6
Implementação

Neste capítulo descrevemos algumas opções para a implementação de finalizadores e referências fracas, abordando do ponto de vista prático as principais questões e alternativas semânticas exploradas no Capítulo 5. Naturalmente a estratégia de implementação depende do tipo de coletor de lixo. Assim iremos considerar separadamente sistemas que utilizam contagem de referências e sistemas que utilizam rastreamento.

6.1
Coletores de Lixo Baseados em Contagem de Referências

Conforme já discutimos, sistemas que empregam contagem de referências conseguem identificar mudanças na conectividade de objetos de forma síncrona. Isso traz implicações importantes para a implementação tanto de finalizadores quanto de referências fracas.

6.1.1
Finalizadores

A implementação de finalizadores em coletores que utilizam contagem de referências envolve algumas sutilezas. Para invocar finalizadores de forma síncrona é necessário apenas modificar a rotina responsável por decrementar a contagem de referências e por desalocar objetos. Como ilustração, a definição de `decrementCount` apresentada na Seção 2.1 pode ser modificada da seguinte forma:

```
void decrementCount(){
  // verifica se o finalizador deve ser invocado
  if ((refCount == 1) && (finalizerEnabled())){
    // desabilita novas chamadas do finalizador
    disableFinalizer();
    //invoca o finalizador
    finalize();
```

```
  }
  // verifica se o objeto deve ser desalocado
  if (refCount == 1){
    // obtém todos os objetos diretamente referenciados
    // por este objeto
    Object child[] = getChilds();
    // decrementa a contagem de referências dos filhos
    // de forma recursiva
    for (int i = 0; i < child.length; i++)
      child[i].decrementCount();
    // desaloca a memória associada a este objeto
    free(this);
  } else
    synchronize(this){
      --refCount;
    }
}
```

A invocação do finalizador é efetuada imediatamente após a última referência para o objeto ser destruída (note que nesta implementação a contagem de referências é testada antes de ser decrementada). Como o finalizador retém uma referência para o próprio objeto, a contagem de referências permanece em um durante a sua execução. Caso isso não ocorresse, a criação e posterior destruição de referências para o objeto durante a finalização poderia fazer com que a respectiva contagem de referências atingisse um imediatamente antes da invocação de `decrementCount`, possivelmente causando a desalocação do objeto enquanto o mesmo ainda está sendo manipulado.

Como a finalização pode levar à ressurreição, a contagem de referências precisa ser testada novamente após a execução do finalizador. Só então, se não existir nenhuma referência para o objeto, ele é desalocado.

É fácil perceber que neste modelo de implementação a ordem de invocação de finalizadores obedece o sentido das referências entre objetos. Além disso todos os objetos referenciados por um objeto finalizável são implicitamente preservados.

Para executar finalizadores de forma assíncrona podemos usar a seguinte implementação:

```
void decrementCount(){
  synchronize(this){
    --refCount;
```

```
  }
  // verifica se o finalizador deve ser invocado
  if ((refCount == 0) && (finalizerEnabled())){
    // desabilita novas chamadas do finalizador
    disableFinalizer();
    // incrementa a contagem de referências por conta da referência
    // implícita passada para o finalizador
    ++refCount;
    // insere o objeto em uma fila de finalização e notifica o
    // thread responsável pela execução dos finalizadores
    GC.finalizationQueue.add(this);
    notify();
  // verifica se o objeto deve ser desalocado
  } else
    if (refCount == 0){
      // obtém todos os objetos diretamente referenciados
      // por este objeto
      Object child[] = getChilds();
      // decrementa a contagem de referências dos filhos
      // de forma recursiva
      for (int i = 0; i < child.length; i++)
        child[i].decrementCount();
      // desaloca a memória associada a este objeto
      free(this);
    }
}
```

A invocação dos finalizadores pode ser feita por um thread dedicado que executa algo como:

```
while (true) {
  if (GC.finalizationQueue.isEmpty())
    wait();
  else {
    // remove um objeto da lista de finalização
    Object obj = GC.finalizationQueue.remove();
    // invoca o finalizador
    obj.finalize();
    // restabelece a contagem de referências
    // e eventualmente desaloca o objeto
```

```
    obj.decrementCount();
  }
}
```

A rotina acima é responsável por ajustar a contagem de referências após a execução do finalizador. Note que este ajuste é feito de forma indireta através de uma nova chamada para `decrementCount`. Caso o finalizador não ressuscite o objeto finalizado, `decrementCount` efetua a sua desalocação.

De forma análoga ao caso síncrono, nessa implementação a ordem de invocação dos finalizadores também segue a ordem das referências entre os objetos (desde que a fila usada siga o padrão FIFO), e os objetos referenciados por um objeto finalizável são preservados até que o mesmo seja finalizado.

6.1.2 Referências Fracas

A implementação de referências fracas em coletores de lixo que utilizam exclusivamente contagem de referências segue diretamente da definição dessa abstração: a criação de uma referência fraca para um objeto não altera a contagem de referências associada ao mesmo. Para evitar o surgimento de ponteiros quebrados, a limpeza das referências fracas deve ocorrer imediatamente após a contagem de referências do respectivo objeto atingir zero. Para que seja possível efetuar essa limpeza é necessário manter algum mecanismo para encontrar todas as referências fracas que apontam para um determinado objeto. Uma alternativa consiste na utilização de uma tabela hash contendo listas de referências fracas. Toda vez que uma referência fraca é criada, ela é inserida na lista associada ao objeto referenciado (o objeto referenciado é usado como a chave de busca). Antes de desalocar qualquer objeto, o coletor verifica se existe uma lista não nula associada a esse objeto na tabela de referências fracas. Todas as referências fracas encontradas são limpas, e só então o objeto é desalocado. Para tornar mais eficiente este processo de verificação pode ser usado um bit extra no header de objetos alocados na heap. Somente objetos fracamente referenciados são marcados.

Conforme já discutimos, em sistemas nos quais referências fracas são limpas assim que a contagem de referências do objeto referenciado atinge zero, não é possível usar referências fracas para implementar caches de objetos. Considere o seguinte exemplo:

```
Blob b = loadBlob(); \\ carrega um objeto binário qualquer
WeakReference cache = new WeakReference(b);  \\cria o cache
```

```
b = null;  \\ limpa a referência original
```

Se a coleta de lixo for feita usando contagem de referências, o cache será limpo imediatamente após a referência contida em `b` ser destruída, tornando esta solução completamente inefetiva.

É possível efetuar uma implementação mais flexível de referências fracas, mas somente em sistemas que empregam um coletor secundário baseado em rastreamento. Neste caso referências fracas são implementadas como referências normais, ou seja, modificam a contagem de referências dos objetos fracamente referenciados. Porém, quando o coletor secundário é acionado, o rastreamento de referências fracas acontece conforme o esperado: os endereços armazenados por referências fracas são ignorados, e se um objeto for apenas fracamente acessível, ele é coletado e a respectiva referência fraca é limpa.

6.2 Coletores de Lixo Baseados em Rastreamento

Objetos com finalizadores e objetos que representam referências fracas obviamente devem ser tratados de forma diferenciada durante o processo de coleta de lixo. Para isso é necessário que o coletor seja capaz de distinguir tais objetos dinamicamente. Em coletores que utilizam rastreamento essa identificação pode ser feita de duas maneiras:

- Através do próprio processo de rastreamento. A informação sobre como tratar cada objeto pode, por exemplo, ser armazenada no header do objeto, ou ser recuperada diretamente através do seu tipo.

- Através de uma estrutura externa, mantida pelo coletor de lixo ou pelo runtime, na qual são armazenadas referências para os objetos especiais. Estas estruturas são similares aos remembered sets discutidos na Seção 2.5.

A escolha entre estas duas opções depende basicamente do número relativo de objetos que devem ser tratados de forma especial pelo coletor. Como na maioria dos sistemas este valor é baixo, o uso de estruturas externas normalmente proporciona um melhor desempenho (em termos de memória), e é a solução mais comum em implementações reais, sendo adotada por exemplo nas distribuições oficiais de Java [16] e C# [56].

6.2.1 Finalizadores

Para implementar finalizadores em sistemas com coletores baseados em rastreamento são usadas até duas estruturas de dados específicas:

- A *lista de objetos finalizáveis* armazena referências para todos os objetos cujos finalizadores devem ser executados antes do objeto ser desalocado pelo coletor de lixo (objetos com finalizadores habilitados). Toda vez que um objeto finalizável é alocado, ou que um objeto qualquer é registrado para finalização, é criada uma referência para o objeto na lista de objetos finalizáveis[1]. Note que a habilitação e a reabilitação dinâmica de finalizadores podem ser facilmente implementadas através de rotinas que respectivamente inserem e removem referências desta lista.

- A *lista de finalização* armazena referências para objetos que tornaram-se inacessíveis, mas que não podem ser desalocados até que o respectivo finalizador seja invocado.

Algumas implementações de finalizadores em coletores mark-and-sweep não utilizam listas de objeto finalizáveis, mas apenas uma lista de finalização. Ainda assim, o processo de coleta neste tipo de sistema pode ser implementado através de uma sequência comum de passos, descritos na Figura 6.1.

Nesse esquema o rastreamento de objetos a partir de objetos finalizáveis (passo (iii)), ou de finalizadores se estes forem representados por fechamentos, é necessário para impedir o surgimento de ponteiros quebrados. Uma alternativa para evitar esse rastreamento adicional seria efetuar a invocação de finalizadores assim que cada objeto finalizável não marcado fosse identificado. Porém, como a execução de um finalizador pode implicar na alocação de novos objetos e em ressurreições, a invocação de finalizadores durante a coleta possivelmente iria interferir destrutivamente com o processo de coleta.

Devemos observar ainda que o uso de uma lista de objetos finalizáveis apresenta duas vantagens importantes: elimina a necessidade de varrer toda a memória em busca de objetos finalizáveis, e permite que a invocação dos finalizadores aconteça sempre na ordem de elaboração dos objetos, ou da associação entre objetos e os

[1]Em linguagens orientadas a objetos o finalizador compõe a definição da classe, e qualquer finalizador pode ser facilmente encontrado a partir do tipo do objeto. Já em linguagens onde finalizadores são registrados dinamicamente, a lista de objetos finalizáveis normalmente é usada também para armazenar referências para as rotinas de finalização.

(i) Rastrear todos os objetos encontrados na memória heap, começando pelo conjunto-raiz. Os objetos encontrados devem ser marcados.

(ii) Se os objetos finalizáveis forem identificados através de informações contidas no próprio objeto, varrer toda a memória heap. Para cada objeto finalizável que não está marcado, criar uma referência na lista de finalização. Se os objetos finalizáveis forem identificados através de uma lista de objetos finalizáveis, verificar o status de todos os objetos referenciados por esta lista. Transferir as referências para objetos não marcados para a lista de finalização.

(iii) Efetuar um rastreamento a partir de cada referência encontrada na lista de finalização. Todos os objetos encontrados devem ser marcados como acessíveis.

(iv) Varrer toda a memória heap. Os objetos marcados devem ser desmarcados, e os objetos desmarcados devem ser desalocados.

(v) Invocar os finalizadores dos objetos diretamente referenciados pela lista de finalização, removendo da lista as referências correspondentes.

Figura 6.1: Coleta de lixo e de finalização em coletores mark-and-sweep.

respectivos finalizadores, conforme a linguagem considerada. Para garantir esta ordenação é necessário apenas que a remoção de referências nas listas de objetos finalizáveis e de finalização obedeça ao padrão LIFO ou FIFO, dependendo da ordem de invocação desejada.

A invocação de finalizadores seguindo o sentido das referências entre objetos demanda uma solução um pouco mais elaborada. Uma opção é usar sempre uma lista de objetos finalizáveis. Após o rastreamento inicial da memória heap efetua-se um rastreamento adicional a partir de cada objeto referenciado pela lista de objetos finalizáveis que não estiver marcado. Durante cada rastreamento, todos os objetos visitados, com exceção do objeto inicial, são marcados (se o objeto inicial for alcançado *durante o rastreamento que teve origem nele próprio*, ele não deve ser marcado). Ao final desta fase de rastreamento o coletor verifica quais objetos finalizáveis não estão marcados, e transfere as referências correspondentes para a lista de finalização.

Além de permitir a invocação ordenada de finalizadores, esta implementação garante a finalização de objetos mesmo quando estes formam ciclos de referências.

Considere o grafo de conectividade apresentado na Figura 6.2, onde todos os objetos são inacessíveis (não foram marcados), e os objetos A e B precisam ser finalizados.

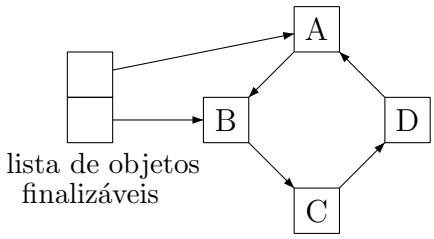

Figura 6.2: Invocação ordenada de finalizadores com referências cíclicas.

Se o rastreamento da lista de objetos finalizáveis começar pela referência que aponta para A, os objetos B, C e D serão marcados nesta mesma ordem. O objeto A entretanto permanece desmarcado. Em seguida é feito um rastreamento a partir da referência que aponta para B. Como B já está marcado, o rastreamento é interrompido, e como não há mais referências na lista de objetos finalizáveis, a fase de rastreamento termina. O coletor verifica então quais objetos finalizáveis não estão marcados, neste caso A, e transfere a referência correspondente para a lista de finalização. No próximo ciclo de coleta, supondo B não seja ressuscitado, a referência para o objeto B será transferida para a lista de finalização, garantindo assim a finalização deste objeto.

Se o rastreamento da lista de objetos finalizáveis começar pela referência que aponta para B, de forma análoga ao caso anterior, este objeto será transferido para a lista de finalização. No ciclo seguinte, supondo que A não seja ressuscitado, a referência para o objeto A será transferida para a lista de finalização.

Cabe ressaltar que dentre as poucas linguagens que garantem a invocação de finalizadores seguindo a ordem de referências entre os mesmos (Dolphin Smalltalk, Modula-3, etc), não conhecemos nenhuma implementação que efetua a invocação de finalizadores que compõem ciclos. A solução que acabamos de descrever constitui portanto uma alternativa interessante para estes casos, já que evita vazamentos de memória.

Na maioria das vezes o custo adicional de processamento associado a finalização ordenada de objetos é baixo (é linear com relação ao número de objetos finalizáveis), mas a desalocação de cadeias acíclicas compostas por objetos finalizáveis leva no mínimo um número de ciclos igual ao comprimento da cadeia, atrasando ainda mais o processo de coleta. Por outro lado, mesmo quando a in-

vocação de finalizadores não é feita de forma ordenada, a necessidade de evitar ponteiros quebrados faz com que a coleta de objetos finalizáveis, e de todos os objetos referenciados por objetos finalizáveis, seja atrasada em pelo menos um ciclo.

Em sistemas que empregam coletores stop-and-copy, ou coletores generacionais, a implementação de finalizadores precisa usar uma lista de objetos finalizáveis para evitar a necessidade de percorrer toda a memória heap durante cada ciclo de coleta. Nestes sistemas o processo de coleta pode ser estruturado de forma semelhante ao esquema apresentado para coletores mark-and-sweep. Após copiar todos os objetos acessíveis para o semi-espaço de destino (o tospace, ou a geração seguinte), o coletor verifica quais objetos referenciados pela lista de objetos finalizáveis não foram copiados. As referência correspondentes a estes objetos são transferidas para a lista de finalização, e a partir de cada elemento desta lista é feito um rastreamento. Ao invés dos objetos serem marcados, eles são transferidos imediatamente para o semi-espaço de destino.

Por fim, é interessante comparar as estratégias de implementação discutidas aqui com as regras de transição descritas na Seção 5.3. Podemos observar que, exceto pelo fato das regras formais descreverem transições que afetam o estado de um único objeto, a sua representação é semelhante a estrutura das soluções apresentadas nesta seção. Mais especificamente, temos que:

- A lista de objetos finalizáveis desempenha um papel equivalente ao do ambiente F, introduzido para armazenar finalizadores e indicar quais objetos devem ser finalizados. Como λ_{fin} não é tipada, finalizadores precisam ser registrados explicitamente, tal como acontece na maioria das linguagens com coleta de lixo que não são orientadas a objetos.

- Para impedir o surgimento de ponteiros quebrados, objetos finalizáveis e objetos referenciados por objetos finalizáveis não podem ser coletados ao menos até que os respectivos finalizadores sejam executados. Esta semântica, descrita por **gc3**, é implementada através do passo (iii) na Figura 6.1.

- Em sistemas que garantem a invocação ordenada de finalizadores, um finalizador só pode ser invocado se não existirem referências para o objeto a ser finalizado, ou se as referência existentes comporem um ciclo. Esta semântica é representada através da regra **fin-exec3**, e é implementada exatamente da mesma forma através da adaptação que descrevemos nesta seção para realizar a invocação ordenada de finalizadores. Se existir um ciclo entre

objetos finalizáveis, um destes objetos é selecionado arbitrariamente para ser finalizado, rompendo o ciclo de referências. Os demais finalizadores são invocados seguindo a ordem das respectivas referências.

6.2.2 Referências Fracas

Em sistemas que usam rastreamento, durante a construção do grafo de conectividade os ponteiros originados em referências fracas devem ser ignorados. Para isso é preciso que o coletor de lixo seja capaz de identificar objetos que representam referências fracas enquanto o rastreamento está sendo feito. De forma análoga ao que acontece com finalizadores, existem ao menos duas alternativas para efetuar essa identificação:

- Incluir a informação diretamente no header dos objetos.
- Usar uma estrutura de dados externa (a *lista de referências fracas*), mantida pelo coletor de lixo ou pelo runtime.

Na implementação de finalizadores descrita na seção anterior, a lista de objetos finalizáveis armazena referências para os objetos que devem ser finalizados. A lista de referências fracas, diferentemente, armazena os próprios objetos que representam referências fracas. Com isso o coletor de lixo pode ignorar automaticamente, durante a fase de rastreamento, todas as referências armazenadas na área de memória ocupada pela lista de referências fracas. Ao final desta fase o coletor verifica o status de cada objeto referenciado pela lista. Sempre que for encontrado um objeto não marcado, o seu endereço é substituído por zero (a referência fraca é limpa).

Como referências fracas são alocadas diretamente na lista de referências fracas, esta lista também precisa ser coletada. Se referências fracas forem representadas por objetos de tamanho fixo, como acontece por exemplo em Java e em C#, a área de memória ocupada pela lista de referências fracas não irá sofrer fragmentação. É suficiente então marcar cada um dos slots desta lista como ocupado ou vazio, pois assim a desalocação acontecerá de forma automática durante a fase de rastreamento (slots não marcados são considerados livres)[2].

Em linguagens que permitem transformar qualquer objeto em uma referência fraca (por exemplo, Smalltalk), o uso de listas de referências fracas é bem mais

[2]As marcas associadas aos slots da lista de referências fracas, ao contrário do que acontece no resto da memória heap, só são apagadas no início da fase de rastreamento.

complicado. É necessário copiar objetos marcados como fracos da memória heap para a lista de referências fracas, instalando forward pointers nos endereços originais. Uma alternativa de implementação mais simples consiste na alocação de um bit extra no header de objetos para indicar uma referência fraca. No início de cada ciclo de coleta a lista de referências fracas é limpa. Durante a fase de rastreamento o coletor ignora as referências contidas em objetos marcados, mas cria uma referência na lista de referências fracas para cada objeto fraco (marcado como fraco) que for encontrado. Ao final da fase de rastreamento o coletor verifica o estado dos objetos fracamente referenciados (que podem ser facilmente encontrados através da lista de referências fracas), e limpa os campos dos objetos fracos que contêm referências para os objetos inacessíveis.

A implementação de referências fracas com diferentes gradações de força pode ser feita usando listas de referências fracas específicas para representar cada gradação. Os endereços armazenados na lista de referências suaves por exemplo devem ser rastreados normalmente, como no resto da memória heap, mas somente enquanto o nível de utilização da memória for baixo. Se esta condição mudar esses endereços passam a ser ignorados pelo coletor.

Em sistemas que empregam coletores generacionais é possível implementar gradações de força de uma forma bem mais simples. Basta alocar objetos fracamente referenciados em gerações diferentes, dependendo do grau de força de cada referência. Objetos referenciados por referências suaves podem ser alocados diretamente na geração mais antiga, o que garante que só serão coletados quando houver uma coleta completa da memória (disparada apenas quando a disponibilidade de memória for reduzida).

A implementação de mecanismos de notificação associados a referências fracas é similar a implementação de finalizadores. No caso da notificação via callback é necessário usar uma lista externa, denominada *lista de notificação*, que desempenha um papel semelhante ao da lista de finalização. Sempre que o coletor limpar uma referência fraca deve ser criada uma nova referência para a mesma na lista de notificação. Os callbacks associados aos objetos referenciados pela lista de notificação são invocados posteriormente de forma assíncrona, geralmente recebendo a referência fraca como parâmetro. No caso de filas de notificação, uma referência para a referência fraca é inserida na fila assim que esta é limpa.

Ephemerons não podem ser implementados em coletores que usam exclusivamente contagem de referências devido a incapacidade destes sistemas em reclamar objetos que formam ciclos de referências, por isso não consideramos tal mecanismo

na Seção 6.1.2.

Em coletores baseados em rastreamento podemos implementar ephemerons usando duas listas de referências fracas modificadas para armazenar pares de referências (chave/valor), ao invés de referências simples. Os ephemerons (os pares chave/valor) são inicialmente alocados em uma das listas (a *lista ativa*), enquanto a outra permanece vazia. Cada ephemeron é rastreado normalmente durante a fase de rastreamento, sendo marcado quando acessado pelo coletor. Os objetos referenciados pela chave e pelo valor no entanto são inicialmente ignorados. O processo de coleta e limpeza obedece a sequência de passos apresentada na Figura 6.3, que é semelhante ao algoritmo descrito por Hayes [35].

(i) Rastrear todos os objetos encontrados na memória heap começando pelo conjunto-raiz. Os objetos encontrados são marcados.

(ii) Para cada ephemeron na lista ativa, se o ephemeron e o objeto referenciado pela chave estiverem marcados, efetuar um rastreamento a partir do valor, marcando todos os objetos encontrados. Após cada rastreamento, transferir o ephemeron rastreado para a outra lista de referências fracas (que estava vazia no início deste ciclo de coleta).

(iii) Repetir o passo (ii) até que nenhum ephemeron marcado contenha uma chave cujo objeto referenciado esteja marcado.

(iv) Examinar a marca dos ephemerons que ainda estão na lista ativa. Ephemerons que estiverem marcados devem ser limpos e transferidos para a outra lista de referências fracas.

(v) Limpar completamente a lista ativa (os ephemerons remanescentes serão desalocados). A lista de referências fracas para a qual os ephemerons foram transferidos passa a ser usada como a lista ativa.

Figura 6.3: Algoritmo para a coleta e limpeza de ephemerons.

A iteração definida no passo (iii) é necessária porque quando o coletor descobre que o objeto referenciada pela chave de um ephemeron é acessível, os objetos referenciados por chaves de outros ephemerons podem vir a ser considerados acessíveis também.

Note que na implementação acima a conectividade do valor depende da conectividade da chave e da conectividade do próprio ephemeron. Para implementar uma semântica semelhante a de Glasgow Haskell, na qual a conectividade do valor

depende exclusivamente da conectividade da chave, podemos alterar os passos (ii) e (iv) da seguinte forma:

- Em (ii) devem ser rastreados todos os valores de ephemerons cujas chaves estiverem marcadas.

- Em (iv), além de examinar as marcas dos ephemerons que ainda estão na lista ativa, devem ser verificadas também as marcas dos objetos referenciados pelas chaves destes ephemerons. Ephemerons que estiverem marcados devem ser limpos e transferidos para a outra lista de referências fracas. Ephemerons não marcados, mas com chaves marcadas, devem ser marcados e também transferidos para a outro lista de referências fracas, evitando assim a sua coleta.

Ao contrário do que aconteceu com finalizadores, as regras de transição desenvolvidas na Seção 5.4 não representam tão bem a dinâmica real do processo de coleta e limpeza de referências fracas, tal como descrevemos aqui. Além das diferenças semânticas intrínsecas ao modelo de referências que adotamos, na nossa formalização (regras **gc4** e **w-clear**) a criação de uma referência fraca para um objeto impede a sua coleta enquanto a respectiva referência não for limpa. Na prática porém, como foi visto nesta seção, o processo de rastreamento e a efetiva limpeza das referências fracas acontece de forma atômica. Essa restrição de coleta portanto é totalmente irrelevante. Referências fracas ordinárias normalmente não interferem com o ciclo de vida dos objetos que referenciam.