiempo es predecible, sus costos de espacio son mucho más difíciles de entender. Un algoritmo de copiado generalmente libera un área grande y contigua de memoria, y las solicitudes de memoria para objetos de tamaños variados puede satisfacerse en una operación como de stack en un tiempo constante. Los algoritmos sin copiado están sujetos a fragmentación –la memoria que es liberada puede no ser contigua, por lo que puede ser imposible asignar un objeto de un tamaño dado, aún cuando exista la cantidad necesaria de memoria disponible.

En las siguientes sub secciones discutiremos técnicas para obtener rendimiento en tiempo real a partir de recolectores de recorrido. Suponemos que el sistema es puramente tiempo real duro –esto es, que el sistema consiste sólo de cálculos que *deben* completarse antes de sus plazos; también suponemos que sólo hay una escala de tiempo para los plazos de tiempo real. En tales sistemas, la meta principal es hacer que el rendimiento del peor caso sea tan bueno como sea posible También suponemos que usamos un algoritmo de copiado o que todos los objetos son de tamaño uniforme[9]. Esto nos permite suponer que cualquier solicitud de memoria puede ser satisfecha por cualquier espacio de memoria disponible, e ignorar la posible fragmentación de área de almacenamiento disponible.

3.8.1 Recorrido del conjunto raíz

Un determinante importante del rendimiento en tiempo real es el tiempo que se lleva el revisar el conjunto raíz. Recordemos que en el recolector incremental de Baker,

[9]En algunos sistemas, es posible fragmentar transparentemente los objetos al nivel del lenguaje en bloques de tamaño fácilmente manejables, para hacer que la recolección de basura sea más fácil y reducir o eliminar los problemas de fragmentación

el conjunto raíz es actualizado, y los objetos inmediatamente alcanzables son copiados al tospace, en una sola operación atómica, ininterrumpible por la ejecución del mutante. Esto significa que –ocasionalmente– habrá una pausa proporcional al tamaño del conjunto raíz. Esta pausa es muy probable que sea mayor que una pausa del recorrido incremental, y puede constituir la limitación principal para dar garantías de tiempo real.

Pausas similares ocurren en los algoritmos de actualización incremental cuando intentan terminar una recolección. Antes de que la recolección se considere terminada, el conjunto raíz debe ser recorrido (junto con cualquier objeto gris guardado por la barrera de escritura, en el caso del algoritmo de Steele), y todos los datos alcanzables deben ser recorridos y pintados de negro atómicamente. Con esto aseguramos que ninguna referencia fue escondida en alguna raíz previamente revisada. Si este trabajo no puede ser realizado dentro de los límites de tiempo real, el recolector debe suspenderse, el mutante continúa, y el proceso de terminación completo debe volverse a intentar más tarde. Los algoritmos de fotografía instantánea no presentan tal problema ya que ninguna trayectoria puede ocultarse al recolector.

Una forma de limitar el trabajo requerido para un flip o para la terminación es mantener el conjunto raíz pequeño. En lugar de considerar todas las variables globales y locales como parte del conjunto raíz, algunas o todas pueden ser tratadas como objetos en el heap. Las lecturas o escrituras de estas variables serán detectadas por la barrera de lectura o escritura, y el recolector mantendrá la información relevante de manera incremental.

El problema con mantener el conjunto raíz pequeño es que el costo de una lectura o escritura sube proporcionalmente –un gran número de variables están protegidas por una barrera de lectura o escritura, lo que agrega una sobrecarga cada vez que son leídas o modificadas. Un posible intercambio es abolir el costo de la barrera (de lectura o escritura) para las variables asignadas en los registros, y revisar (solamente) el conjunto de registros de manera atómica cuando sea necesario. Si hay muchas operaciones en las que intervengan las variables asignadas en el stack se frenará significativamente la ejecución. En este caso, el stack completo puede ser revisado atómicamente. Esto puede sonar como algo muy costoso; sin embargo, la mayoría de los programas de tiempo real nunca tienen stacks de activación muy profundos o ilimitados, y el costo puede ser insignificante a la escala de los tiempos de respuesta requeridos para el programa. De manera similar, para sistemas pequeños con procesadores rápidos (o con escalas de tiempo relativamente grandes para los requerimientos de tiempo real) puede ser deseable evitar la barrera de lectura o escritura para todas las variables globales, y revisarlas atómicamente (igual que las que están en los registros). También son posibles estrategias intermedias, tratando algunas variables de alguna forma y otras de otra forma.

3.8.2 Garantía de progreso suficiente

La sección previa se enfoca en asegurar que el recolector no use mucho tiempo de CPU, considerando la escala de tiempo relevante, lo que evitaría que el proceso alcanzara sus plazos establecidos. Inversamente, el recolector tiene su propio plazo de tiempo real que cumplir: debe terminar su recorrido y liberar más memoria antes de que la memoria libre actual se agote. Si no lo hace, la aplicación tendrá que parar y esperar a que el recolector termine y libere más memoria.

Para programas de tiempo real duro, por lo tanto, debe existir una forma de asegurar que el recolector obtiene el tiempo suficiente de CPU para terminar su tarea antes de que la memoria disponible se agote, aún en el peor de los casos. Para proveer tal garantía, es necesario calificar el peor caso –esto es, poner algún límite en lo que el recolector debe poder hacer. Dado que un recolector de recorrido debe recolectar datos vivos, esto significa poner una cota en la cantidad de datos vivos. En general, el programador de una aplicación debe asegurar que el programa no tiene más de una cierta cantidad de datos vivos que recorrer, y el recolector puede determinar que tan rápido debe operar para alcanzar sus plazos. Por ejemplo, puede determinar si requiere más tiempo de CPU que la cantidad que se le permite consumir. Naturalmente, esto permite hacer algunos trueques en las configuraciones de los parámetros. Si se tiene más memoria disponible, el recolector generalmente requiere sólo una pequeña fracción del tiempo del CPU para asegurar que termina antes de que la memoria se agote.

La estrategia usual para asegurar que la memoria libre no se agotará es utilizar un *reloj de asignación de memoria* –por cada unidad de asignación, una unidad correspondiente de recolección se lleva a cabo, donde esta última es lo suficientemente grande para asegurar que el recorrido se completa antes de que el espacio libre se agote. La forma más simple de esto es cargar el trabajo de recolección directamente a la rutina de asignación de memoria –cada vez que un objeto es asignado a memoria, una cantidad proporcional de recolección de basura se lleva a cabo. Esto garantiza que no importando que tan rápido un programa asigne objetos a la memoria, el recolector se acelerará en la misma proporción. En implementaciones reales, el trabajo usualmente se realiza usando esta técnica pero con unidades más grandes, donde por cada cierto número de objetos asignados a memoria se ejecuta una unidad de recolección (por razones de eficiencia).

En el resto de esta sección mostraremos como calcular el ritmo de recorrido de recolección mínimo para asegurar que el espacio libre no se agotará antes de completar la recolección. Empezaremos con un recolector de fotografía instantánea sin copiado, que asigna objetos nuevos de color negro (i.e. no sujetos a recolección). Hacemos la suposición de que todos los objetos tienen un tamaño uniforme, por lo que un solo bloque de memoria nos basta para asignar y regresar ahí la memoria liberada. Después de describir este caso sencillo, explicaremos como difiere de otros algoritmos de recorrido incremental.

En el algoritmo de fotografía instantánea, al final de la recolección, todos los datos vivos al inicio, ya debieron ser revisados. En el peor de los casos, otros algoritmos también hacen esto, ya que los objetos se revisan aún si éstos se liberan durante la recolección. En ausencia de información extra por parte del programador, el recolector debe suponer –en general– al inicio de la recolección, que la máxima cantidad de información viva está en realidad viva.

Como supusimos que los objetos asignados durante la recolección son negros, i.e. no están sujetos a recuperación, no necesitamos recorrerlos –esos objetos serán ignorados hasta el siguiente ciclo de recolección.

A primera vista, parece que para la máxima cantidad de datos vivos L y un tamaño de memoria M, tendríamos $(M - L)$ memoria disponible para asignar –esto implica que el ritmo mínimo de recorrido seguro es $(M - L)/L$, para recorrer L cantidad de datos antes de que nuestro espacio libre se agote. Desafortunadamente, también debemos tomar en cuenta

la basura flotante. Los datos que están vivos al inicio de la recolección pueden convertirse en basura durante la recolección, pero demasiado tarde como para ser recuperados durante el ciclo actual de recolección. Los datos que recién hemos asignado, también puede ser basura, pero como los asignamos con color negro, no lo podemos saber. Si usamos $(M-L)$ memoria, tal vez no recuperaremos *nada* del espacio ocupado, y no tendríamos ningún espacio libre para intentar una nueva recolección. Por lo tanto, la máxima cantidad de datos que debemos asignar es la mitad de nuestra memoria disponible, o bien $(M-L)/2$. El ritmo mínimo de recorrido seguro nos permite asignar esa cantidad en el mismo tiempo que nos lleva recorrer la máxima cantidad de datos vivos, por lo tanto, el ritmo es $((M-L)/2)/L$, o bien, $(M-L)/2L$. Esto es suficiente para el peor caso, en el cual toda la basura flota durante el ciclo completo de recolección, pero será reclamada en el siguiente ciclo.

Como lo mencionamos antes, la situación es en esencia la misma para otros algoritmos de recorrido incremental, siempre y cuando asignen a sus nuevos objetos de color negro, ya que en el peor de los casos retienen todos los objetos al igual que el algoritmo de fotografía instantánea. El ritmo mínimo de recorrido seguro es proporcional a la cantidad de datos vivos e inversamente proporcional a la cantidad de memoria disponible; por lo tanto tiende a cero conforme la memoria se hace grande con respecto a la cantidad máxima de datos vivos.

Cuando asignamos los nuevos objetos con color blanco, la situación se torna más difícil, ya que estamos apostando que los nuevos objetos son de vida muy corta; por lo que los sometemos a la recolección de basura con la esperanza de recuperar su espacio rápidamente en el ciclo actual de recolección. Esto nos obliga a recorrer los objetos blancos alcanzables, lo cual en el peor de los casos equivale a recorrer *todo lo que recién* asignamos antes de que se convierta en basura. Aún cuando suponemos que hay un límite en la cantidad de datos vivos (información provista por el programador), debemos tomar en consideración el conservadurismo del proceso de recorrido, y el hecho de que las referencias puedan ser recorridas por el recolector antes de que sean destruidas por el mutante.

Por lo tanto, cuando asignamos objetos con color blanco, el peor caso para el ritmo mínimo de recorrido no se aproxima a cero conforme la memoria se hace grande –se aproxima al ritmo de asignación; el recorrido se debe mantener al nivel del ritmo de asignación, y tal vez ir un poco más rápido, para asegurar que eventualmente se pondrá al corriente– siempre debemos recorrer, al menos, tan rápido como asignamos.

El análisis expuesto arriba se aplica a recolectores sin copiado con objetos de tamaño uniforme. En recolectores de copiado, se requiere más memoria para mantener las nuevas versiones de los objetos que se están copiando; deben existir al menos otras L unidades de memoria disponible, para asegurar que aún en el peor de los casos el *tospace* no se agota antes de que el *fromspace* sea recuperado. Esto es un costo muy alto si L es relativamente grande con respecto a la cantidad real de memoria disponible en el sistema. En recolectores sin copiado para objetos de tamaños no uniformes, debemos considerar la fragmentación. La fragmentación reduce la memoria disponible efectiva, lo que requiere un recorrido más rápido para poder terminar la recolección con una cantidad limitada de memoria. Los cálculos del peor caso con fragmentación son específicos para cada programa.

3.8.3 Discusión

En la sección anterior suponemos un modelo muy simple de recolección de basura, ya que sólo existe un bloque de memoria disponible para satisfacer todas las solicitudes de memoria. En un sistema sin copiado, con objetos de muy diversos tamaños, éste no sería el caso, ya que al liberar varios objetos pequeños no necesariamente tendríamos la oportunidad de poder asignar a un objeto grande. Por otro lado, en estudios experimentales, se ha encontrado que en la mayoría de las aplicaciones, el uso de la memoria está dominado por unos cuantos objetos; por lo que pensar en recolectores de tiempo real no es tan difícil como parece a primera vista. Sin embargo, este análisis se tiene que hacer para cada aplicación. Para algunos programas, garantizar rendimiento en tiempo real puede ser muy costoso debido a problemas de fragmentación, a menos que los objetos de la aplicación puedan ser subdivididos en objetos de tamaños más uniformes. Otra posibilidad es asignar estáticamente los tipos de datos más problemáticos –como generalmente se hace en sistemas de tiempo real– y permitir que el recolector de basura maneje automáticamente al resto de los objetos.

3.9 ¿Cuál algoritmo incremental escoger?

Al escoger una estrategia incremental, es importante asignar prioridades al rendimiento en el caso promedio y en el peor de los casos. Los algoritmos que son "menos conservadores" pueden no ser más atractivos que los otros, porque en el peor caso se comportan igual de conservadores.

Aún en el caso promedio, los algoritmos menos conservadores resultan ser más lentos (v.gr. porque sus barreras de escritura requieren más instrucciones), y por lo tanto menos atractivos. Paradójicamente, esto hace que los algoritmos *menos conservadores* sean *más conservadores* en la práctica, ya que debido a su costo no son ejecutados muy seguido. Debido a la sobrecarga, el conservadurismo reducido en términos de estrategias incrementales puede introducir un mayor conservadurismo en la frecuencia de la recolección.

Las metas a lograr por el sistema son importantes a la hora de decidir cual algoritmo incremental usaremos. Como explicaremos en el siguiente capítulo, las técnicas generacionales de recolección de basura pueden eliminar la sobrecarga de los esquemas incrementales. Claro que tienen un costo y no son convenientes para sistemas de tiempo real duro. Para otro tipo de sistemas, puede ser deseable combinar técnicas incrementales y generacionales, pero debemos prestar especial atención en cómo combinarlas.

Capítulo 4

Recolección de basura generacional

Dada una cantidad realista de memoria, la eficiencia de la recolección de basura de copiado está limitada por el hecho de que el sistema debe copiar todos los datos vivos durante un ciclo de recolección. En la mayoría de programas en una variedad de lenguajes, *la mayoría de los objetos viven un tiempo muy corto, mientras que un pequeño porcentaje vive mucho más.* Los datos que se han obtenido experimentalmente varían de lenguaje a lenguaje, usualmente entre 80 y 98 por ciento de los objetos asignados al heap mueren en unos cuantos millones de instrucciones de máquina, o antes de que otro mega-byte sea asignado; la mayoría de los objetos mueren incluso más rápido, antes de que unas cuantas decenas de kilo-bytes sean asignadas.

La asignación al heap es utilizada frecuentemente como métrica para la ejecución de programas, en lugar de utilizar un reloj, por dos razones. Una es que es independiente de la máquina y de la velocidad de implementación –varía de manera apropiada con la velocidad a la cual el programa se ejecuta, mientras que un reloj no puede hacerlo; esta métrica evita la necesidad de citar continuamente la velocidad del procesador[1]. También es apropiada para hablar en términos de cantidades asignadas porque el tiempo entre recolecciones de basura está determinado en gran medida por la cantidad de memoria disponible[2].

La *recolección de basura generacional*, evita repetir buena parte del copiado, segregando a los objetos por edad en múltiples áreas, y recolectando las áreas que contienen objetos más viejos menos frecuentemente que las áreas que tienen objetos nuevos. Una vez que los objetos han sobrevivido un cierto número de recolecciones, son movidos a un área recolectada con menos frecuencia. Las áreas que contienen objetos jóvenes son recolectadas con gran frecuencia, porque muchos de esos objetos morirán rápidamente, liberando espacio, y copiar los pocos que sobreviven no resulta muy costoso. Los sobrevivientes son *avanzados* a un estrato de vejez más alto después de cierto número de recolecciones, con la finalidad

[1] Sin embargo debemos ser cuidadosos y no interpretar esta métrica como la medida abstracta ideal. Por ejemplo, los ritmos de asignación al heap para Lisp y Smalltalk son más altos, ya que más información de control, y datos intermedios de cálculos se pasan como referencias a objetos en el heap, en lugar de como estructuras en el stack.

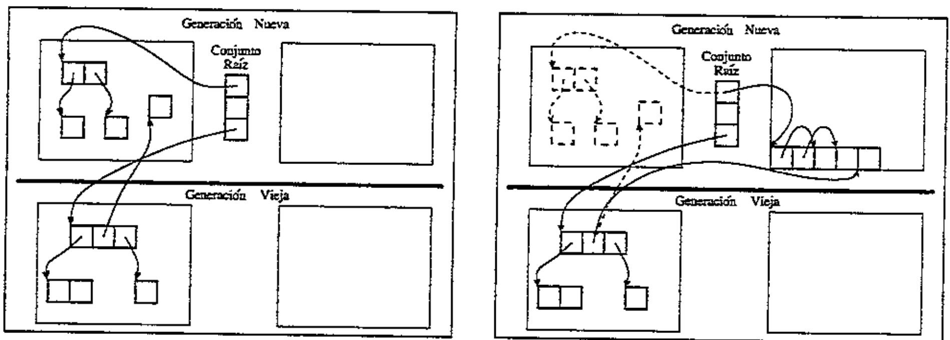
[2] Las métricas relativas a asignación no son la última palabra en medición de la eficiencia de un recolector, ya que detectar un decremento de trabajo por unidad de asignación no es muy importante si el programa no está asignando mucha memoria, inversamente, cambios pequeños en el trabajo realizado para la recolección significan mucho para programas cuyas demandas de memoria son grandes

de mantener los costos bajos.

Para la recolección de detente-y-recolecta (no incremental), la recolección generacional presenta algunos beneficios adicionales, ya que todas las recolecciones toman un tiempo muy pequeño –recolectar sólo la generación más joven es más rápido que hacer una recolección completa. Esto reduce la frecuencia de pausas quebrantadoras y para muchos programas sin plazos muy estrictos de tiempo real, esto es suficiente para obtener un uso interactivo aceptable. La mayoría de las pausas son tan breves (una fracción de segundo) que es poco probable que el usuario las detecte; las pausas largas usadas para hacer recolección multi-generacional pueden ser pospuestas hasta que el sistema no está en uso, o escondidas dentro de fases no interactivas de la operación del programa. Las técnicas generacionales se usan frecuentemente como sustituto de técnicas incrementales más costosas, y para mejorar la eficiencia.

Por razones históricas y por simplicidad de explicación, nos enfocaremos en recolectores generacionales de copiado. Sin embargo, la decisión de recolección de copiado o marcado es ortogonal al problema de recolección generacional.

4.1 Múltiples sub-heaps con frecuencias de recolección distintas



4.1.1: Antes de la recolección

4.1.2: Después de la recolección

Figura 4.1: Un recolector generacional de copiado simple

Consideremos un recolector de basura generacional basado en una organización con semi-espacios: la memoria está dividida en áreas que contendrán objetos de edades muy parecidas, o *generaciones*; la memoria de cada generación está además, divida en semi-espacios. En la Figura 4.1.1 mostramos un esquema generacional con sólo dos grupos de edades, una generación *Nueva* y una generación *Vieja*. Los objetos nuevos son asignados en la generación Nueva, hasta que su semi-espacio actual está lleno. En ese momento la

generación Nueva (únicamente) es recolectada, copiando sus datos vivos al otro semi-espacio, como se muestra en la Figura 4.1.2.

Si un objeto sobrevive lo suficiente como para ser considerado viejo, puede ser copiado de la generación Nueva a la Vieja, en lugar de ser copiado al otro semi-espacio en la generación Nueva. Esto evita que lo consideremos en subsecuentes recolecciones de una sola generación, por lo que ya no será copiado en cada ciclo de recolección. Dado que son relativamente pocos los objetos que viven tanto, la memoria Vieja se llenará más lentamente que la memoria Nueva. Eventualmente, la memoria Vieja se llenará y tendrá que ser recolectada. La Figura 4.2 muestra el patrón generacional de memoria usado en este esquema. Nótese que la figura no está a escala ya que típicamente la generación más joven es varias veces más pequeña que la más vieja.

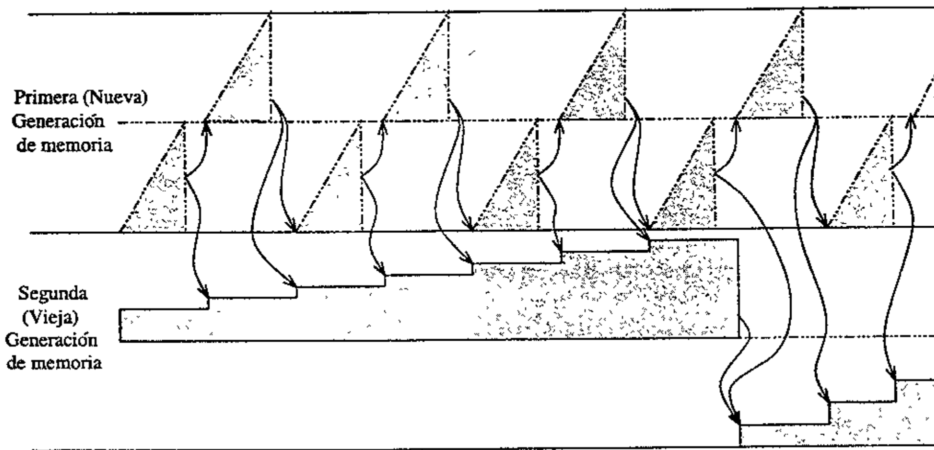


Figura 4.2: Uso de la memoria en un recolector de copia generacional con semi-espacios para cada generación.

El número de generaciones puede ser mayor a dos, donde cada generación sucesiva contiene objetos más viejos y es recolectada con menor frecuencia (Smalltalk Tektronix 4406 utiliza un sistema con ocho generaciones y cada generación tiene dos semi-espacios).

Para que este esquema funcione, debe ser posible recolectar la(s) generación(es) más joven(jóvenes) sin necesidad de recolectar la(s) más vieja(s). Ahora, sabemos que la propiedad de estar vivo es global, por lo tanto, los datos en la memoria de generaciones viejas deben ser tomados en cuenta. Por ejemplo, si hay una referencia proveniente de un objeto en la memoria vieja que apunta a un objeto en la memoria nueva, dicha referencia debe ser localizada en tiempo de recolección y usada como parte del conjunto raíz durante el recorrido. De otra forma, un objeto que está vivo puede ser recuperado por el recolector, o la referencia puede no ser actualizada correctamente cuando se mueva al objeto. En cualquier caso, la integridad y la consistencia de la estructura de datos será destruida en el heap.

Asegurar que el recolector encontrará referencias hacia generaciones más jóvenes requiere del uso de algo como la "barrera de escritura" de un recolector incremental –el programa en ejecución no puede guardar referencias *libremente* en el heap. Cada referencia potencial guardada debe ser acompañada de instrucciones extra para llevar la "administración" y asegurar que si una referencia inter-generacional está siendo creada, entonces el recolector la encontrará. Al igual que en los recolectores incrementales, esto se logra –usualmente– haciendo que el compilador emita una cuantas instrucciones extra junto con cada escritura de una (posible) referencia en un objeto en el heap.

La barrera de escritura puede checar cada escritura, o puede simplemente utilizar *dirty bits* y revisar las áreas donde se almacenan estos *dirty bits* en tiempo de recolección. El punto importante aquí es asegurar que todas las referencias de memoria vieja hacia memoria nueva serán localizadas en tiempo de recolección, y usadas como raíces para el recorrido de copiado.

Si utilizamos estas referencias inter-generacionales como raíces, entonces aseguramos que todos los objetos alcanzables en las generaciones más jóvenes serán alcanzados por el recolector; en el caso de los recolectores de copiado, aseguramos además que todas las referencias a objetos "movidos" se actualizan.

Al igual que en los recolectores incrementales, el uso de una barrera de escritura resulta en una *aproximación conservadora* de la propiedad de estar vivo; todas las referencias de memoria vieja a nueva son usadas como raíces, pero no todas estas raíces están necesariamente vivas. Un objeto en la memoria vieja puede haber muerto, pero ese hecho es desconocido hasta el siguiente ciclo de recolección para la memoria vieja. Por lo tanto, algunos objetos basura pueden ser preservados porque están siendo referidos desde objetos flotantes (basura no detectada). Algunos estudios parecen indicar que –en la práctica– esto no es un problema.

También sería posible seguir el rastro de todas las referencias de objetos nuevos hacia objetos viejos, permitiendo así que objetos viejos fueran recolectados independientemente de los nuevos. Sin embargo esto es más costoso, porque típicamente existen más referencias de objetos nuevos a objetos viejos que al revés. Tal flexibilidad es consecuencia de la forma en que se crean las referencias –creando objetos nuevos que hacen referencias a otros objetos que ya existen. Algunas veces una referencia se instala en un objeto ya existente hacia uno nuevo, pero esto es considerablemente menos común. Este tratamiento asimétrico permite que el uso de código que genera muchos objetos (como la operación cons de Lisp, que se utiliza frecuentemente) no incurra en el gasto extra que representa registrar las referencias intergeneracionales.

Aún sin mantener un registro de estas referencias joven-a-viejo, puede ser posible recolectar una generación sin recolectar a los más jóvenes. En este caso, *todos* los datos en las generaciones más jóvenes son considerados como posibles raíces, y sólo buscamos referencias en dichos datos. Mientras que esta búsqueda es proporcional a la cantidad de datos en las generaciones más jóvenes, cada generación es considerablemente más pequeña que la siguiente, y el costo puede ser relativamente pequeño comparado con el costo de recolectar la generación completa. Además, revisar los datos en las generaciones más jóvenes es preferible que recolectar ambas generaciones, porque revisar es generalmente más rápido que recorrer y copiar; también puede resultar en una mejor localidad.

El costo de registrar referencias intergeneracionales es típicamente proporcional al ritmo de ejecución del programa, i.e. no está particularmente ligado al ritmo de creación de objetos. Para algunos programas, puede ser el costo más alto en que incurre la recolección, porque deben ejecutarse varias instrucciones por cada posible instalación de una referencia en el heap. Esto puede hacer considerablemente más lenta la ejecución del programa.

Dentro del marco de estrategia generacional que hemos presentado, aún tenemos que contestar varias preguntas:

Política de avance. ¿Qué tanto tiempo debe sobrevivir un objeto en una generación antes de que lo avancemos a la siguiente?

Organización del heap. ¿Cómo dividimos y usamos el espacio entre las generaciones? ¿Cómo afecta la localidad al nivel de la memoria virtual el patrón de reuso? ¿Cómo afecta al nivel de la memoria cache de alta velocidad?

Calendarización de recolección. Para un recolector no incremental, ¿cómo podemos mitigar o evitar el efecto de pausas quebrantadoras, especialmente en aplicaciones interactivas? ¿Podemos mejorar la eficiencia utilizando calendarizaciones "oportunistas"? ¿Podemos adaptar este esquema a esquemas incrementales para reducir la cantidad de basura flotante?

Referencias intergeneracionales. Dado que debe ser posible recolectar generaciones más jóvenes sin recolectar a las más viejas, debemos poder encontrar las referencias vivas en generaciones más viejas que apunten a las que estamos recolectando. ¿Cuál es la mejor forma de hacer esto?

En las siguientes secciones trataremos de responder a estas preguntas.

4.2 Política de avance

La política más sencilla de avance es simplemente avanzar los datos vivos a la siguiente generación en cuanto son recorridos Esto tiene la ventaja de que es sencillo de implementar, ya que no es necesario distinguir entre objetos de distintas edades dentro de una generación En un recolector de copiado, esto permite el uso de una sola área contigua de espacio para la generación, sin división en semi-espacios, y no requiere de ningún tipo de campos en encabezados para mantener información de edades.

Una desventaja es que impide la acumulación de objetos de larga vida dentro de una generación –un objeto de larga vida no puede ser copiado a la misma escala de tiempo, ya que será rápidamente avanzado a la siguiente generación, que es recolectada con menos frecuencia.

El problema aquí es que los objetos avanzan *muy rápido* –los objetos recientes que sean asignados a memoria poco antes de la recolección serán avanzados a la siguiente generación, a pesar de ser muy jóvenes, y tal vez mueran inmediatamente. Esto causará que las generaciones más viejas se llenen muy rápido y requieran ser recolectadas más seguido El problema de los objetos de vida corta puede ser aminorado retrasando el avance de objetos

por sólo un ciclo de recolección; esto asegura que los objetos son aproximadamente de la misma edad (dentro de un factor de dos) cuando son avanzados a la siguiente generación.

No es claro si mantener los objetos dentro de una generación por más de dos ciclos de recolección vale el costo extra que implica el copiado. Bajo casi cualquier condición, parece que copias sucesivas no reducen la cantidad de datos que debemos avanzar, aunque esto es dependiente de la naturaleza de la aplicación; también puede ser deseable variar la política de avance dinámicamente.

La decisión de mantener los datos dentro de una generación por un cierto número de ciclos de recolección se ve afectada por el número y tamaño de las generaciones. En general, si hay muy pocas generaciones (v.gr. dos, como en el sistema de Ungar, Generation Scavenging), es más deseable mantener los datos dentro de la primera generación, para evitar que la segunda se llene muy rápido y tenga que ser recolectada. Sin embargo, si existen generaciones intermedias, es usualmente preferible avanzar los datos más rápido, ya que muy probablemente morirán en alguna generación intermedia y nunca llegarán a la generación más vieja.

4.3 Organización del heap

Un recolector generacional debe tratar a objetos de distintas edades de manera distinta. Mientras revisa, debe ser capaz de decir a qué generación un objeto pertenece, para poder decidir si recorrer a sus descendientes o no, y decidir también si avanzarlo o no a otra generación. La barrera de escritura debe ser capaz de determinar la generación a que pertenece un objeto, para detectar si una referencia a un objeto más joven está siendo guardada en el objeto más viejo.

En un recolector de copiado, esto se logra usualmente manteniendo a los objetos de distintas edades en distintas áreas de memoria. En muchos sistemas, estas son áreas contiguas de memoria, y la generación de un objeto puede ser determinada simplemente comparando direcciones. En otros sistemas, las "áreas" pueden ser conjuntos contiguos de páginas –la generación de un objeto puede ser determinada usando la parte que representa la página de su dirección de memoria para indexarlo en una tabla que dice a qué generación pertenece esa página.

En otros sistemas, tales como recolectores sin copiado, cada objeto pertenece a una generación, pero objetos de distintas generaciones pueden estar mezclados en la memoria. Típicamente, cada objeto tiene un campo en un encabezado que indica a qué generación pertenece.

4.3.1 Sub-áreas en esquemas de copiado

Los recolectores generacionales de copiado dividen el espacio de cada generación en varias áreas. Por ejemplo, cada generación puede consistir de un par de semi-espacios, para que los objetos puedan ser copiados de un semi-espacio a otro, y así retenerlos en una generación a través de varias recolecciones. Si sólo se utiliza un espacio, los objetos deben ser avanzados a otras generaciones inmediatamente porque no hay lugar en dónde copiarlos dentro de la misma generación.

La localidad en el uso de le memoria de los semi-espacios es muy pobre –sólo la mitad de la memoria de una generación dada puede estar en uso en cualquier momento, pero ambos semi-espacios son tocados completamente cada dos ciclos de recolección. En las máquinas Lisp, los recolectores de basura evitan este problema utilizando un sólo espacio por generación. En lugar de copiar objetos de un semi-espacio a otro hasta que son avanzados, la recolección de basura de una generación avanza a *todos* los objetos sobrevivientes a la siguiente generación. Esto evita la necesidad de tener dos semi-espacios, excepto en la última generación, que no tiene otra generación a la cual copiar sus objetos. Desafortunadamente, esto tiene la desventaja de que objetos jóvenes pueden ser avanzados junto con objetos relativamente viejos –los objetos que son asignados poco antes de una recolección no tienen tiempo suficiente para morir antes de ser asignados. Estos objetos relativamente jóvenes tienen una alta probabilidad de morir poco después de ser avanzados, ocupando así, innecesariamente, espacio en la siguiente generación y forzando una recolección prematura.

La solución de Ungar a este problema en la generación más joven (de su recolector Generation Scavenging) es usar *tres* espacios en lugar de dos, con todos los objetos asignados originalmente en el tercer espacio. Los objetos recién creados en este tercer espacio son copiados a otro espacio, junto con los objetos del otro espacio. El tercer espacio es vaciado en cada ciclo de recolección, y puede ser reutilizado inmediatamente. Por lo tanto, tiene características de localidad similares a las de un sistema con un solo espacio por generación. A primera vista, parece ser que agregar un tercer espacio por generación aumentará considerablemente nuestro requerimiento de memoria, dado que aún requerimos semiespacios en esa generación para poder mantener a los objetos por varios ciclos de recolección en esa generación. El área de creación es utilizada en su totalidad en cada ciclo de asignación y recolección, mientras que los otros dos espacios son usados para mantener a los sobrevivientes de cada ciclo de recolección. Típicamente, sólo una pequeña minoría de objetos nuevos sobrevive, incluso a la primera recolección, por lo que sólo una pequeña parte de cada espacio es usado la mayor parte del tiempo y el uso promedio de memoria es bajo.

El recolector de basura oportunista de Wilson [Wilson and Moher, 1989b], utiliza una variación de este esquema, con las sub-áreas dentro de cada generación usadas con el propósito adicional de decidir cuando avanzar un objeto de una generación a la siguiente –los objetos son avanzados de los semi-espacios a la siguiente generación en cada ciclo, en lugar de ser copiados de un semi-espacio a otro una y otra vez. En efecto, esto es un mecanismo de avance por "trincheras", donde los objetos se van segregando en sub-áreas para codificar sus edades y utilizar esta información como política de avance. Nos ahorra la necesidad de usar campos de edad en encabezados, lo cual es una ventaja para muchos sistemas. Esto provee una garantía de que los objetos no serán avanzados fuera de la generación sin haber sobrevivido al menos una recolección (y a lo más dos), lo cual es suficiente para evitar avance prematuro de objetos de vida corta.

En varios sistemas de recolección generacional de copiado, la generación más vieja es tratada de manera especial. En los recolectores de las máquinas Lisp, esto es necesario ya que la mayoría de las generaciones son vaciadas en cada ciclo de recolección –para la generación más vieja no hay otra (más vieja) a la cual copiar los objetos. La última generación ("espacio dinámico") es estructurada como un par de semi-espacios, que se usan alternadamente. Una mejora extra es proveer un área especial, conocida como "espacio es-

tático", donde no se lleva a cabo recolección de basura durante la operación normal. Esta área mantiene información del sistema y código compilado que esperamos no cambie.

Algunos recolectores de copiado basados en el sistema de Ungar, tratan la generación más vieja estructurándola como un solo espacio de memoria y utilizan un sistema de marcado y compactación. En estos sistemas, todas las generaciones residen en RAM durante la ejecución normal, y el uso de un solo espacio reduce la cantidad de memoria RAM requerida para mantener toda la última generación en memoria principal. El algoritmo de marcado y compactación es más caro que un recolector de copiado típico; sin embargo, la habilidad de realizar una recolección completa sin necesidad de paginar vale el pago extra. Pueden ser usadas técnicas sin copiado para el mismo propósito, pero están sujetas a problemas de fragmentación.

4.3.2 Generaciones en sistemas sin copiado

En nuestra discusión de recolección generacional, nos hemos enfocado principalmente en esquemas de recolección de basura de copiado, donde las generaciones pueden ser vistas como "áreas" de memoria que mantienen objetos de distintas edades. Esto es innecesario, siempre y cuando sea posible distinguir objetos de diferentes edades y tratarlos de manera distinta. De la misma manera que los algoritmos de recolección incremental son mejor entendidos en términos de la abstracción del marcado tricolor, los algoritmos generacionales son mejor entendidos en términos de conjuntos de objetos que son recolectados a distintas frecuencias. Cada uno de estos conjuntos de edades se puede dividir en conjuntos sombreados y no sombreados para propósitos del recorrido.

El recolector de basura Xerox PARC PCR (Portable Common Runtime) es generacional y utiliza marcado y barrido, con un encabezado por objeto que indica su edad. Objetos de distintas edades pueden ser asignados a la misma página, aunque el sistema utiliza una heurística para minimizar esta situación, por razones de localidad[3]. Cuando se recolecta sólo objetos jóvenes, el recolector PCR revisa el conjunto raíz y recorre aquellos objetos cuyos encabezados cazan con la "edad de recolección". La barrera de escritura generacional utiliza *dirty* bits, que son mantenidos por el sistema con técnicas de protección de acceso como los que se usan en un sistema de memoria virtual común y corriente.

4.3.3 Discusión

Muchas variantes de recolección generacional son posibles, y son comunes esquemas híbridos, lo que permite ajustar detalles de espacio-eficiencia.

Es común que los recolectores de copiado manejen objetos grandes de manera distinta, guardándolos en un área especial para *objetos grandes*, lo que ahorra el tener que copiarlos. Esto combina el copiado de objetos pequeños (que es barato) con marcado y barrido para objetos grandes, lo que elimina la sobrecarga en espacio y tiempo que representa copiar objetos grandes.

[3]Las páginas que contienen objetos viejos no son usadas para mantener objetos nuevos, excepto en el caso en que menos de la mitad del espacio en la página esté libre; esto evita mezclar gratuitamente datos nuevos y viejos.

El recolector de basura de ParcPlace Smalltalk-80 combina recolección de detente-y-copia para la generación más joven (donde la pausa, incluso en el peor caso, no es muy grande) con recolección incremental de marcado y barrido de datos más viejos.

4.4 Seguimiento de referencias intergeneracionales

Los recolectores generacionales deben detectar referencias de generaciones más viejas a generaciones jóvenes, por lo que requieren de *barreras de lectura* similares a las usadas en los algoritmos de recorrido incremental. Esto es, un programa no puede simplemente guardar referencias en objetos en el heap –el compilador o el hardware deben asegurarse de que cada operación de escritura va acompañada de operaciones de grabado o revisión, para asegurar que si una referencia a una generación más joven ha sido creada, el recolector la podrá encontrar. Típicamente, el compilador emite instrucciones extra junto con cada instrucción potencial de escritura, para realizar las operaciones requeridas por la barrera de escritura.

Para muchos sistemas, éste puede ser el costo más grande de la recolección generacional. Por ejemplo, en un sistema Lisp moderno con un compilador con optimización, el almacenamiento de referencias es aproximadamente el uno por ciento de toda la cuenta total de instrucciones. Si cada referencia requiere veinte instrucciones para la barrera de escritura, el rendimiento se verá degradado en un veinte por ciento. Por lo tanto, optimizar la barrera de lectura es muy importante para el desempeño promedio del recolector de basura, y han sido creadas algunas barreras de escritura significativamente más rápidas. Las discutiremos con más detalle.

Se han usado muchas técnicas de barrera de escritura, con distintos rendimientos en hardware distinto, y para distintos lenguajes y estrategias de implementación de lenguajes.

Algunos sistemas utilizan una estrategia de recolección que es "casi generacional" pero sin usar una barrera de escritura, para obtener algunos de los beneficios de la recolección generacional. En lugar de seguir la pista de las referencias de objetos viejos a jóvenes conforme son creados, los datos en las generaciones viejas son explorados en busca de tales referencias. Esto requiere más trabajo de exploración, pero puede ser considerablemente más rápido que recorrer todos los datos alcanzables. Explorar es varias veces más rápido que recorrer, y tiene una localidad espacial de referencia muy fuerte.

4.4.1 Tablas de indirección

El recolector generacional original para las máquinas Lisp, utilizaba hardware especializado o microcódigo para acelerar el chequeo de referencias hacia generaciones más jóvenes, y las referencias que eran encontradas eran direccionadas (por una rutina en microcódigo) hacia una *entrada en la tabla.* No se permitían referencias directamente a objetos en generaciones más jóvenes, solamente referencias a una entrada en la tabla que contenía la referencia al objeto. Cada generación tenía su propia tabla, donde se almacenaban referencias a todos los objetos en la generación. Cuando el mutante ejecutaba una instrucción de almacenamiento, e intentaba crear una referencia hacia una generación más joven, la instrucción de almacenamiento era atrapada en microcódigo. Así, en lugar de crear una re-

ferencia directamente al objeto, un *apuntador de avance invisible* se guardaba. La máquina Lisp y el microcódigo detectaban y hacían la referencias a los apuntadores de avance automáticamente, por lo que tales indirecciones eran invisibles para el programa en ejecución.

Cuando se recolectaba una generación particular, solo era necesario usar la tabla de entradas asociada como un conjunto adicional de raíces, en lugar de encontrar las referencias hacia otras generaciones y actualizarlas.

Variantes de este esquema son posibles; sin embargo, todas ellas no son lo suficientemente rápidas, especialmente en hardware común sin soporte para hacer transparentes las referencias a los apuntadores de avance. Al igual que el sistema de copiado incremental de Baker, el costo de las operaciones comunes con referencias aumenta de manera importante si cada referencia debe ser revisada por si existe alguna indirección. Los recolectores generacionales recientes evitan las indirecciones, y permiten referencias de una generación a cualquier otra. Y en lugar de rastrear tales referencias, simplemente mantienen sus posiciones, para que puedan ser localizadas en tiempo de recolección. Nos referimos a tales esquemas como *esquemas con grabado de referencias*, ya que simplemente graban la localización de las referencias.

4.4.2 Conjuntos memoria de Ungar

El recolector *Generation Scavenging* de Ungar utiliza un esquema de grabado de referencias inteligente con respecto a los objetos: graba qué objetos tienen referencias a generaciones más jóvenes. Cada almacenamiento potencial de una referencia, activa la barrera de escritura, que checa si una referencia intergeneracional está siendo creada –verificando si el valor almacenado es una referencia, que apunta hacia la generación joven y constatando si está siendo guardada en un objeto en la generación vieja. Si es así, el objeto en el cual se graba la referencia es añadido al *conjunto memoria* de objetos que tienen tales referencias, si es que aún no está en el conjunto. Cada objeto tiene un bit en su encabezado que dice si el objeto está o no en el conjunto memoria, para evitar tener duplicados. Esto hace que el costo de la exploración en tiempo de recolección dependa del número y tamaño de los objetos almacenados, y no en el número de operaciones de almacenamiento.

En el caso promedio, este esquema trabaja bastante bien para la máquina virtual de Smalltalk. Desafortunadamente, en el peor caso, este chequeo incurre en un costo de decenas de operaciones por cada operación de almacenamiento, lo cual resulta muy costoso para una implementación de alto rendimiento.

Un gran problema de este esquema es que el conjunto memoria de objetos debe ser recorrido en su totalidad en la siguiente recolección de basura, lo que puede ser muy costoso por dos razones:

1. Algunos de los costos de chequeo son repetidos, ya que una localidad puede ser ocupada para almacenar varias veces entre recolecciones, siendo revisada cada vez y porque los objetos almacenados tienen que ser revisados en su totalidad en tiempo de recolección.

2. Peor aún, pueden grabarse objetos muy grandes con regularidad, y tienen que ser revisados durante la recolección. Esto ocasiona una gran cantidad de trabajo de exploración. Incluso en la implementación de Tektronix Smalltalk, provocaba una caída del sistema.

4.4.3 Marcado de páginas

El recolector de basura efímero de *Moon* para las máquinas Lisp (*Symbolics*) utiliza un esquema distinto de grabado de referencias. En lugar de grabar qué objetos tienen referencias intergeneracionales almacenadas en ellos, graba en qué páginas de memoria virtual fueron almacenadas. El uso de páginas como granularidad de grabado evita el problema de revisar objetos muy grandes, pero incrementa el costo para objetos pequeños, ya que de cualquier forma revisa toda la página. El costo de explorar no es muy grande en el hardware de Symbolics, ya que tiene un soporte especial con etiquetas que permite hacer el chequeo de la generación muy rápido, y porque además las páginas son muy pequeñas. Gran parte de la barrera de escritura está implementada directamente en hardware (en lugar de agregar instrucciones adicionales a cada escritura de una referencia), por lo que el costo de cada almacenamiento de una referencia es pequeño. En este sistema, la información sobre referencias hacia generaciones más jóvenes se mantiene en una tabla relacionada con las páginas. Esto tiene la ventaja de que la tabla, implícitamente, elimina los duplicados –una página puede ser guardada cualquier número de veces, pero la página solo será revisada una sola vez durante la siguiente recolección. Esta eliminación de duplicados es equivalente al uso de los bits en encabezados de Ungar, para asegurar la unicidad de entradas en el conjunto memoria. El tiempo requerido para recorrer los objetos grabados durante la recolección, es proporcional al número de páginas donde se guardaron referencias y al tamaño de las páginas, pero no al número de operaciones de almacenamiento realizadas.

Desafortunadamente, este esquema será considerablemente más lento si se implementa en hardware común, con páginas más grandes y sin hardware especializado para checar escrituras de referencias o explorar páginas.

Recientemente, los *dirty bits* de memoria virtual han sido usados como una forma de barrera de escritura. La mayor parte del trabajo hecho para mantener estos bits se hace en hardware de memoria dedicado, así que desde el punto de vista del implementador, es gratis. Desafortunadamente, la mayoría de los sistemas operativos no ofrecen ninguna facilidad para examinar los dirty bits, por lo que requerimos modificar el kernel del sistema operativo. Alternativamente, herramientas de protección de memoria se pueden usar para simular dirty bits, poniendo una protección contra escritura en las páginas para que cualquier escritura sea detectada por el hardware y se invoque a un manejador; esta técnica se usa en el recolector *Xerox Portable Common Runtime, PCR* [Weiser et al., 1989]. El manejador simplemente graba que la página ha sido escrita desde la última recolección, y quita la protección a la página para que el programa pueda continuar. En el recolector PCR, objetos de distintas generaciones pueden residir en la misma página, por lo que cuando una página está sucia y es revisada en la recolección, los objetos de generaciones irrelevante se omiten. El uso de protecciones de memoria virtual imposibilita que este esquema pueda satisfacer requerimientos duros de tiempo real. Sin embargo, como explicaremos más adelante, este tipo de barrera de escritura tiene varias ventajas cuando tratamos con compiladores que no cooperan y que no emiten instrucciones de barrera de escritura.

4.4.4 Marcado de palabras

Al adaptar el recolector de Moon a hardware normal, Sobalvarro[Sobalvarro, 1988] evitó el costo de explorar páginas muy grandes usando un sistema de *marcado de palabras*, que usa un mapa de bits para grabar qué palabras de memoria tienen referencias almacenadas en ellas. Esto evita la necesidad de revisar una página arbitraria para encontrar referencias, ya que solo las localidades de las referencias relevantes fueron almacenadas.

Sobalvarro también optimizó este esquema para hardware normal usando una barrera de escritura más simple –casi todo el chequeo de la barrera de escritura fue eliminado, y pospuesto hasta tiempo de recolección. Las localidades en donde se guardan referencias se checan en tiempo de recolección para ver si son referencias intergeneracionales o no. Mientras que esto es menos preciso que la verificación que hacen Ungar o Moon, y puede causar que se revisen más palabras en tiempo de recolección, tiene el beneficio de que la eliminación de duplicados se hace primero, y los demás chequeos solo se hacen una vez por cada referencia almacenada.

El inconveniente del esquema de Sobalvarro es que para un heap razonablemente grande, la tabla de bits es muy grande, cerca del treinta por ciento de la memoria total. Explorar tal tabla puede ser relativamente costoso si está representada como un arreglo lineal de bits. La solución de Sobalvarro fue utilizar una representación dispersa (en dos niveles) de la tabla; esto incurre en un costo adicional de la barrera de escritura, ya que las operaciones en arreglos dispersos son significativamente más lentas que las operaciones en arreglos contiguos.

4.4.5 Marcado de cartas

Una alternativa más a usar páginas o palabras es dividir conceptualmente la memoria en unidades de tamaño intermedio llamadas *cartas*. El uso de cartas relativamente pequeñas tiene la ventaja de que una sola operación de almacenamiento, provoca solo una pequeña cantidad de trabajo de exploración en tiempo de recolección, en promedio. Mientras que las cartas no sean excesivamente pequeñas, la tabla usada para almacenar las cartas que contienen referencias, es mucho más pequeña que la tabla usada para mantener el mapa de bits en el marcado de palabras. Para la mayoría de los sistemas, esto hace posible representar la tabla como un arreglo lineal contiguo, manteniendo así la barrera de escritura rápida.

Un problema de utilizar marcado de cartas en hardware convencional es que requiere revisión de las cartas para buscar referencias. Y el problema se presenta cuando un objeto es muy grande y no cabe en una sola carta, ya que el recolector revisará cada carta, aún cuando alguna(s) de ellas no empiecen con el inicio de un objeto. El recolector oportunista, propuesto por Wilson [Wilson and Moher, 1989a], resuelve este problema manteniendo un *mapa de cruce*, en el cual lista las cartas que comienzan con una parte *imposible de revisar* de un objeto. Cuando una carta no puede ser recorrida desde el inicio, el mapa se usa para encontrar una carta anterior que sí pueda ser revisada, localiza el encabezado de un objeto en esa carta y comienza a saltar de objeto en objeto, hasta que encuentra los encabezados de los objetos en la carta que deseábamos revisar originalmente. En el esquema de Wilson, el bit que corresponde a la carta se deja activado si la carta contiene una referen-

cia hacia una generación más joven. Tales cartas deben ser recorridas nuevamente al inicio de la siguiente recolección de basura, aún si no hubo ninguna operación de almacenamiento en ella.

4.4.6 Listas de almacenamiento

El enfoque más sencillo para grabar referencias es, simplemente, guardar cada dirección en la que se almacena una referencia en una lista. Esta puede ser una lista ligada, o un arreglo.

Andrew W. Appel [Appel, 1989] utilizó este esquema para crear su recolector generacional para *Standard ML of New Jersey*. Éste es un programa de tan solo 500 líneas escrito en C. El funcionamiento de su recolector es muy sencillo, en cada ciclo de recolección revisa la lista y actualiza las direcciones de memoria almacenadas en ella.

Las listas de almacenamiento sencillas tienen una desventaja para muchos lenguajes de programación, ya que son implementadas como *bolsas* o *multiconjuntos* de localidades, y no como conjuntos. Esto es, la misma locaclidad puede aparecer en la lista muchas veces si es frecuentemente utilizada para almacenar referencias (por el mutante), por lo que el recolector de basura debe revisar cada una de esas entradas en cada ciclo de recolección. El costo en tiempo de la recolección es, por lo tanto, proporcional al número de referencias almacenadas, en lugar de al número de localidades en las que se almacenó algo. La imposibilidad de poder eliminar duplicados también puede conducir a un uso excesivo de espacio si el almacenamiento de referencias es muy frecuente (para ML esto no es un problema, ya que los efectos secundarios son usados relativamente con poca frecuencia; al igual que para la mayoría de los lenguajes funcionales puros)

Se han creado variantes de las listas de almacenamiento; por ejemplo, Moss propuso una técnica que permite que un número limitado de entradas sean almacenadas en una clase especial de lista, llamada *buffer de almacenamiento estático*, y llama a una rutina especial cuando este buffer se llena La rutina especial procesa la lista, utilizando una tabla de hash muy rápida para eliminar los duplicados. Esta técnica, reduce los costos de espacio, y elimina la necesidad de todo el procesamiento de la lista en tiempo de recolección, pero no tiene la misma ventaja de eliminar los duplicados que tienen las tablas –los duplicados son eliminados, pero solo después de que han sido puestos en la lista; cuando la rutina especial los ha movido a la tabla de hash.

4.4.7 ¿Qué estrategia de barrera de escritura escoger?

Al escoger una estrategia de barrera de escritura para un recolector generacional, es importante tomar en cuenta la interacción con otros aspectos de la implementación del sistema. Por ejemplo, el recolector Xerox PARC utiliza técnicas de memoria virtual (para implementar los dirty bits con conocimiento de páginas) en parte, porque está diseñado para trabajar con una variedad de compiladores que pueden no cooperar en la implementación de la barrera de escritura –V.gr compiladores de C (sin modificaciones) no emiten instrucciones de barrera de escritura junto con cada almacenamiento de una referencia. En otros sistemas, especialmente los que utilizan esquemas de tipos estáticos o con capacidades de inferencia

de tipos, el compilador puede reducir significativamente el costo de la barrera de escritura, omitiendo los chequeos de la barrera que pueden ser hechos estáticamente por el compilador.

Otro punto importante, es saber si se requiere una respuesta de tiempo real. Los esquemas basados en tablas, tales como marcado de páginas o cartas, pueden dificultar la labor de revisar las referencias almacenadas de manera incremental. Las listas de almacenamiento son más fáciles de procesar en tiempo real; de hecho, el trabajo realizado por la barrera de escritura puede ser similar al trabajo hecho por una técnica de recorrido incremental, permitiendo que algunos costos sean optimizados, combinando las dos barreras de escritura.

El costo mismo de las barreras de escritura es controversial. Muchos estudios han medido la sobrecarga que representan las barreras de escritura para sistemas interpretados, pero muy pocos se han hecho para sistemas con compiladores de alto rendimiento; lo cual dificulta cualquier comparación deseable. Lo más razonable es combinar medidas de sistemas de alto rendimiento con un entendimiento analítico del costo de un recolector de basura, para inferir cual es el costo aproximado de un recolector bien implementado para un sistema bien implementado.

Como mencionamos anteriormente, los sistemas Lisp ejecutan aproximadamente una instrucción de almacenamiento de referencia en un objeto en el heap por cada cien instrucciones; un barrera de escritura con marcado de cartas debe reducir el rendimiento en aproximadamente cuatro o cinco por ciento, ejecutando dos o tres instrucciones extra por cada referencia almacenada, más un pequeño costo del recorrido de cartas en tiempo de recolección. Para muchos programas (con pocos datos vivos, o distribuciones de vida favorables a la recolección generacional), los costos de recorrido y recuperación serán similarmente pequeños, y el costo de la recolección de basura estará por debajo del diez por ciento.

Sin embargo, esto puede variar considerablemente, dependiendo de la carga de trabajo, información de tipos en el lenguaje de programación, representación de datos, y optimizaciones realizadas por el compilador. Algunas selecciones de implementación serán revisadas en secciones posteriores.

Si un compilador genera código rápido, el costo de la barrera de escritura se puede convertir en una fracción grande del tiempo (pequeño) promedio de ejecución. Por otro lado, el compilador puede también reducir los costos de la barrera de escritura infiriendo que algunas referencias almacenadas son redundantes, o que algunos valores dinámicamente tipificados nunca serán referencias.

Desafortunadamente, el costo de las barreras de escritura en sistemas convencionales imperativos y estáticamente tipificados es pobremente entendido. Los sistemas estáticamente tipificados generalmente distinguen tipos referenciados y no referenciados, lo cual puede ayudar al compilador, pero las declaraciones de tipos pueden mejorar otras áreas del rendimiento del sistema aún más, haciendo el rendimiento relativo del recolector aún peor. Por otro lado, los sistemas convencionales estática y fuertemente tipificados tienen razones promedio muy bajas de asignación y mutación de objetos en el heap, lo que reduce tanto la barrera de escritura como los costos de recorrido a una pequeña fracción del tiempo de ejecución total.

El estilo de programación también puede tener un impacto significativo en los costos

de la barrera de escritura. En muchos lenguajes diseñados para usarse con un recolector de basura, las rutinas de asignación son primitivas que toman valores como argumentos, e inicializan los encabezados de los objetos asignados. En otros lenguajes, se espera que el programador inicialice los campos de un nuevo objeto directamente. En los primeros, la implementación del lenguaje puede omitir la barrera de escritura para escrituras de inicialización de objetos, pero en los segundos, generalmente no puede.

4.5 Revisión del principio generacional

Los recolectores de basura generacionales explotan el hecho de que los objetos asignados al heap viven poco tiempo. Los objetos de larga vida son recolectados menos frecuentemente, bajo la suposición de que la minoría de objetos que viven por un periodo significativo de tiempo, vivirán aún más. Esta idea está muy difundida, pero no es obvio que sea verdadera en el sentido estricto; es incierto también el hecho de que la recolección generacional valga la pena en la práctica.

Consideremos un sistema en el cual lo anterior no se cumpla –i.e. la probabilidad de que un objeto muera en un momento particular no está relacionada con su edad. Las distribuciones de vida en tal sistemas pueden estar sesgadas hacia los objetos de vida corta. Un ejemplo sencillo de tal sistema es uno con la propiedad de *decaimiento exponencial*, en donde una fracción fija de los objetos morirá en un periodo fijo de tiempo, tal y como la propiedad de "media-vida" de los isótopos radioactivos.

En tal sistema, la distribución de vida parece ideal para aplicar recolección generacional, porque los objetos jóvenes mueren jóvenes. Sin embargo, examinando la situación con más detalle, si escogemos *cualquier* subconjunto de los objetos encontraremos iguales proporciones de objetos vivos y muertos sobre un periodo dado de tiempo. En ese caso, cualquier ventaja de la recolección generacional estaría dada por la restricción del alcance de la recolección y no por una mayor tasa de mortalidad entre los objetos sujetos a recolección.

Esta analogía parece *mostrar* que la noción de recolección generacional no es una buena idea. Sin embargo, aún en un modelo con decaimiento exponencial, la recolección generacional puede mejorar la *localidad*, a pesar del hecho de que no mejorará la eficiencia —recuperar y reutilizar espacios usados recientemente mejora la localidad, comparado con reutilizar memoria que ha estado inutilizada por mucho tiempo. Además, recorrer los objetos vivos es probablemente más barato si los objetos fueron asignados —por lo tanto tocados— recientemente.

4.6 Trampas de la recolección generacional

La recolección generacional intenta mejorar el rendimiento heurísticamente, tomando ventaja de las características de los programas típicos; naturalmente, esto no puede ser exitoso para todos los programas. Para algunos programas la recolección generacional fallará en tratar de mejorar el rendimiento y lo puede decrementar.

4.6.1 El problema del "cerdo en la serpiente"

Un tipo de datos problemáticos para la recolección generacional es un racimo de objetos de larga vida, que son creados aproximadamente al mismo tiempo y persisten por un periodo significativo de tiempo. Esto ocurre frecuentemente, ya que las estructuras de datos son creadas durante una fase de la ejecución del programa y son recorridas y utilizadas en fases subsecuentes de la ejecución, y se convierten en basura de un solo golpe (v.gr. cuando la raíz de un gran árbol se convierte en basura). Este tipo de estructuras de datos será copiada repetidamente, hasta que la política de avance logre avanzarla hasta una generación lo suficientemente grande para mantener el racimo completo de datos relacionados. Esto aumenta el costo de recorrido primero en la generación más joven, luego en la siguiente generación, y así sucesivamente, como el bulto en la serpiente que avanza a lo largo del cuerpo de la serpiente después de comer. Hasta que el bulto avanza hasta una generación en la cual se mantendrá hasta que muera, la heurística de edad del recolector fallará y causará trabajo extra de recorrido.

El problema del cerdo en la serpiente favorece el uso de una política de avance rápido de una generación a la siguiente, lo cual debe estar balanceado con la desventaja que representa avanzar muchos datos y forzar que la siguiente generación sea recolectada más seguido de lo necesario. Ungar y Jackson varían la política de avance dinámicamente en un intento de avanzar los racimos grandes de datos fuera de la primera generación antes de que incurran en un costo de copiado muy alto; eso parece funcionar bien para la mayoría de programas, pero puede provocar que la generación más vieja se llene rápidamente. Wilson utiliza más generaciones para aliviar este problema, junto con temporización oportunista de la recolección de basura, en un intento por recolectar cuando sólo pocos datos vivos persisten. El oportunismo "objeto clave" de Hayes refina este esquema usando cambios en el conjunto raíz para influenciar la política de recolección de basura. Todas estas técnicas parecen ser benéficas pero se requiere más tiempo de experimentación en sistemas grandes.

4.6.2 Objetos pequeños alojados en el heap

Una de las suposiciones detrás de la heurística generacional es que habrá pocas referencias de objetos viejos a nuevos; sin embargo, algunos programas pueden violar esta suposición. Un ejemplo son los programas que usan grandes arreglos de referencias a pequeños números de punto flotante alojados en el heap. En muchos sistemas dinámicamente tipificados, los números de punto flotante no caben en una palabra de la máquina, y en el caso general deben ser representados como referencias etiquetadas a objetos asignados al heap. Actualizar los valores en un arreglo de números de punto flotante puede ocasionar que nuevos objetos de punto flotante sean asignados, y referencias a ellos agregadas al arreglo. Si el arreglo es grande y vive un periodo considerable de tiempo, es probable que resida en una generación vieja, y cada actualización del valor de un número de punto flotante creará un nuevo objeto y una referencia intergeneracional. Si un gran número de elementos en el arreglo son modificados (v.gr. por una pasada secuencial a lo largo de todo el arreglo), cada objeto (número de punto flotante) puede vivir un periodo largo de tiempo –lo suficientemente grande como para ser recorrido por varias recolecciones en las generaciones más jóvenes, y avanzado a generaciones más viejas. Por lo tanto, estos grandes números de

objetos numéricos de vida intermedia provocarán una sobrecarga considerable, tanto en la barrera de escritura como en el recorrido.

Para aliviar estos costos, hay dos enfoques comunes –usar formatos pequeños de punto flotante, que pueden ser representados como valores inmediatos con una etiqueta, y usar arreglos con campos de tipo, que pueden contener valores de punto flotante en lugar de referencias a objetos alojados en el heap.

El problema con los valores pequeños de punto flotante es que tal vez no tengan las características matemáticas deseables. En primer lugar, los números de punto flotante lo suficientemente pequeños para caber en una palabra de máquina, pueden no tener la precisión necesaria para algunas aplicaciones. Además, algunos bits deben ser sacrificados para la etiqueta, reduciendo aún más la precisión o el rango de los números. El esquema más simple es sacrificar los bits de precisión eliminando bits de la mantisa, pero esto tiene el problema de que los números resultantes no se pueden mapear bien a hardaware típico de punto flotante, que tiene características muy cuidadosamente diseñadas para mantener cierta precisión y redondeo. Sin las propiedades esperadas de redondeo, algunos algoritmos pueden calcular respuestas incorrectas, o incluso fallar para determinar una respuesta. Simular estas características en software es extremadamente costoso. La alternativa es sacrificar bits de la parte del exponente del formato soportado por el hardware, y restringir el rango de números que puede ser representado. Esto introduce una sobrecarga considerable, ya que se tienen que ejecutar muchas instrucciones extra para convertir entre el formato soportado por el hardware y el formato etiquetado.

Usar arreglos con campos de tipo introduce irregularidades en los sistemas dinámicamente tipificados (v.gr. la mayoría de los arreglos pueden contener cualquier tipo de datos, pero alguno no pueden), pero esta estrategia es fácil de implementar eficientemente, y usada muy frecuentemente en sistemas Lisp.

Desafortunadamente, ninguna de estas soluciones arregla el problema en el caso general, ya que los números de punto flotante no son lo únicos datos que pueden provocar este problema. Considere los números complejos, objetos que representan puntos en un espacio tridimensional, etc., es muy poco probable que tales datos sean mapeados a una palabra de máquina. Similarmente, el problema no solo ocurre con los arreglos –cualquier estructura de datos de agregación (tal como un árbol binario) puede exhibir el mismo problema. En tales casos, el programador puede escoger usar una representación distinta (v.gr. arreglos paralelos de componentes reales e imaginarias, en lugar de un arreglo de números complejos) para evitar la sobrecarga innecesaria de la recolección de basura.

4.6.3 Conjuntos raíz muy grandes

Otro problema potencial con la recolección generacional es el manejo de los conjuntos raíz. Es en esencia el mismo problema que se presenta en la recolección incremental. Las técnicas generacionales reducen el rango de recorrido hecho en la mayoría de las recolecciones, pero eso no reduce por sí misma el número de raíces que deben ser revisadas en cada recolección Si la generación más joven es pequeña y recolectada frecuentemente, y las variables globales y el stack son revisados cada vez, eso puede representar un costo significativo de la recolección de basura en sistemas de gran escala, ya que este tipo de sistemas pueden tener decenas de miles de variables globales o de módulos.

Una alternativa es considerar pocas cosas como parte del conjunto raíz usual, y tratar a la mayoría de las variables como si fueran objetos en el heap, con una barrera de escritura. Las referencias almacenadas en tales objetos pueden crear referencias hacia generaciones jóvenes y serán grabadas en la forma usual, así que se pueden encontrar las referencias en tiempo de recolección. El problema con este enfoque es que puede incrementar significativamente el costo de almacenamiento en las variables locales. Este costo puede reducirse si el compilador puede determinar tipos en tiempo de compilación, y omitir la barrera de escritura para almacenamiento de cosas que no son referencias.

El trato de variables locales es más complicado en lenguajes que soportan *cerraduras* (*closures*) –procedimientos que pueden capturar ambientes locales de enlace de variables, forzando a las variables locales a ser alojadas en el heap que es recolectado, en lugar de en el stack. Si todos los enlaces de una variable están alojados en el heap, esto requiere que el almacenamiento de referencias en variables locales utilicen una barrera de escritura; esto puede incrementar significativamente los costos de la barrera de escritura para muchos programas, en los cuales los efectos secundarios a través de variables locales son muy comunes. Para tales sistemas, es deseable tener optimizaciones del compilador que eviten que variables locales, que no serán referenciadas nunca desde cerraduras, sean almacenadas en el heap y en su lugar sean puestas en el stack o los registros. Tales técnicas son valiosas por sí mismas ya que mejoran la velocidad del código que opera en esas variables, al igual que reducen la asignación promedio al heap.

En la mayoría de los programas de aplicación, el tiempo de recorrido del conjunto raíz es despreciable, ya que solo hay unos cuantos miles de variables globales o de módulos. Sin embargo, en sistemas grandes, con ambientes de desarrollo de programación integrados, este conjunto puede ser muy grande; para evitar grandes cantidades de tiempo invertidas en recorrer el conjunto raíz en cada recolección, es de gran utilidad utilizar una barrera de escritura para que sólo las variables en las que una referencia ha sido almacenada sean revisadas en la siguiente recolección.

4.7 Recolección generacional de tiempo real

La recolección generacional de basura puede combinarse con técnicas incrementales, pero este matrimonio no es particularmente feliz. Típicamente, la recolección de basura de tiempo real está orientada a proveer garantías absolutas en el peor de los casos, mientras que las técnicas generacionales mejoran el rendimiento esperado a cambio de un mal comportamiento en el peor de los casos. Si la heurística generacional falla, y la mayoría de los datos viven largo tiempo, recolectar las generaciones jóvenes será una pérdida de esfuerzo, ya que no se recuperará absolutamente nada de espacio. En ese caso, la recolección completa de basura deberá proceder tan rápido como si el recolector fuera un simple esquema no-generacional incremental.

Sin embargo, la recolección de basura generacional de tiempo real puede ser deseable para muchas aplicaciones, siempre y cuando el programador pueda dar garantías acerca de los tiempos de vida de los objetos, para asegurar que el esquema generacional será efectivo. Alternativamente, el programador puede suplir garantías más débiles, a riesgo de encontrarse con una falla para cumplir los plazos establecidos si una de esas garantías es

errónea. El primer tipo de razonamiento es necesario para sistemas de tiempo real duro, y es necesariamente específico a cada aplicación. El segundo enfoque es adecuado para muchas otras aplicaciones tales como programas de control interactivos de audio y video, donde la posibilidad de una reducción en la respuesta no es fatal.

Cuando es deseable combinar técnicas generacionales e incrementales, los detalles del esquema generacional pueden ser importantes para permitir un apropiado rendimiento incremental. Por ejemplo, los recolectores para las máquinas Lisp (Symbolics, LMI y TI) son los sistemas generacionales de tiempo real mejor conocidos, pero las interacciones entre sus características generacionales e incrementales tienen un efecto negativo en su rendimiento en el peor caso.

En lugar de recolectar las generaciones viejas lentamente durante el curso de varias recolecciones a varias generaciones más jóvenes, sólo una recolección de basura se lleva a cabo en todo momento, y la recolección recolecta sólo la generación más joven, o las dos más jóvenes, o las tres más jóvenes, etc. Esto es, cuando una generación vieja es recolectada, ella y todas las generaciones más jóvenes son vistas como una sola generación, y se recolectan juntas. Esto hace imposible beneficiarse del efecto generacional de las generaciones jóvenes mientras se recolectan las viejas; en el caso de una recolección completa de basura, el sistema se degenera en un esquema de copiado incremental no-generacional.

Durante tales recolecciones de gran escala, el recolector debe operar lo suficientemente rápido para terminar el recorrido antes de que el espacio libre se agote —no hay generaciones jóvenes de donde el sistema pueda recuperar espacio y reducir la razón de recorrido seguro. Alternativamente, la velocidad de recolección puede mantenerse constante, pero los requerimientos de espacio serán mucho mayores durante las recolecciones de gran escala. Para programas con una cantidad considerable de datos de vida larga, este esquema puede esperar pérdidas periódicas y sistemáticas de rendimiento, aún si el programa tiene una distribución de vida favorable para la recolección generacional, y el programador puede proveer las garantías apropiadas al recolector. El recolector debe operar mucho más rápido durante recolecciones totales, o el consumo de memoria subirá dramáticamente. La primera opción ocasiona degradación del rendimiento porque el recolector utiliza la mayor parte de los ciclos del CPU; la segunda requiere grandes cantidades de memoria –negando la ventaja de la recolección generacional– o incurren en una degradación del rendimiento debido a la paginación de la memoria virtual.

Capítulo 5

Consideraciones de localidad

Las estrategias de recolección de basura tienen un efecto determinante en la forma que la memoria es usada y reutilizada; naturalmente, esto tiene un impacto significativo en la localidad de referencia.

5.1 Variedades de localidad de referencia

Los efectos de localidad de la recolección de basura pueden dividirse en tres clases:

1. Efectos sobre el estilo de programación que cambia la forma en que las estructuras de datos son creadas y manipuladas,

2. Efectos directos sobre el proceso de recolección de basura mismo, y

3. Efectos indirectos de la recolección de basura, especialmente patrones de reasignación de memoria libre y racimos de datos vivos.

El primero de estos efectos —aquellos causados por el estilo de programación— son pobremente entendidos. En sistemas con un recolector de basura eficiente, los programadores pueden adoptar un estilo que es apropiado para la tarea inmediata, frecuentemente un enfoque funcional u orientado a objetos. Los nuevos datos serán asignados a memoria dinámicamente para contener nuevos datos calculados, y los objetos serán desechados cuando los datos no sean interesantes. Idealmente, el programador expresa los cómputos en la forma más natural, con mapeos relativamente directos de los datos al nivel de aplicación a datos al nivel del lenguaje de programación.

En contraste, la asignación explícita y recuperación de memoria muy frecuentemente provocan un estilo de programación deformado en el cual el programador reutiliza objetos al nivel del lenguaje para representar datos conceptualmente distintos durante el transcurso de la ejecución, simplemente porque es muy costoso liberar el espacio ocupado por un objeto y luego asignar otro objeto nuevo. Similarmente, en sistemas con recolección de basura ineficiente (tal como la mayoría de las implementaciones de Lisp antiguas) los programadores muy frecuentemente recurren a este tipo de reutilización de objetos de nivel de lenguaje, por ejemplo destruyendo como efecto secundario la estructura de una

lista para evitar asignar nuevos elementos a la lista, o asignar un solo arreglo grande usado para mantener varios tipos de datos a lo largo del tiempo. La recuperación o liberación explícita de memoria generalmente nos lleva a distorsiones de otra variedad, al mapear un solo dato conceptual en varios objetos a nivel del lenguaje. Los programadores pueden asignar muchos objetos extra para simplificar la liberación explícita. Cualquier módulo interesado en una estructura de datos puede copiar la estructura de datos, de tal forma que pueda hacer una decisión local de cuándo la memoria puede ser reclamada. Tales distorsiones hacen extremadamente difícil comparar directamente la localidad de referencia de sistemas con recolección de basura y sin recolección de basura. Los estudios típicos que lo intentan, comparan programas en uso, escritos sin tener la recolección de basura en mente, ambos en su forma original y con la asignación y liberación explícita sustituida por recolección de basura. Esto implícitamente *esconde* los efectos provocados por estilos de programación distorsionados.

La segunda categoría de efectos de localidad –localidad del proceso de recolección de basura mismo– son los primeros que vienen a la mente. Son, en algunos casos, muy significativos; sin embargo, puede ser el menos importante de los tres. Para llevar a cabo una recolección completa de basura, todos los datos vivos deben ser recorridos, y esto puede interactuar muy pobremente con jerarquías de memoria convencionales. La mayoría de los objetos vivos serán tocados una sola vez durante la recolección, así que habrá muy poca localidad *temporal*, i.e. muy pocos toques repetidos al mismo dato en un periodo corto de tiempo. Por otro lado, puede existir una considerable localidad *espacial* –se tocan varias áreas contiguas (o muy cercanas) de memoria (v.gr. dentro de la misma página de memoria virtual) en periodos de tiempo muy cortos.

La tercera categoría de los efectos es probablemente la más importante, pero su relevancia no es ampliamente apreciada. La estrategia para *reasignación* de memoria impone características de localidad en la forma en que la memoria es tocada en forma repetida, *aún si los objetos mismos mueren rápido y nunca más son tocados.* Esta es una de las principales razones para tener recolección de basura generacional –reutilizar pequeñas áreas de memoria (la generación más joven) repetidamente asignando la mayoría de los objetos de vida corta allí. Es también la razón por la que los stacks de activación típicamente tienen excelente localidad de referencia —cerca del tope del stack la memoria es reutilizada muy frecuentemente, asignando muchos objetos de vida corta, como son los registros de activación allí. Un recolector generacional puede ser visto como una imposición al heap, que lo fuerza a tener un patrón de reuso parecido al de un stack, explotando el hecho de que las distribuciones de vida son similares.

Como mencionamos anteriormente, un recolector de basura simple no-generacional tiene una localidad muy pobre si el ritmo de asignación es considerablemente alto, simplemente porque se toca mucha memoria en cualquier periodo de tiempo. Ungar señala que el simple hecho de paginar (mandar al área de swap una página de memoria virtual) la basura resultante de la recolección, representa un costo inaceptable para un sistema de alto rendimiento. Un recolector de basura generacional restringe el alcance de este desastre de localidad a un área manejable –la generación más joven, o las dos más jóvenes. Esto permite tener características de localidad *reales* (de accesos repetidos a datos de larga vida) que se muestren en las generaciones más viejas. Es por este efecto, que las características

de localidad promedio de los sistemas con recolección de basura, parecen ser comparables a aquellos sistemas sin recolección de basura –la generación más joven filtra la mayoría de los datos alojados en el heap que pueden ser alojados en el stack en otros lenguajes.

Sin embargo, hacer comparaciones directas es difícil, por el número de decisiones de diseño involucradas tanto en los recolectores de basura como los de manejo explícito del heap.

Con recolección de basura con copiado, hay otro efecto de localidad indirecto — al mover objetos dato en la memoria, el recolector afecta la localidad de referencia de los accesos generados por el programa— esto es, afecta el mapeo entre las referencias lógicas del programa con datos de nivel del lenguaje y de las referencias resultantes con localidades particulares de memoria.

Este tipo de efecto no está limitado sólo a los recolectores de copiado. Los recolectores sin copiado también tienen una brecha considerable para moverse al decidir cómo mapear los objetos del nivel del lenguaje en la memoria disponible. Cuando el programa solicita una pieza de memoria de un tamaño dado, la rutina de asignación es libre de regresar *cualquier* pieza de memoria libre siempre y cuando sea del tamaño adecuado. No es claro qué criterios deben gobernar esta decisión[1].

La importancia de los distintos efectos de la localidad es dependiente del tamaño relativo de los datos usados por un sistema y la memoria principal del hardware en el que se ejecuta. Para muchos sistemas, el heap completo cabe en memoria principal, incluyendo un editor, un compilador y código y datos de la aplicación; en tales sistemas, la memoria RAM es suficiente y –en general– los programas no tienen que paginarse. Sin embargo, en otros sistemas, la generación más vieja o las últimas dos son muy grandes para caber en la memoria principal, ya sea porque el sistema mismo incluye otras aplicaciones de software complejas o porque el código y datos de la aplicación son demasiado grandes.

En un recolector con copiado, hay una distinción binaria entre generaciones que residen en memoria y las que son muy grandes y dependen de la memoria virtual. Para las primeras, que bien puede ser todo el sistema, la localidad a la hora de asignar memoria es irrelevante para el desempeño de la memoria virtual –el costo en espacio es el costo fijo de mantener todo en la memoria RAM. Para el segundo tipo de generaciones, en contraste, la localidad al nivel de la memoria virtual es crucial.

En un recolector que no mueve datos, la situación es distinta –el costo en espacio de la generación más joven también depende del grado de fragmentación y de cómo los datos de varias generaciones están mezclados en la memoria. La magnitud de estos problemas no es totalmente entendida aún en nuestros días; sin embargo, algunos estudios han mostrado que sus efectos son menos serios de lo que es considerado.

[1]Por supuesto, esto también ocurre en los sistemas con manejo explícito del heap, pero tampoco ha sido sistemáticamente estudiado en esa área La mayoría de los estudios de técnicas de asignación explícita estudian la fragmentación y los costos de CPU, pero han ignorado los efectos provocados por el almacenamiento en memoria inmediata (cache) de memorias jerárquicas

5.2 Localidad y objetos de vida corta

Como se puede apreciar en la discusión previa, el patrón de asignación de memoria tiene un efecto muy importante en la localidad de un recolector sencillo. En un recolector con copiado generacional, este efecto se ve reducido desde el punto de vista de la memoria virtual –las páginas que conforman la generación más joven o las dos generaciones más jóvenes, son reutilizadas tan frecuentemente que se mantienen en RAM e incurren en un costo de espacio fijo. Por otro lado, la reutilización frecuente de la generación más joven puede tener un efecto negativo en el siguiente nivel más pequeño en la jerarquía de memoria –caches de alta velocidad. El ciclo de reutilización de memoria se ha hecho más pequeño, pero si el ciclo no entra en el cache, el cache sufrirá fallas extras, de la misma manera que la memoria principal en los recolectores sencillos. Los efectos de tales fallas no son tan dramáticos como los de la memoria principal, ya que la velocidades del cache a la memoria principal es varios órdenes de magnitud más rápido que el de la memoria principal al área de swap.

Si se utiliza recolección con copiado, el número de fallas puede reducirse considerablemente debido a que el patrón de asignación de memoria es fuertemente secuencial, ya sea utilizando estrategias de pre-cargado o utilizando bloques de tamaño mayor. En los procesadores modernos, el costo es un porcentaje muy pequeño del tiempo de ejecución, aún cuando los ritmos de asignación son relativamente grandes. Entre más rápido es el procesador, más sufre los efectos de las fallas en el cache, sobre todo porque el ancho de banda entre el cache y la memoria no se escala al mismo ritmo que la velocidad de los procesadores (véase la Sección 1).

Zorn mostró que caches más grandes tienen efectos positivos para sistemas con recolectores de basura generacionales. Wilson, por otro lado, mostró que los tamaños relativos del cache y de la generación más joven son particularmente importantes y que el rendimiento puede ser mejor cuando son aproximadamente del mismo tamaño. Las peculiaridades de la política de reemplazo en el cache son también importantes, debido al patrón de reasignación de memoria.

Muchos investigadores han sugerido optimizaciones para evitar recolectar basura en áreas que están a punto de ser reasignadas; esto puede reducir el requerimiento de ancho de banda entre el cache y la memoria casi a la mitad.

La diferencia en localidad entre recolectores con copiado y sin copiado no es muy grande cuando tenemos memorias cache de alta velocidad –el tipo de recolector no es tan importante como el ritmo de asignación y el tamaño de la primera generación, i.e. que tan rápido se usa, recupera y reusa la memoria en el caso promedio.

5.3 Localidad de recorrido de memoria

La localidad de los recorridos de memoria de un recolector de basura es difícil de estudiar por separado, pero tiene algunas características obvias. La mayoría de los objetos en la generación que está siendo recolectada serán tocados exactamente una vez, ya que la mayoría de los objetos son referidos desde exactamente otro objeto –las estructuras de datos usuales no contienen un gran número de ciclos, y la mayoría de los ciclos son lo suficientemente pequeños por lo que su impacto en la localidad de un recorrido es despreciable.

Considerando lo anterior, notamos que la mayor característica del recorrido es tocar todos los datos vivos exhaustivamente, pero en la mayoría de los casos, brevemente. Hay muy poca localidad de referencia *temporal*, i.e revisiones repetidas de los mismos datos. La mayor característica de localidad que puede ser explotada, es la de localidad *espacial* de las estructuras de datos en la memoria —si objetos que se encuentran ligados, se encuentran cerca unos de otros en la memoria, cuando el recorrido toca a uno de ellos, la probabilidad de que todos los objetos cercanos sean llevados a la memoria rápida,antes de ser alcanzados por el recorrido, es muy alta.

En un recolector con copiado, los recorridos tienen buena localidad de referencia, ya que los objetos se organizan por el orden de recorrido cuando son copiados por primera vez, y subsecuentes recorridos seguirán el mismo orden. Por supuesto que durante el primer recorrido, la relación entre la rutina de asignación de memoria y el recorrido del recolector es significativa.

5.4 Agrupamiento de objetos de larga vida

Han habido muchos estudios acerca del efecto indirecto sobre la localidad que tienen los recolectores con copiado –es decir, el efecto que provoca el reorganizar los datos que serán subsecuentemente accedidos por el programa en ejecución. A continuación discutiremos brevemente algunos de los resultados más importantes.

5.4.1 Agrupamiento estático

Estudios sobre los efectos de distintos algoritmos de recorrido y copiado para reorganizar objetos de larga vida, fueron realizados en sistemas de Smalltalk. Estos algoritmos reorganizan los datos durante la ejecución del programa (i e. en tiempo de recolección de basura); sin embargo, son conocidos como algoritmos de *agrupamiento estático* porque reorganizan los datos de acuerdo a cómo están ligados los datos al momento que la recolección de basura ocurre –el agrupamiento no está basado en el patrón de acceso a los objetos por parte del programa en ejecución.

En dichos estudios la conclusión fundamental es que conviene más utilizar búsqueda en profundidad que búsqueda en amplitud, aunque no por un margen muy amplio. Hay información importante de localidad en la topología de las estructuras de datos, pero cualquier recorrido razonable hará un trabajo razonable reorganizando los datos. También mostraron que tanto búsqueda en profundidad como en amplitud son claramente superiores sobre una reorganización aleatoria.

Más adelante Wilson *et al.* realizaron un estudio similar pero en esta ocasión para un sistema Lisp, y mostraron que distintos algoritmos de recorrido producen diferencias considerables sobre la localidad La diferencia más importante en esos experimentos no era entre los algoritmos de recorrido mismos, sino en la forma en que grandes tablas de dispersión (hash) eran tratadas Los datos de un sistema son generalmente almacenados en tablas de dispersión, que implementan grandes ambientes para ligado de variables, tales como el espacio de nombres globales o un paquete Las tablas de dispersión guardan sus datos en un orden pseudo aleatorio, y esto puede provocar que un recolector con copiado recorra y

copie las estructuras de datos en una forma pseudo aleatoria. Esto reduce grandemente la localidad, pero es fácil de evitar tratando las tablas de dispersión de manera distinta[2].

Una vez que las tablas de dispersión son tratadas adecuadamente, algunas mejoras a la localidad pueden hacerse usando un algoritmo que agrupe las estructuras de datos jerárquicamente.

5.4.2 Reorganización dinámica

En 1980, *White* propuso un sistema en el cual la recolección de basura era postergada por largos periodos de tiempo, pero en el cual el algoritmo de copiado de la barrera de lectura de Baker podía copiar datos por sus efectos de localidad; la meta no era recuperar espacio vacío, sino simplemente agrupar los datos activos para que pudieran permanecer en memoria principal. Una propiedad interesante de este esquema es que reorganiza los datos en el orden en que son tocados por el mutante, y si los accesos subsecuentes son similares, la mejora en la localidad puede ser muy grande.

En su forma original este esquema es impracticable (ya que el volumen de basura no reclamada sobrepasaría el ancho de banda de la barrera en discos típicos), pero la misma idea básica ha sido incorporada en el recolector de las máquinas *Texas Instruments Explorer Lisp*. Este recolector evita realizar barridos exhaustivos hasta cerca del final del ciclo de recolección, para mejorar la probabilidad de que los objetos sean alcanzados primero por el mutante, y copiados en un orden que mejora la localidad.

Un enfoque similar fue utilizado en simulaciones realizadas por el proyecto *MUSHROOM* de la Universidad de Manchester. En lugar de apoyarse en el recolector, el sistema se dispara por fallas en el cache; por lo que puede responder directamente a las características de localidad de un programa, en lugar de responder a la interacción entre el programa y el recolector de basura.

Desafortunadamente, tales esquemas dependen de hardware especializado para que valgan la pena. El sistema Explorer explota el soporte de hardware para una barrera de lectura tipo Baker, y el sistema MUSHROOM está basado en una arquitectura nueva "orientada a objetos".

Wilson clasifica estas técnicas como una forma de pre-selección que se adapta dinámicamente, y concluye que conforme las memorias sigan creciendo, tales reorganizaciones tan finamente especializadas estarán de más. Conforme las memorias crecen, la unidad óptima del cache se hace más grande también, y los esquemas basados en paginación tienden a funcionar tan bien como los esquemas basados en objetos.

5.4.3 Coordinación con la paginación

Muchos sistemas han coordinado la recolección de basura con el sistema de memoria virtual para mejorar el rendimiento de la paginación.

[2]Una técnica es modificar la estructura de las tablas de dispersión para almacenar el orden en el cual se crean las entradas, o imponer algún ordenamiento no aleatorio y después modificar el recolector para que utilice esta información para ordenar su recorrido de las entradas en la tabla. Otra técnica es hacer las tablas de dispersión como índices indirectos a un arreglo ordenado de entradas; esto tiene la ventaja de que puede ser implementado sin modificar el recolector y por lo tanto, puede ser usado incluso para estructuras en forma de tabla creadas por el usuario.

Las máquinas Symbolics Lisp son un buen ejemplo de este tipo de coordinación. El asignador de memoria de estas máquinas, notificaba al sistema de memoria virtual cuando una página ya no contenía información importante.

La cooperación de la memoria virtual también es usada en el mecanismo de almacenamiento de referencias intergeneracionales. Antes de que una página sea enviada al área de swap, la página es revisada y las referencias intergeneracionales encontradas. Esto le permite al recolector evitar paginar datos sólo para revisar si tienen referencias a generaciones más jóvenes. También se aplicaron técnicas de mapeo de memoria virtual para optimizar el copiado de objetos grandes –en lugar de copiar la información almacenada en un objeto grande, las páginas que contienen la información pueden ser mapeadas fuera de su viejo rango de direcciones e insertadas en el nuevo rango.

Recientemente, sistemas con operaciones de microkernel ofrecen la habilidad de modificar políticas de memoria virtual sin modificar el kernel. El kernel llama rutinas especificadas por el usuario para controlar la paginación de un proceso, en lugar de utilizar rutinas con una política de paginación preestablecida en el kernel mismo. En los sistemas *Mach* por ejemplo, un proceso *externo de paginación* puede ser usado para controlar la actividad de paginación; esta característica ha sido utilizada para reducir la paginación de Standard ML of New Jersey de maneras similares a aquellas utilizadas en los sistemas Symbolics.

Capítulo 6

Detalles de implementación de bajo nivel

Hasta este momento, sólo hemos discutido detalles básicos sobre el diseño de recolectores de basura, y los distintos intercambios entre características y rendimiento. Además de estas consideraciones básicas, un diseñador de un recolector de basura se encuentra con muchas opciones de diseño que pueden tener un impacto significativo en el rendimiento final del sistema, y en qué tan fácilmente el recolector puede ser integrado con otros componentes del mismo. En este capítulo, discutiremos estas opciones de diseño, las cuales tienen que ver con la implementación de bajo nivel.

6.1 Referencias etiquetadas y encabezados de objetos

En gran parte de este trabajo hemos supuesto que las referencias tienen pequeñas etiquetas y que los objetos referidos tienen un encabezado que codifica más información sobre el tipo; esta información específica de tipo puede ser usada para determinar la disposición del objeto, incluyendo la localización de campos de referencia internos. Este es el esquema más común en lenguajes dinámicamente tipificados, como es el caso de Lisp, Smalltalk o Scheme. Es común en tales lenguajes que los objetos estén divididos en dos categorías: aquellos que contienen sólo valores etiquetados (valores inmediatos y referencias) que deben ser examinados por el recolector para encontrar las referencias, y aquellos que contienen sólo campos no-referencia que pueden ser ignorados por el recolector de basura.

Otra posibilidad es que cada campo contenga tanto el objeto (si es un valor inmediato pequeño) o una referencia, y la información detallada del tipo. Esto generalmente requiere que los campos sean dos palabras –una palabra lo suficientemente grande como para contener una referencia o un valor inmediato sin procesar, y otra palabra para mantener un patrón de bits lo suficientemente grande como para codificar todos los tipos en el sistema. Generalmente, una palabra completa se usa para almacenar la información del tipo, debido a restricciones de alineamiento para operaciones `load` y `store`[1]. A pesar del desperdicio de espacio, este esquema puede ser atractivo en algunas arquitecturas, especialmente aquellas

con buses amplios.

Para un lenguaje con un sistema de tipos estático, hay aún una posibilidad más y es que todos los objetos tengan encabezados, y que las referencias no contengan etiquetas. Esto requiere de un sistema de tipos estático que asegure que no se puedan almacenar valores inmediatos en los mismos campos que las referencias –si pudieran, entonces necesitaríamos etiquetas para distinguir entre valores inmediatos y referencias. Simplemente saber qué campos de un objeto pueden contener referencias es suficiente, siempre y cuando los objetos referidos tengan encabezados para decodificar su estructura. Algunos sistemas dinámicamente tipificados utilizan esta representación también, y evitan tener valores inmediatos en una sola palabra –aún enteros pequeños se representan como objetos con encabezado[2].

Si se utiliza un esquema de tipos estático muy rígido, incluso los encabezados pueden ser eliminados –una vez que un campo de referencia es encontrado, y el tipo del objeto referido es conocido, el tipo del objeto al que hace referencia es obvio. Para poder llevar a cabo los recorridos de memoria, sólo es necesario saber los tipos de las referencias en el conjunto raíz (v.gr., variables locales en un stack de activación); más adelante explicaremos varias formas de lograr este objetivo. A partir del conjunto raíz, podemos determinar los tipos de los objetos referidos, y por lo tanto los campos referencia de los objetos referidos, y así sucesivamente. Aun así, algunos sistemas estáticamente tipificados ponen encabezados en los objetos, ya que el costo no es muy alto y simplifica algunos aspectos de la implementación[3].

La decisión de etiquetas y esquemas de referencias usualmente se toma considerando la recolección de basura, pero hacer que las operaciones normales de un programa sean eficientes es la consideración más importante. Los esquemas de etiquetas se escogen principalmente para lograr que el envío de tipos, las operaciones aritméticas y la resolución de referencias sean lo más rápidas posibles; cuál esquema es mejor depende en gran medida de la semántica del lenguaje y de la estrategia para asegurar que las operaciones más frecuentes sean rápidas.

En algunos sistemas, los objetos individuales no tienen encabezados, y la información de tipo es codificada segregando objetos de tipos particulares en conjuntos separados de páginas. Estas "grandes bolsas de páginas" (*big bags of pages*) o técnica *BiBOP*, por sus siglas en inglés, asocia tipos con páginas, y restringe a la rutina de asignación a asignar a los objetos en las páginas apropiadas. Un chequeo de tipos requiere poner una máscara y recorrer una referencia para derivar el número de página, y después una búsqueda en una tabla para encontrar el descriptor del tipo para los objetos en esa página. La codificación BiBOP puede ahorrar espacio permitiendo que una etiqueta por página sea suficiente para codificar los tipos de muchos objetos.

Otra variante de los sistemas con etiquetas convencionales, consiste en evitar poner encabezados a objetos directamente en los objetos, y salvarlos en un arreglo paralelo; esto puede tener ventajas para la localidad de referencia, ya que separa la información que es

[1]En muchas arquitecturas, las operaciones comunes `load` y `store` deben ser alineadas en los límites de una palabra, y en otras hay un castigo en tiempo por accesos no alineados.

[2]Típicamente, esto requiere de implementaciones muy ingeniosas para optimizar la asignación al heap de la mayoría de los enteros.

[3]Por ejemplo, si se utiliza marcado de páginas o de cartas para recolección generacional, los encabezados hacen mucho más simple el recorrer las páginas.

relevante para el programa de la información que sólo concierne al recolector y a la rutina de asignación de memoria.

6.2 Localización conservadora de referencias

Otro aspecto a considerar acerca de la implementación de un lenguaje es la *localización conservadora de referencias*, que es una estrategia para copiar con compiladores que no ofrecen *ningún* soporte para identificación de tipos en tiempo de ejecución[4]. En tales sistemas, el recolector trata cualquier cosa que *pueda* ser una referencia como tal –v.gr. cualquier patrón de bits alineados apropiadamente que pueda ser la dirección de un objeto en el heap. El recolector puede confundir algunos valores (tales como un entero con la misma secuencia de bits) con referencias, y retener objetos innecesariamente, pero algunas otras técnicas pueden ser usadas para que la probabilidad de tales errores sea pequeña. Sorprendentemente, esta técnica es lo suficientemente efectiva para que la mayoría de los programas en C puedan ser recolectados eficientemente, con muy poca o sin modificación alguna. Esto simplifica la recolección de basura de programas escritos sin tener a la recolección de basura en mente, y programas escritos en varios lenguajes, algunos de los cuales son poco o nada cooperativos.

Hacer un recolector de este tipo generacional requiere técnicas especiales, debido a la falta de cooperación del compilador al implementar una barrera de escritura para detectar referencias intergeneracionales. Se pueden usar los bits de suciedad (dirty bits) de la memoria virtual o trampas de protección para detectar qué páginas son escritas, para que puedan ser revisadas en tiempo de recolección y detectar así referencias a generaciones más jóvenes.

La localización conservadora de referencias impone restricciones adicionales sobre el recolector de basura. En particular, el recolector no es libre de mover objetos ni de actualizar referencias, ya que una no-referencia puede ser confundida erróneamente con una referencia y erróneamente actualizada, lo que puede introducir cambios misteriosos e inexplicables en datos no-referencia como enteros y cadenas de caracteres. Por lo tanto, los recolectores conservadores no pueden usar un algoritmo de recorrido directo.

La localización conservadora de referencias puede combinarse con otras técnicas para hacer frente a implementaciones de lenguajes que cooperan mínimamente. Por ejemplo, los recolectores "mayoritariamente-con-copiado" (mostly-copying) de *Barlett* y *Detlefs*, utilizan encabezados para decodificar los campos de objetos en el heap, pero dependen de técnicas conservadoras para encontrar referencias del stack de activación. Esto soporta técnicas de copiado que reasignan y compactan la mayoría de (pero no todos) los objetos. Los objetos identificados conservadoramente como referenciados desde el stack, son "fijados" en su sitio y no pueden ser movidos[5]

---

[4]Estas técnicas son generalmente asociadas con Boehm y sus asociados, quienes las han desarrollado hasta un grado de sofisticación muy alto; sin embargo, técnicas similares aparecieron anteriormente en el sistema *Kyoto Common Lisp* y tal vez en algunos otros sistemas

[5]Tales objetos son fijados en su sitio en la memoria, pero pueden ser avanzados a una generación más vieja cambiando el conjunto al cual la página pertenece –esto es, los objetos pertenecen a páginas, y las páginas pertenecen a generaciones; sin embargo una página completa puede ser cambiada de generación simplemente cambiando las tablas que indican qué tablas pertenecen a qué generaciones. En esencia, las edades de los objetos son codificadas utilizando algo como el esquema de etiquetado BiBOP

La decisión de escoger revisión conservadora del stack y no técnicas más precisas de revisión del heap, es razonable en la mayoría de los casos, ya que es más fácil modificar encabezados de objetos dentro de un lenguaje que modificar el compilador para lograr que los formatos de los registros de despliegue (*display registers*) del stack sean decodificables. Los encabezados pueden ser añadidos a objetos por una rutina de asignación al heap, que puede simplemente ser una rutina de biblioteca susceptible de cambiarse o sustituirse fácilmente. Los compiladores frecuentemente graban información suficiente para decodificar formatos de registros, para efectos de depuración de programas, y tal información puede ser capturada y "ajustada" para convertirla en identificación de tipos para objetos alojados en el heap.

La localización conservadora de referencias puede ser derrotada por enunciados al nivel del lenguaje tales como la habilidad de convertir referencias en apuntadores (con un *cast*), destruir la referencia original, y realizar operaciones aritméticas arbitrarias sobre el valor entero. Si el valor original es recuperado y se vuelve a convertir en una referencia, el objeto referido puede ya no existir –el recolector de basura pudo haber recuperado al objeto ya que no tenía forma de saber que el valor entero "apuntaba" al objeto, aún cuando éste era visto como una referencia. Afortunadamente, la mayoría de los programas no llevan a cabo esta secuencia de operaciones –pueden convertir la referencia a un entero, pero la referencia original muy probablemente seguirá allí y el objeto será retenido.

Algunas optimizaciones realizadas por el compilador pueden hacer operaciones similares sobre referencias, y esto es –desafortunadamente– más difícil de evitar para el programador, ya que el compilador puede utilizar transformaciones algebraicas en expresiones referencia, disfrazando así las referencias, o pueden realizar operaciones sobre sub-palabras de una referencia, lo que ocasiona inconsistencias temporales en el estado de la referencia. Si bien muchos compiladores no llevan a cabo este tipo de optimizaciones, sí llegarán a ocurrir, e incluso algunos compiladores las hacen rutinariamente. Para la gran mayoría de compiladores, es suficiente desactivar los niveles más altos de optimización para evitar tales optimizaciones sobre referencias, pero no es una cuestión confiable para realizarla en distintos compiladores, y típicamente cuesta un porcentaje considerable en eficiencia debido a las optimizaciones perdidas. Dado que la gran mayoría de los compiladores no proveen la habilidad de desactivar selectivamente sólo aquellas optimizaciones que ocasionan problemas al recolector, usualmente es necesario desactivar muchas optimizaciones, i.e. las optimizaciones de alto nivel. Para evitar este problema, los diseñadores de recolectores de basura han propuesto un conjunto de restricciones que los compiladores pueden preservar para asegurar que la recolección de basura sea posible; estas restricciones no imponen cambios mayores a compiladores existentes.

Similarmente, algunos lineamientos de programación han sido propuestos para asegurar que los programadores de C++ eviten construcciones que imposibiliten llevar a cabo una correcta recolección de basura; esto se logra programando en un subconjunto levemente restringido de C++.

Algunas personas se oponen a las técnicas de localización conservadora de referencias, porque dichas técnicas no son "correctas" –es posible, por ejemplo, que un entero sea confundido con una referencia, provocando que el recolector de basura conserve basura. En el peor caso, eso puede ocasionar que un número considerable de basura sea mantenida en el sistema, y un programa puede simplemente quedarse sin memoria y caerse. Los que

soportan la técnica de localización conservadora de referencias, se inclinan a pensar que la probabilidad de que tal evento ocurra se puede hacer muy pequeña, y que tales errores son menos probables que aquéllos provocados por los mismos programadores al hacer un uso inadecuado del heap. El plan a largo plazo es que las técnicas de localización conservadora de referencias conviertan a la recolección de basura en algo ampliamente utilizado, y una vez que sea usada por muchos compiladores para distintos lenguajes, los creadores de compiladores proveerán soporte, al menos en cierta medida, para técnicas más precisas.

6.3 Soporte lingüístico y referencias inteligentes

Otro enfoque para añadir recolección de basura en sistemas existentes es usar las herramientas de extensión provistas por el lenguaje mismo. La forma más común de esto es implementar un conjunto especial de tipos de datos que son recolectados, con un conjunto restringido de funciones de acceso que preserven las restricciones del recolector de basura. Por ejemplo, un tipo de datos que utilice conteo de referencias puede ser implementado, y funciones de acceso (o macros) para hacer asignaciones –las funciones de acceso mantienen el contador de referencias y al mismo tiempo llevan a cabo las asignaciones.

Un enfoque un tanto más elegante es usar otras herramientas de extensión tales como sobrecarga de operadores para permitir que operadores comunes puedan ser usados sobre tipos recolectados, y hacer que el compilador automáticamente seleccione la operación apropiada definida por el usuario. En C++, es común definir clases de "referencias inteligentes" que pueden ser usadas con clases sujetas a recolección de basura, lo único que debemos cuidar es definir apropiadamente el significado de las operaciones para manipulación de referencias (tales como * y ->) y para el operador de indirección (&) sobre objetos recolectados. Desafortunadamente, estas referencias inteligentes definidas por el usuario, pueden ser usadas en exactamente las mismas formas que las referencias normales, que incluye el lenguaje. Hay varias razones: la primera es que no hay forma alguna de definir todas las coerciones que el compilador realiza automáticamente para tipos integrados al lenguaje. Otro problema es que no todas las operaciones pueden ser sobrecargadas de esta forma. C++ provee la mayoría de, pero no toda, la extensibilidad necesaria para integrar recolección de basura al lenguaje de manera natural. Algunos estudios prácticos han mostrado que aparentemente es más fácil integrar recolección de basura a Ada que a C++, ya que el sistema de sobrecarga es más poderoso y los tipos referencia integrados en el lenguaje tiene menos sutilezas que deben ser emuladas. Otra limitante más es que es imposible redefinir operaciones en clases integradas, por lo que sólo las clases generadas por los usuarios pueden ser recolectadas.

Una limitante más es que la recolección de basura es difícil de implementar *eficientemente* dentro del lenguaje, ya que es imposible decirle al compilador cómo compilar ciertos casos especiales que son importantes. Por ejemplo, en C++ o Ada, no hay forma de especializar una operación para objetos que se sabe en tiempo de compilación que no serán almacenados en el heap[6]

Trabajos más actuales con sistemas reflexivos (*reflective systems*) han explorado

[6] En terminología de C++, los operadores sólo pueden ser especializados sobre los tipos de sus argumentos, no sobre la *la clase de almacenamiento* de sus argumentos

lenguajes con mecanismos de extensión más estables y poderosos, y que *exponen* parte de la implementación interna para permitir re-implementaciones eficientes de algunas de las características del lenguaje.

6.4 Cooperación del compilador y optimizaciones

En cualquier sistema con recolección de basura, debe existir un conjunto de convenciones usadas tanto por el recolector de basura como por el resto del sistema (el código del intérprete o del compilador), para asegurar que el recolector de basura pueda reconocer tanto objetos como referencias. En una sistema con localización conservadora de referencias, el "contrato" entre el compilador y el recolector es muy débil, pero existe –si el compilador evita hacer optimizaciones extrañas que puedan burlar al recolector. En la mayoría de los sistemas, el compilador es mucho más cooperativo, asegurando que el recolector puede encontrar referencias en el stack y en los registros.

6.4.1 Recolección en cualquier momento Vs. recolección en puntos seguros

Típicamente, el contrato entre el recolector y el código en ejecución toma una de dos posibles formas, que llamaremos las estrategias de *recolección en cualquier momento* o *recolección en puntos seguros*. En un sistema con recolección en cualquier momento, el compilador asegura que el código en ejecución puede ser interrumpido en cualquier momento, y es seguro realizar la recolección –es decir, que se encuentra disponible la información necesaria para encontrar todas las variables referencia activas en ese momento, y decodificar sus formatos para que los objetos alcanzables sean encontrados.

En un sistema de puntos seguros, el compilador sólo asegura que la recolección de basura será posible en ciertos puntos seleccionados durante la ejecución del programa, y que esos puntos ocurrirán frecuentemente y de manera regular. En muchos sistemas, llamadas a procedimientos y ramas que regresan están garantizados como puntos seguros, asegurando que el programa no puede caer en un ciclo (o recursar) indefinidamente sin alcanzar ningún punto seguro –el tiempo más largo posible entre puntos seguros es el tiempo que toma una rama directa en un procedimiento, sin llamadas a funciones. Se pueden introducir puntos intermedios para ofrecer mayor flexibilidad al recolector, si esto resulta necesario.

La ventaja del esquema de puntos seguros es que el compilador está libre de hacer optimizaciones arbitrariamente complejas dentro de las regiones entre puntos seguros, y no está obligado a almacenar la información necesaria para localizar y des-optimizar valores de referencias[7]. Una desventaja de los puntos seguros es que restringe las estrategias de implementación para procesos ligeros, o hilos de control (*threads*). Si varios hilos se ejecutan simultáneamente en el mismo heap, que está siendo recolectado, y un hilo obliga al recolector a ejecutarse, la recolección debe esperar hasta que todos los hilos han alcanzado un punto seguro y se detengan.

[7]Esto es particularmente importante cuando se usa un lenguaje de alto nivel, C por ejemplo, como si fuera un lenguaje intermedio, donde a veces es deseable pero imposible prevenir que el compilador de C utilice optimizaciones que enmascaren referencias.

Una forma de implementación de este esquema es poner máscaras sobre las interrupciones del hardware entre puntos seguros; otra forma más común es proveer una pequeña rutina que pueda manejar la interrupción de bajo nivel en cualquier momento, simplemente guardando información básica al respecto, prendiendo una bandera y permitiendo que la ejecución se complete. El código compilado checa la bandera en cada punto seguro, y mueve el control a un manejador de interrupciones de alto nivel si la bandera está activada. Esto introduce la noción de interrupciones de alto nivel, que se encuentran de alguna manera aisladas de las interrupciones del hardware –y por supuesto, con una latencia mayor.

Ya sea con puntos seguros o recolección en cualquier momento, hay varios convencionalismos para asegurar que las referencias pueden ser identificadas por el recolector de basura; sin embargo, con un sistema de puntos seguros, el compilador tiene la libertad de violar la convención cuando se encuentra entre puntos seguros.

6.4.2 Conjuntos de registros particionados Vs. almacenamiento de representaciones variables

En muchos sistemas, el compilador respeta una convención muy simple sobre qué registros pueden ser utilizados para mantener qué tipo de valores. En el sistema *T* (utilizando el compilador *Orbit*), por ejemplo, algunos registros son utilizados solamente para valores etiquetados, y otros para valores no-referencia directos. Las referencias se supone que son objetos en el formato común –referencias directas a un desplazamiento (*offset*) conocido dentro del objeto, más una etiqueta; de esta forma, los encabezados pueden ser extraídos directamente de los objetos referidos por una simple instrucción `load` indexada.

Otros tipo de particiones del conjunto de registros y convencionalismos son posibles. Por ejemplo, sería posible tener registros que mantengan referencias directas, tal vez referencias en cualquier parte dentro de un objeto, si el recolector puede asegurar que los encabezados de los objetos pueden ser derivados utilizando restricciones de alineamiento. Si se usa un recolector sin copiado, se pueden permitir registros "no rastreados" para mantener referencias optimizadas (que bien pueden no apuntar hacia ningún objeto, debido a transformaciones algebraicas), siempre y cuando una referencia no optimizada al mismo objeto se encuentre en un registro "rastreado" en un formato conocido.

El problema principal con las particiones de registros es que reduce la libertad del compilador para asignar variables de programa y temporales en cualquier registro disponible. Algunas secuencias de códigos pueden requerir varios registros de referencias y muy pocos registros de no-referencias, y otras secuencias de código exactamente lo contrario. Esto significa que más variables tendrán que ser "derramadas", i.e. alojadas en el stack o en el heap en lugar de ocupar un registro para ellas, y por ende el código se ejecutará más lentamente.

Una alternativa a la partición en conjuntos de registros es dar al compilador más libertad en su uso de los registros, pero requerirle que comunique más información al recolector de basura –i.e., que le diga en dónde están las referencias y cómo interpretarlas apropiadamente. Llamamos a esta estrategia *almacenamiento de representaciones variables*, ya que el compilador debe registrar todas sus decisiones acerca de qué registros (o variables de stack) mantienen referencias, y cómo recobrar la dirección del objeto si es que la referencia ha sido transformada por alguna optimización Para cada rango de instrucciones donde difieran

representaciones variables de referencias, el compilador debe emitir alguna anotación. Esta información es similar a la requerida para depurar código optimizado, y la mayoría de las optimizaciones pueden ser soportadas con una muy leve sobrecarga (provocada por las anotaciones extras). Los requerimientos de espacio adicionales para añadir esta información (que esencialmente anota el código ejecutable) pueden ser considerables; sin embargo, puede ser deseable en algunos casos.

6.4.3 Optimizaciones sobre la recolección de basura misma

Mientras que los recolectores de basura se pueden ver limitados por su relación con el compilador y sus optimizaciones, también es posible que el compilador ayude a hacer más eficiente al recolector. Por ejemplo, un compilador con optimización puede ser capaz de optimizar operaciones de barrera de lectura o escritura innecesarias o redundantes, detectando que un objeto en el heap puede ser asignado al stack sin ocasionar ningún problema.

En la mayoría de los algoritmos generacionales e incrementales, la barrera de escritura es sólo necesaria cuando una referencia está siendo almacenada, pero en tiempo de compilación puede ser difícil saber si un valor será o no una referencia. En lenguajes dinámicamente tipificados, las variables pueden tomar valores referencia o no, e incluso en lenguajes estáticamente tipificados variables referencia pueden obtener un valor nulo. Los compiladores pueden realizar análisis de flujo de datos que puede permitirles omitir la barrera de escritura para la mayoría de las operaciones de asignación que no involucren referencias.

El compilador también puede ayudar eliminando chequeos redundantes o marcar cuando una barrera de escritura checa la misma referencia o marca el mismo objeto en repetidas ocasiones. Al parecer actualmente ningún compilador existente hace esto. Para que eso sea posible, el optimizador debe ser capaz de suponer que ciertas cosas no son modificadas por el recolector en formas no obvias en tiempo de compilación. Por ejemplo, con un recolector de basura en cualquier momento y control de múltiples hilos, un hilo puede ser detenido arbitrariamente, y un ciclo de recolección puede ser ejecutado antes de que el hilo termine su ejecución. En tal sistema, hay pocas oportunidades para optimizar porque el recolector puede ser invocado entre cualesquiera dos instrucciones consecutivas. Sin embargo, con un sistema de puntos seguros, el optimizador puede ser capaz de realizar más optimizaciones a través de varias secuencias de instrucciones que se ejecuten atómicamente con respecto a la recolección de basura.

Ciertas formas restringidas de análisis de vida –para ambientes de ligados de variables locales solamente– pueden ser simples pero efectivas para evitar la necesidad de asignar al heap la mayoría de los ligados en lenguajes con cerraduras, como Scheme.

6.5 Manejo de almacenamiento libre

Los recolectores sin copiado deben manejar el problema de que el espacio libre puede estar esparcido a lo largo de la memoria, intercalado con objetos vivos. La forma tradicional de manejar tal situación es utilizar una o más listas libres, pero es posible adaptar cualquiera de las técnicas que se han creado para manejo explícito del heap.

El esquema más sencillo, utilizado en la mayoría de los primeros intérpretes de Lisp,

es soportar sólo un tamaño para los objetos alojados en el heap, y utilizar una sola lista para mantener a los objetos liberados. Cuando se detecta basura (v.gr., durante una fase de barrido o cuando las referencias llegan a cero), los objetos simplemente son integrados en una lista, utilizando uno de los campos de datos normales para mantener una referencia a la lista.

Cuando se generaliza este esquema para soportar objetos de tamaños variados, tenemos dos opciones posibles: mantener listas separadas para tamaño de objeto (o tamaño aproximado de objeto), o mantener una lista con los tamaños de los espacios liberados. Ambas técnicas son utilizadas en la práctica. Una estrategia intermedia es utilizar una sola estructura de datos, por ejemplo un árbol, y mantener en ella los espacios libres ordenados por tamaño (y/o por dirección). La mayoría de las técnicas con listas o estructuras híbridas, han sido ampliamente estudiadas en la literatura sobre manejo de almacenamiento, por lo que no las discutiremos aquí. Una excepción, es el manejo de memoria con mapas de bits, que es una técnica muy utilizada en recolectores de basura y la cual no se encuentra frecuentemente en la literatura.

El manejo de memoria con mapas de bits simplemente mantiene un mapa de bits que corresponden a unidades de memoria (típicamente palabras, o pares de palabras si los objetos están siempre alineados), los valores de los bits indican si una unidad está siendo usada o no. El mapa de bits es actualizado cuando los objetos son asignados a memoria o su espacio es recuperado. El mapa de bits puede ser revisado para construir una lista libre, o puede ser revisado cuando se asigna un objeto a memoria, de manera análoga a una lista libre con un algoritmo secuencial hasta encontrar un espacio lo suficientemente grande para almacenar al objeto.

6.6 Representación compacta de datos en el heap

Las representaciones en el heap son generalmente optimizadas para alcanzar una mayor velocidad, y no para reducir el espacio ocupado. En lenguajes dinámicamente tipificados, por ejemplo, la mayoría de los campos de los objetos son –usualmente– de tamaño uniforme, lo suficientemente grandes como para contener una referencia etiquetada. Las referencias, en cambio, se representan como direcciones virtuales de tamaño real en un espacio plano (sin segmentación). La mayoría de este espacio está "desperdiciado" en cierto sentido, ya que muchos valores no-referencias podrían ser representados con menos bits, y porque las referencias típicamente contienen muy poca información debido a tamaños de heap limitados y a la localidad de referencia.

Al menos dos mecanismos se han desarrollado para explotar las regularidades en las estructuras de datos y permitirles almacenar en una forma más compacta (la mayor parte del tiempo), y ser expandidas por demanda; esto es, cuando alguna operación se lleva a cabo con ellas. Uno de estos mecanismos es llamado *cdr coding*, que es específico a celdas de listas, tales como las celdas cons de Lisp. El otro es conocido como paginación comprimida (*compressed paging*), que es un mecanismo de propósito más general que opera sobre las páginas virtuales. Ambos mecanismos son invisibles al nivel del lenguaje.

Cdr-coding fue utilizado en muchos de los primeros sistemas Lisp, en los cuales la memoria de acceso aleatoria (RAM) era muy cara. Desafortunadamente, este meca-

nismo resulta muy caro en tiempo de CPU, ya que las representaciones cambiantes de listas complican las operaciones comunes sobre una lista. Operaciones tales como CDR requieren instrucciones extra para checar el tipo de celda que se está recorriendo –¿es una celda cons normal, o es una celda comprimida?.

La representación comprimida de una lista en un sistema de cdr codificado es realmente un arreglo de objetos que corresponden a los objetos en la lista. El sistema cdr-coding trabaja colaborando con el recolector de basura, linealizando listas y empaquetando objetos consecutivos en arreglos que contienen los valores CAR (los objetos mismos de la lista); los valores CDR –las referencias que mantienen ligada la lista– se omiten. Para que esto funcione, es necesario guardar un bit en alguna parte (v.gr. un bit extra en la etiqueta del campo que contiene el valor de CAR) diciendo que el valor del CDR es *implícitamente* una referencia al siguiente valor en la memoria. Con este esquema, las actualizaciones destructivas al CDR se convierten en operaciones muy costosas, ya que requieren no sólo la actualización de una referencia, sino el movimiento de –potencialmente– buena parte del arreglo que contiene a la lista.

Este esquema sólo vale la pena cuando se cuenta con hardware especial o con rutinas en microcódigo, por ejemplo en las máquinas Lisp. En los procesadores modernos de propósito general, el trabajo necesario no es equiparable con los ahorros alcanzables. Groso modo, en un sistema Lisp aproximadamente la mitad de los datos son celdas cons, por lo que comprimirlas de dos palabras a una sola nos ahorra en el mejor de los casos 25% del espacio.

Otra alternativa (más reciente) es la *paginación comprimida*, que es una técnica para reducir los requerimientos de memoria en hardware convencional. La idea básica es dedicar una fracción de la memoria principal para guardar páginas en un formato comprimido, de forma tal que una página de memoria principal puede contener la información de varias páginas de memoria virtual. El hardware normal para protección de acceso a la memoria virtual se usa para detectar referencias a páginas comprimidas, y las redirecciona hacia una rutina que las descomprime. Una vez tocada, una página se mantiene en formato normal (no-comprimido) por un cierto tiempo, para que el programa pueda operar con la información contenida en ella, sin la sobrecarga que representa la rutina de descompresión. Después de que una página no ha sido tocada en un cierto tiempo, se re-comprime y se protege contra accesos. La rutina de compresión no está limitada a comprimir campos CDR –puede comprimir otro tipo de referencias, y datos no-referencia (tales como enteros y cadenas de caracteres, o código ejecutable).

La paginación comprimida, en efecto, añade otro nivel a la jerarquía de memoria, intermedia entre la memoria principal RAM y el disco. Esto no sólo puede decrementar el requerimiento de RAM, sino también mejorar la velocidad al reducir el número de faltas de página. Utilizando algoritmos de compresión sencillos y rápidos (pero eficientes), los datos en el heap pueden ser comprimidos por un factor de dos a cuatro –en mucho menos tiempo de lo que tomaría hacer una búsqueda en el disco, para subir una página a memoria principal (incluso en un procesador relativamente lento). Conforme la velocidad de los procesadores aumenta, la paginación comprimida se vuelva cada vez más atractiva.

Capítulo 7

Características del lenguaje relacionadas a la recolección de basura

El uso principal de la recolección de basura es soportar la simple abstracción de que una cantidad infinita y uniforme de memoria se encuentra disponible, por lo que se pueden crear objetos a voluntad y –conceptualmente– vivirán por siempre. Sin embargo, algunas ocasiones es deseable tener referencias que no impidan que los objetos referidos sean recuperados, o disparar rutinas especiales cuando un objeto es recuperado. También es deseable tener más de un heap, y que los objetos asignados a distintos heaps sean tratados de manera distinta.

7.1 Referencias débiles

Una extensión sencilla a la abstracción de la recolección de basura, es permitir que los programas mantengan referencias a objetos *sin* que esas referencias eviten que los objetos sean recolectados. Las referencias que no impiden que un objeto sea recuperado, son conocidas como *referencias débiles*, y son útiles en una variedad de situaciones. Una aplicación común es el mantenimiento de tablas que hacen posible la enumeración de todos los objetos de una cierta clase. Por ejemplo, podemos tener una tabla de todos los objetos archivo en un sistema, de forma que sus buffers pueden ser vaciados periódicamente para ser tolerantes a fallas. Otra aplicación común es el mantenimiento de colecciones de información auxiliar acerca de objetos, donde la información por si misma es inútil y no debe mantener vivos a los objetos; ejemplos de esto incluyen a las tablas de propiedades y documentación de objetos –la utilidad de la descripción depende del objeto descrito y no al revés.

Las referencias débiles son típicamente implementadas mediante el uso de una estructura de datos especial, conocida por el recolector de basura, que guarda la localización de campos de referencias débiles. El recolector de basura recorre primero todas las demás referencias en el sistema, para determinar cuáles objetos son alcanzables vía referencias normales. A continuación recorre las referencias débiles, pero si sus referentes (los referidos por las referencias débiles) han sido alcanzados por rutas normales, las referencias débiles se tratan de manera usual (en un recolector con copiado, por ejemplo, son actualizados para

reflejar la nueva localización del objeto). Sin embargo, si sus referentes no han sido alcanzados, las referencias débiles son tratadas de manera especial, usualmente son reemplazados por un valor no-referencia (tal como `null`) para señalar que sus referentes ya no existen.

7.2 Finalización

Muy relacionado con el concepto de referencias débiles está el concepto de *finalización*, i.e. las acciones que se realizan cuando un objeto es recuperado. Esto es particularmente común cuando un objeto maneja un recurso además de memoria del heap, tal como un archivo o una conexión de red. Por ejemplo, puede ser importante cerrar el archivo cuando el objeto correspondiente en el heap es recuperado. En el ejemplo de anotaciones y documentación, es deseable borrar la descripción de un objeto una vez que éste ha sido recuperado. La finalización, por lo tanto, generaliza al recolector de basura, ya que otros recursos son tratados exactamente igual que la memoria en el heap, y con una estructura similar de programa. Esto hace posible escribir código reutilizable más general, en lugar de tener que tratar con ciertos tipos de objetos de manera distinta que los objetos "normales". Por ejemplo, consideremos una rutina que itera sobre una lista, aplicando una función arbitraria a cada elemento de la lista. Si los descriptores de archivo son recolectados, la misma rutina de iteración se puede utilizar para una lista de descriptores de archivos que para una lista de objetos en el heap. Si la lista se convierte en inalcanzable, el recolector de basura recuperará los descriptores de archivo junto con la estructura misma de la lista.

La finalización se implementa, típicamente, marcando objetos como finalizables de alguna manera y registrándolos en una estructura de datos muy parecida a la de las referencias débiles (de hecho, pueden utilizar la misma estructura), en lugar de simplemente destruirlos si los objetos no son alcanzados por el recorrido principal. Las referencias se graban para darles un tratamiento especial después de que la recolección termine. Una vez que la recolección termina y el heap se encuentra en un estado consistente, se invoca la operación de finalización de los objetos referidos.

La finalización es útil en un variedad de circunstancias, pero debe usarse con cuidado, ya que la finalización ocurre asíncronamente —i.e. cuando el recolector nota que los objetos son inalcanzables y hace algo al respecto— y esto puede conducir a condiciones de concurso y otro tipo de errores sutiles.

7.3 Múltiples heaps manejados de manera distinta

En algunos sistemas, por conveniencia, se usan dos heaps: uno recolectado y otro explícitamente manejado, para permitir un control más preciso sobre el uso de la memoria. Algunas implementaciones de los lenguajes *Modula-3* y una versión extendida de *C++* son ejemplos de este tipo de sistemas, donde un heap recolectado coexiste con uno explícitamente manejado. Así, mientras un recolector de basura maneja gran parte de la memoria, el programador puede manejar explícitamente la asignación y liberación de algunos objetos para alcanzar un mejor funcionamiento o al menos un sistema más predecible.

En otros sistemas, dónde contamos con memorias distribuidas compartidas o sistemas con grandes memorias persistentes, es deseable tener varios heaps con diferentes políti-

cas de distribución (v.gr., compartidas Vs. no compartidas), con distintos privilegios de acceso y distinto manejo de los recursos. Por supuesto, existen implementaciones de lenguajes para estos sistemas con tales características, pero la documentación es insuficiente en la mayoría de los casos y la terminología no está estandarizada, por lo que es común que estos heaps sean llamados: "heaps", "zonas", "áreas", "arenas", "segmentos", "piscinas", "regiones", etc.

Capítulo 8

El recolector conservador por excelencia

En la práctica, muchos sistemas especializados y varias implementaciones de lenguajes de programación utilizan recolectores de basura implementados *ad hoc* para tales fines. Sin embargo, en la última década, un recolector de basura conservador, en el más estricto sentido de la palabra, se ha consolidado como el recolector genérico más usado. Nos referimos al recolector de basura[1] escrito por *H. Boehm, A. Demers* y *M. Weiser*, aunque se le atribuye esencialmente a *Hans-J. Boehm.* Este recolector de basura/asignador es de propósito general y los algoritmos usados en él se describen con detalle en [Boehm, 1993], [Boehm et al., 1991] y [Boehm and Weiser, 1988]. A continuación haré una descripción general del recolector de basura.

El recolector de basura utiliza un algoritmo modificado de marcado y barrido. Conceptualmente opera en cuatro fases:

1. La etapa de preparación inicializa todos los bits de marcado, indicando que todos los objetos son potencialmente inalcanzables.

2. La fase de marcado marca a todos los objetos que pueden ser alcanzados vía cadenas de referencias a partir de las variables. Normalmente el recolector no tiene información real acerca de la localización de variables referencia en el heap, así que potencialmente todas las áreas de datos estáticos, stacks y registros pueden contener referencias Cualquier patrón de bits que represente direcciones dentro de algún objeto manejado por el recolector es visto como una referencia (nótese que por este motivo, este recolector es conocido como "el recolector conservador", ya que lleva al límite el conservadurismo en la selección del conjunto raíz). A menos que el programa cliente ponga a disposición del recolector información acerca de la forma de los objetos en el heap, todos los objetos que son alcanzables en el heap a partir de las variables también son revisados de la misma forma.

3. La fase de barrido revisa el heap buscando objetos inalcanzables (y por lo tanto no

[1] Para mayor información sobre este recolector de basura puede visitar la siguiente página de WEB *http://reality sgi com/boehm/gc html*

marcados), y los añade a la lista libre para su reutilización. Esto en realidad no es una fase separada; se hace por demanda durante la asignación de memoria, si al realizarse la asignación se encuentra una lista libre. Por tal motivo es poco probable que la fase de barrido "toque" una página que de cualquier forma no estuviese a punto de ser tocada por la asignación de memoria.

4. La fase de finalización, en la cual los objetos no alcanzables que han sido registrados para finalización, son formados en una cola para tal fin fuera del recolector.

El recolector de basura incluye su propio asignador de memoria. El asignador obtiene su memoria del sistema operativo de forma independiente de la plataforma. En `Unix`, por ejemplo, utiliza `malloc`, `sbrk`, o bien `mmap`.

El asignador puede asignar objetos de distintas clases. Cada clase es manejada de manera distinta por ciertas partes del recolector de basura. Algunas clases son revisadas en busca de referencias, otras no. Algunas pueden tener descriptores de tipo por objeto que determinan las posiciones de las referencias, e incluso algunas clases corresponden a una arquitectura de objeto específica. En la siguiente sección mencionaré algunas de las instrucciones que el asignador proporciona para asignar estas clases distintas a memoria.

El recolector no requiere que las referencias estén etiquetadas, por lo que no intenta asegurar que todo el espacio de almacenamiento inaccesible sea recuperado. Esto no representa un problema grave para la mayoría de las aplicaciones que usan el recolector, ya que la cantidad de memoria que —típicamente— se mantiene (i.e. que no es recuperada y que sí es basura) es menor que las fugas de memoria que introducen en sus programas los programadores promedio. Nuevamente, estas conclusiones, al igual que muchas otras aseveraciones acerca de las invariantes en recolección de basura, descansan sobre muestreos estadísticos sobre poblaciones muy pequeñas y por tanto no deben ser tomados como verdades absolutas.

El recolector utiliza un asignador de dos niveles. Un bloque grande se define de tal forma que sea uno más que la mitad de `HBLKSIZE`, que es una potencia de 2, típicamente del orden del tamaño de una página. Entonces los bloques de gran tamaño se redondean hacia arriba hacia el siguiente múltiplo de `HBLKSIZE` y son asignados por `GC_allochblk`. Dicha función utiliza un algoritmo de *next fit*, i.e. *first fit* con una referencia rotativa. La implementación checa si el bloque inmediato adyacente ofrece la mejor opción para almacenar, lo cual le da mejores características de fragmentación. Esto ha sido causa de mucha controversia, e incluso los autores están seguros de que sería mejor utilizar el algoritmo de asignación *best fit*.

Se asignan bloques pequeños de tamaño `HBLKSIZE`. En cada bloque sólo pueden ser alojados objetos del mismo tamaño. El recolector entonces, mantiene listas libres separadas para cada tamaño y tipo de objeto. Dado que hay una gran variabilidad en los tamaños de los objetos que pueden presentarse en un programa, no resulta conveniente asignar bloques para todos y cada uno de los tamaños presentes, sino que asigna sólo unos cuantos y cuando tiene que asignar a algún objeto cuyo tamaño no tiene un bloque asignado, rellena el espacio del objeto al tamaño inmediato superior para el que si haya bloque, y lo aloja en el bloque correspondiente. Como todo esto se hace por demanda, para los primeros objetos asignados muy probablemente se adjudicarán bloques de su tamaño exacto, mientras que para los posteriores no.

El recolector no requiere que las referencias estén etiquetadas, por lo que no intenta asegurar que todo el espacio de almacenamiento inaccesible sea recuperado. Sin embargo, la experiencia de los autores (que se describe en las referencias anteriores y el archivo `README` que acompaña la distribución del mismo) es que dicho recolector es más exitoso para recuperar memoria no utilizada que la mayoría de los programas en `C` que utilizan recuperación explícita de memoria.

8.1 Propuesta de modificación al recolector conservador de Boehm

Como mencioné en la sección anterior, muchas aplicaciones utilizan el recolector conservador de Boehm. Una de tales aplicaciones es *PLT Scheme*[2], que es una implementación del lenguaje de programación Scheme, con muchas extensiones al mismo. Hablar de Scheme escapa, por razones de espacio, del alcance de este trabajo, por lo que mencionaré a continuación algunas de las extensiones más importantes que *PLT Scheme* ofrece:

- Un sistema de clases y objetos.
- Un sistema de unidades para compilar por separado componentes de un programa.
- Un sistema de excepciones que es usado para todas las primitivas de error.
- Hilos de control (*threads*) y varios espacios de nombres para variables globales.
- Un conjunto de bibliotecas escritas en `C++` para generar interfaces gráficas. En las versiones iniciales de PLT Scheme que incluían estas bibliotecas, se distribuían dos implementaciones de Scheme separadas: *MzScheme* y *MrEd*, la segunda era la única que incluía estas bibliotecas.

Por el tipo de extensiones que ofrece *PLT Scheme*, sobre todo en las versiones que incluyen las bibliotecas de gráficos, los miembros del grupo PLT se han enfrentado a un buen número de problemas con el uso del recolector de Boehm. Sobre todo por la situación que se presenta entre objetos explícitamente manejados, como todo lo que se asigna desde las bibliotecas de gráficos y la finalización de objetos, como cerrar puertos, objetos de entrada y salida, y cosas similares en la presencia de hilos de control.

El resultado neto de estos problemas es que cuando *PLT Scheme* ejecuta un programa grande, la cantidad de memoria no recuperada por el recolector es cada vez mayor. Por lo tanto se vió la necesidad de modificar el recolector de basura de Boehm para hacerlo menos conservador y poder así recuperar más memoria, aún en presencia de las extensiones arriba mencionadas.

---

[2]El grupo PLT (*Programming Languages Team*), fue creado por *Matthias Felleisen* en la Universidad de Rice, en Houston. Actualmente la central del grupo se ha mudado a la Universidad de NorthEastern, en Boston y tiene sucursales en la Universidad de Brown, y en la Universidad de Utah, todas en ellas en EE UU. Para mayor información sobre *PLT Scheme* y materiales relacionados, por favor visite la siguiente página de WEB:

http://www.ccs.neu.edu/scheme/

Después de nosotros discutir con Boehm sobre el particular, surgieron varias ideas para modificar el recolector y mantener explícitamente una estructura con las referencias que conformarían el conjunto raíz de la siguiente recolección. Estas ideas se exponen en la siguiente sección, junto con los problemas de tales ideas.

8.2 Qué modificar para hacerlo explícito y menos conservador

El recolector de basura de Boehm, como mencionamos al inicio de este capítulo, recorre todas las áreas de datos estáticos, los registros y el stack, o los stacks en la presencia de hilos de control. ¡El reto entonces es identificar eficientemente todas las raíces y no olvidarlas!. Es fácil identificar las partes del código en el recolector que deben ser modificadas, lo que es difícil es identificar una estructura apropiada para no crear un cuello de botella en el mantenimiento de tal estructura de datos.

Más aún, dado que *PLT Scheme* sí usa el soporte para hilos de control que provee el recolector, las modificaciones para mantener el conjunto raíz crecen, ya que el recolector de Boehm, localiza "aproximadamente" los stacks de cada hilo, lo cual implicaría más operaciones por cada referencia que se encuentre en el stack, para agregar precisamente la información del stack correspondiente a la estructura.

La rutina que marca las referencias e inicia el conjunto raíz en el recolector de Boehm, luce así:

```
void GC_push_roots(all, cold_gc_frame)
GC_bool all;
ptr_t cold_gc_frame;
{
    register int i;
#   ifdef USE_GENERIC_PUSH_REGS
        GC_generic_push_regs(cold_gc_frame);
#   else
        GC_push_regs();
#   endif
#      if (defined(DYNAMIC_LOADING) || defined(MSWIN32) || defined(PCR)) \
           && !defined(SRC_M3)
       GC_remove_tmp_roots();
       if (!GC_use_registered_statics)
         GC_register_dynamic_libraries();
#      endif
       for (i = 0; i < n_root_sets; i++) {
         GC_push_conditional_with_exclusions(
                             GC_static_roots[i].r_start,
                             GC_static_roots[i].r_end, all);
       }
#   if !defined(USE_GENERIC_PUSH_REGS)
       GC_push_current_stack(cold_gc_frame);
```

```
#   endif
    if (GC_push_other_roots != 0) (*GC_push_other_roots)();
    if (GC_push_last_roots != 0) (*GC_push_last_roots)();
}
```

Procedamos a analizar el código de esta función, que corresponde a la "forma" en que funciona el recolector:

- Primero los parámetros de la función: Si `all` es falso, entonces sólo mete al conjunto raíz los valores modificados (esto es útil cuando se está usando la opción de recolección incremental). El segundo parámetro, `cold_gc_frame`, es una dirección dentro del marco (*frame*) del recolector de basura y que se mantiene viva hasta que la recolección concluye. Nuevamente, si la recolección es incremental, esta dirección permanece viva en memoria hasta que la última activación del recolector termina.

- La siguiente sección se encarga de meter el contenido de todos los registros al conjunto raíz:

```
    register int i;
#   ifdef USE_GENERIC_PUSH_REGS
        GC_generic_push_regs(cold_gc_frame);
#   else
        GC_push_regs();
#   endif
```

- La siguiente sección se encarga de meter todas las secciones estáticas de datos. Y es importante que esto se haga muy temprano durante la recolección, porque durante esta fase no se hace ninguna especie de chequeo para ver si existe un sobre flujo en el stack de marcas.

```
#     if (defined(DYNAMIC_LOADING) || defined(MSWIN32) || defined(PCR)) \
          && !defined(SRC_M3)
      GC_remove_tmp_roots();
      if (!GC_use_registered_statics)
        GC_register_dynamic_libraries();
#     endif
      for (i = 0; i < n_root_sets; i++) {
        GC_push_conditional_with_exclusions(
                             GC_static_roots[i].r_start,
                             GC_static_roots[i].r_end, all);
      }
```

- Finalmente llegamos a las partes que nos interesan:

```
#    if !defined(USE_GENERIC_PUSH_REGS)
         GC_push_current_stack(cold_gc_frame);
#    endif
```

Si el sistema que utiliza el recolector no ofrece soporte para hilos de control, el trabajo de modificación de recolección debe centrarse exclusivamente en la función `GC_push_current_stack`, desde donde bastaría recorrer el stack buscando las referencias que debemos meter al conjunto raíz.

En la presencia de hilos de control, al menos en teoría, bastaría con cambiar la siguiente línea:

```
if (GC_push_other_roots != 0) (*GC_push_other_roots)();
```

por un apuntador a una rutina que hiciera lo correcto. Es decir, que detectara los stacks en cada hilo de control y los recorriera como lo haría `GC_push_current_stack`.

En cualquier caso, ya sea que la aplicación soporte hilos de control o no, el macro `GC_PUSH_ONE_STACK` debe ser llamado para cada referencia que se encuentre en el stack, es decir debemos meter al stack de marcas esa referencia. Es importante aclarar esto, porque dependiendo de la aplicación, pueden existir variables locales que sobrevivan a la activación de la función que las contiene. En muchos lenguajes de programación esto es común si las funciones son de primera clase, si soporta continuaciones (*continuations*), o cerraduras (*closures*) y en un buen número de contextos más. Nuestro sistema ejemplo, *PLT Scheme*, sí lo hace. Por este motivo, aún manteniendo explícitamente el conjunto raíz, el recolector sigue siendo conservador, aunque en menor escala.

Identificadas las secciones de código que deben ser modificadas, el problema se dirige entonces hacia la estructura a utilizar. Debe ser evidente que la selección de la estructura debe contemplar que muchos de los datos que entran al stack en un determinado momento, por ejemplo, durante la activación de una función: parámetros locales, variables locales, etc., lo hacen por periodos de tiempo muy pequeños, en el orden de nanosegundos. Por tanto, las operación de inserción y localización y borrado de un elemento en la estructura deben ser muy rápidas.

La mayoría de las opciones obvias: listas, arreglos, árboles, tablas de hash, etc. son candidatos que se han usado en muchos recolectores de basura y dado el esquema del recolector de Boehm, podrían funcionar aquí. Sin embargo, en términos de espacio y operaciones de máquina todas ellas serían muy costosas. Por tanto, la opción a seguir sería mantener tablas de rangos (*PC range tables*) que describan de alguna forma el contorno del marco.

La gran pregunta que quedaría por resolver, y para la cual no logramos visualizar ninguna alternativa durante nuestras discusiones sobre el tema, es ¿cómo localizar temporales del compilador en el stack y raíces en cualquier función (en C) que sea parte del ambiente de ejecución (*run time*)? Depurar el recolector una vez realizados los cambios aquí propuestos puede ser una tarea muy difícil. Encontrar raíces en el resultado de las rutinas del ambiente de ejecución es muy importante, ya que muchos lenguajes manejan cadenas, arreglos y todo tipo de estructuras, con rutinas externas al compilador y sin embargo, esos objetos en memoria deben quedar bajo la vigilancia del recolector.

8.3 Conclusiones

Suponiendo que el esquema de modificaciones propuesto en la sección anterior se llevara a cabo, la ganancia *en espacio* para el promedio de las aplicaciones sería notable. Sobre todo en aplicaciones como *PLT Scheme*, debido al tipo de extensiones que no se llevan bien con el esquema actual del recolector. Sin embargo, la ganancia *en tiempo* sería despreciable, suponiendo una estructura de datos para almacenar los datos muy eficiente, o negativa en la mayoría de los casos. De hecho, varios puntos a favor de la localización conservadora de referencias (como la que hace el recolector de Boehm) ya fueron discutidos con detalle en la sección 6.2, página 83, se recomienda leer nuevamente esa sección, porque aquellos comentarios están basados directamente en el recolector de basura que aquí usamos como caso de estudio.

Lo más difícil de hacer, en general, para recolectores de basura es probarlos, ya que eso implica un proceso muy largo y costoso. Los principales clientes de un recolector de basura son implementaciones de lenguajes de programación. Sin embargo, las implementaciones mismas difícilmente ponen a prueba el recolector, es decir, difícilmente una implementación de un lenguaje de programación *usual* en la actualidad y corriendo en una máquina de escritorio promedio, provocaría que el recolector de basura se activara durante la compilación de un programa cliente. Y en todo caso, probablemente el compilador mismo *no* utiliza al recolector de basura, sino que maneja explícitamente su memoria.

Así, es fácil ver que los verdaderos programas para probar un recolector de basura son las aplicaciones escritas en esos lenguajes de programación, cuyas implementaciones usan el recolector de Boehm. Desgraciadamente, no hay, ni remotamente, una clasificación lo suficientemente amplia para cubrir todo tipo de aplicaciones que los programadores en un lenguaje dado pueden escribir. Por este motivo, es recomendable lanzar implementaciones paralelas de las implementaciones de lenguajes de programación que usen el recolector modificado, y después organizar una campaña donde los usuarios de tales lenguajes de programación cooperaran probando sus aplicaciones (ya funcionales) con la versión paralela (que usa el recolector modificado). Después de recabar toda la información obtenida, es posible comparar la versión modificada (menos conservadora) con la versión conservadora de Boehm. Sobra aclarar, que esto es una tarea que sobrepasa y por mucho, el alcance y posibilidades de un trabajo como el presente.

Es interesante considerar que *PLT Scheme*, el sistema que originalmente fue la motivación para investigar los detalles del recolector conservador de Boehm, usó este recolector mucho tiempo, para todas sus versiones, tanto de *MzScheme* como de *MrEd*, hasta hace poco (mayo de 2001). Hasta ese punto, el recolector de Boehm, fue modificado constantemente por Matthew Flatt, uno de los miembros del grupo PLT y autor del núcleo de las implementaciones de Scheme del grupo, para tratar de adaptarlo de mejor forma a los detalles de *MrEd*. Sin embargo, en las versiones de *PLT Scheme*, que al día de hoy se consideran beta (las versiones 199/200). *PLT Scheme* se distribuye con un nuevo recolector explícito de basura (i.e. no conservador).

El diseño del nuevo recolector de basura para *PLT Scheme* soporta recolección precisa tanto con recolectores de barrido como de copiado. Y al igual que el de Boehm, también soporta recolección generacional e incremental, siempre y cuando el hardware ofrezca una implementación de protección de memoria (para detectar escrituras en objetos viejos.

etc.), véa las secciones: 2.2.4 y 4.4.7 y en general los capítulos 3 y 4). El por qué no se intentó modificar el recolector de Boehm, para hacerlo menos conservador, en lugar de diseñar un recolector nuevo partiendo de cero, no lo sabemos , pero muy seguramente llegaron a conclusiones similares a las que se expresan en esta sección.

Bibliografía

[Appel, 1989] Appel, A. W. (1989). Simple generational garbage collection and fasta allocation. *Software Practice and Experience*, 19(2):171–183.

[Boehm, 1993] Boehm, H. (1993). Space efficient conservative garbage collection. *ACM SIGPLAN*, 28(6):197–206.

[Boehm et al., 1991] Boehm, H., Demers, A., and Shenker, S. (1991). Mostly parallel garbage collection. *ACM SIGPLAN*, 26(6):157–164.

[Boehm and Weiser, 1988] Boehm, H. and Weiser, M. (1988). Garbage collection in an uncooperative environment. *Software Practice & Experience*, pages 807–820.

[Cheney, 1970] Cheney, C. J. (1970). A nonrecursive list compacting algorithm. *Communications of the ACM*, 13(11):677–678.

[Dijkstra et al., 1978] Dijkstra, E. W., Lamport, L., Martin, A. J., Csholten, C. S., and Steffens, E. M. (1978). On-the-fly garbage collection: An exercise in cooperation. *Communications of the ACM*, 21(11):966–975.

[Findler et al., 1997] Findler, R. B., Flanagan, C., Flatt, M., Krishnamurthi, S., and Felleisen, M. (1997). Drscheme: A pedagogic programming environment of scheme. Technical report, Rice University, 6100 Main Street. Houston, Tx. 77005-1892.

[Flatt, 1998] Flatt, M. (1998). *PLT MzScheme: Language Manual.* Rice University, Department of Computer Science, 6100 Main street. Houston, Tx. 77005-1892. Version 52.

[Guy Lewis, 1975] Guy Lewis, Jr., S. (1975). Multiprocessing compatifying garbage collection. *Communications of the ACM*, 18(9):155–180.

[Hayes, 1991] Hayes, B., editor (1991). *Using key object opportunism to collect old objects*, volume 26. OOPSLA '91. Conference proceedings on Object-oriented programming systems, languages, and applications, ACM SIGPLAN Notices. Pages 33-46.

[Knuth, 1969] Knuth, D. E. (1969) *The Art of Computer Programming*, volume 1 Addison-Wesley, Reading, Massachusetts.

[Maccabe, 1993] Maccabe, A. B (1993) *Computer Systems Architecture, Organization and Programming*. Richard D. Irwin, Inc.

[Minsky, 1963] Minsky, M. (1963). A lisp garbage collector using serial secondary storage. Technical report, A.I. Memo 58, Massachusetts Institute of Technology, Project MAC, Cambridge, Massachussetts.

[Morrisett et al., 1996] Morrisett, G., Felleisen, M., and Harper, R. (1996). Abstract models of memory management. Technical report, Carnegie Mellon, Rice University and Advanced Research Projects Agency (ARPA), CSTO.

[Patterson and Hennessy, 1996] Patterson, D. A. and Hennessy, J. L. (1996). *Computer Architecture A Quantitative Approach.* Morgan Kauffman, second edition.

[Sobalvarro, 1988] Sobalvarro, P. G. (1988). *A lifet e-based garbage collector for LISP systems on general purpose computers.* PhD thesis, MIT. Bachelor's thesis.

[Ungar and Jackson, 1992] Ungar, D. and Jackson, F. (1992). An adaptive tenuring policy for generation scavengers. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 14(1).

[Weiser et al., 1989] Weiser, M., Demers, A., and Hauser, C. (1989). The portable common runtime approach to interoperability. In *Proceedings of the Twelfth ACM symposium on Operating systems principles*, pages 114 – 122. ACM Press.

[Wilson, 1992] Wilson, P. R. (1992). Uniprocessor garbage collection techniques. *ACM Computing Surveys.*

[Wilson and Moher, 1989a] Wilson, P. R. and Moher, T. G. (1989a). A "card-marking" scheme for controlling intergenerational references in generation-based garbage collection on stock hardware. *ACM SIGPLAN Notices*, 24(5):87–92.

[Wilson and Moher, 1989b] Wilson, P. R. and Moher, T. G. (1989b). Design of the opportunistic garbage collector. In *Conference on Object Oriented Programming Systems, Languages and Applications*, pages 23–35. ACM Press.

[Yuasa, 1990] Yuasa, T. (1990). Real-time garbage collection on general-purpose machines. *Journal of Systems and Software*, 11(3):181–198.

[Zorn and Grunwald, 1994] Zorn, B. and Grunwald, D. (1994). Evaluating models of memory allocation. *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, 4(1):107–131.

Índice

A

Abrahamson y Patel 35
Appel, Andrew W. 65
apuntador .... *véase* referencia

B

búsqueda en amplitud. .... *véase* BFS
búsqueda en profundidad ..... *véase* DFS
Baker, Henry G. 27, 43, 47, 78
barreras de lectura y escritura ...... *véase* técnicas, incremental
basura 13
    detección de 13
    flotante 31, 32
    recolección de 9, 13
        consistencia relajada en 32
        fases de la 11
        técnicas para. . . . *véase* técnicas
BFS 22, 33, 77
bit 62
bits de suciedad .... *véase* dirty bits
Boehm, Hans-J. 95
bolsa 65
Brooks, Frederick P. 39
byte 1

C

cache 4
Cheney, C. J. 33
ciclo de recolección 55, 56, 58, 65
compilador 5. 61
    optimizaciones 84, 86
compiladores
    restricciones 84
condiciones de concurso 92
conjunto raíz 13, 17, 69, 82
    problemas .... *véase* técnicas, generacional, problemas con
    tiempo de recorrido del 70
conteo de referencias
    variantes al 18
CPU .... *véase* procesador

D

Demers, A. 95
DFS 33, 77
Dijkstra, Edsger W. 36, 42
dirty bits 4, 45, 56, 63, 83

E

espacio de memoria dinámica . *véase* heap
Explorer ... *véase* máquinas, lisp, explorer

F

Felleisen, Matthias 97
finalización 92
Flatt, Matthew 101

G

garbage .... *véase* basura

H

hardare
    recursos de 92
hardware
    interrupciones de 87
heap . . 9, 14, 17, 20, 48, 53, 56, 61, 64, 70, 74, 85, 91
    cdr coding 89
    compressed paging 89
    representación compacta 89
    semi-espacios del 22
heaps múltiples 92

I

invariante tricolor .... *véase* técnicas, incremental

L

lenguaje de alto nivel 5
lenguajes de programación
  Ada 85
  C 29, 65, 83
  C++ 84
  C++ extendido 92
  Common Lisp 83
    Kyoto Common Lisp 83
  dinámicamente tipificados 14
  estáticamente tipificados 14, 66
  fuertemente tipificados 66
  Lisp 14, 39, 61, 66, 90
  ML 65, 79
  Modula-3 92
  Scheme 81, 97
    PLT 97
  Smalltalk 14, 62, 77
    ParcPlace Smalltalk-80 61
liga .... *véase* referencia
lista de almacenamiento 89
listas de almacenamiento 65
  con los tamaños 89
  por tamaño 89
LMI .... *véase* máquinas, lisp

M

máquinas
  lisp 63, 79
    explorer 78
    recolectores para 71
  orientadas a objetos 78
métricas 53
mapa de bits 64, 89
marcado de cartas 64
marcado tricolor 33
memoria 1
  área global de datos 9
  área para registros de activación (stack) 8
  cache 2
    política de reemplazo 76
  compactación 20
  dinámica (heap) 9
  disponible 53
  distribuida 92
  fragmentación 19, 20, 47, 60, 75
  jerarquía de 1, 90
  principal 2
  programa asignador de espacio en
    best fit 96
    de dos niveles 96
    first fit 96
    malloc 96
    mmap 96
    next fit 96
    sbrk 96
  programa asinador de espacio en .. 10
  RAM 76, 89
  recuperación explícita 10, 74
  secundaria 4
  velocidad de acceso 4
  virtual 3, 4, 20, 45, 46, 63, 90
    BiBOP 82
microcódigo 61, 90
microkernel 79
Moss, J. Eliot B. 65
multiconjuntos 65
MUSHROOM *véase* máquinas, orientadas a objetos
mutante 31–33, 35, 48, 61, 65, 78

N

números de punto flotante 69
  problema con 69

O

objeto 10, 18, 67
  alcanzable 13
  edad de un 53
  encabezado de 81, 89
  finalizable 92
  tipo de 14
    chequeo de 82
  vivo 10, 35
operación 15

alineamiento de 81
aritmética 82
atómica 48
load 81
store 81

P

paginación 29, 46, 71, 78
comprimida 90
pila .... *véase* stack
principio de localidad 1, 56, 67
efecto indirecto 75
espacial 74
reasignación 74
temporal 74
principio de localidad de referencia . 2, 20, 73
variedades de 73
procesador 1, 2, 5, 46, 53, 90
programa cliente .... *véase* mutante
programación
estilo de 17, 66, 73

R

recolección
en cualquier momento 86
en puntos seguros 86
hilos de control 86
optimización de 88
recolectores
conservadurismo en 13, 29, 30, 51
grados de 30
de copiado 31
de marcado y barrido 31
diferencia entre 31
mayoritariamente con copiado 83
Barlett 83
Detlefs 83
recuperación automatica de almacenamiento .... *véase* basura
referencia 13, 56
débil 91, 92
etiquetada 14, 68, 81, 96, 97
intergeneracional 56, 57, 63
localización conservadora . 83, 84, 101
problemas 84
restricciones 83
registros 13, 48, 70
partición de 87
reloj de asignación de memoria 49
ritmo de recorrido 49

S

sistema operativo 63
kernel 63, 79
sistemas existentes
cómo añadir recolección de basura a
conjunto especial de tipos 85
referencias inteligentes 85
sistemas reflexivos 85
cómo añadir recolección de basura a 83, 85
sistemas paralelos 32
coherencia 32
Sobalvarro, Patrick 64
stack 8, 9, 47, 48, 74, 83
de activación 82
Steele Jr., Guy L. 37, 42
swap 4, 76
Symbolics .... *véase* máquinas, lisp

T

técnicas 14
actualización incremental 34
complejidad de las 27
conteo de referencias 15
problemas 16
ventajas 15
conteo de referencias pospuesto ... 17
copiado 15, 21, 47, 50, 53
Cheney 22, 27
eficiencia en 25
incremental 32
localización 21
semi-espacios 21
copiado de replicación 41
consistencia 41
efímera 63
eficiencia 29

fotografía instantánea a la entrada 34
generacional 51, 54
calendarización de recolección . . . 57
casi 61
de tiempo real 70
Generation Scavenging 58
número de generaciones 55
organización del heap 57
política de avance 57
problemas con 69
referencias intergeneracionales . . . 57
trampas de la 67
híbridas 60
ideal 30
implícita 25
incremental 16, 31, 34
algoritmo de actualización 37
barrera de escritura . . 44, 47, 56, 70
barrera de lectura 34, 44, 61
coherencia 32
comparación 44
dificultad de 31
política de avance 68
incrementales
barrera de escritura 65
marcado 15
incremental 32
marcado y barrido 19, 95
etapa de barrido 95
etapa de finalización 96
etapa de marcado 95
etapa de preparación 95
problemas 19
marcado y compactación 20
ventaja 21
oportunista 59
radical 43
recorrido 50
sin copiado . . *véase* técnicas, recorrido
tiempo real 45
clases de 45
tabla de dispersión . . . *véase* tabla de hash
tabla de hash 65, 77
threads *véase* sistemas operativos, hilos de
control
TI . . . . *véase* máquinas, lisp

U

Ungar, David 59
Unidad Central de Procesamiento . . *véase* procesador

V

valores inmediatos 69
variables
de corta vida 17
de stack 17
en los registros 30
globales 13, 30, 48
locales 13, 17

W

Wang, Tomas 27
Weiser, M. 95
White, Jon L. 78
Wilson, Paul R. 59, 68, 76

X

Xerox PARC PCR 60

Y

Yuasa, T. 35

Z

Zorn, Benjamin 76