4
Semântica: Uma Análise Informal

Finalizadores e referências fracas tornam a dinâmica do coletor de lixo aparente para o programa cliente, criando uma dependência intrínseca entre a semântica da linguagem e a implementação das rotinas de gerenciamento automático de memória. Ainda assim a especificação da maioria das linguagens de programação é bastante vaga em relação ao coletor de lixo, impondo poucas ou nenhuma restrição à sua implementação. Isso cria dificuldades consideráveis para descrever adequadamente as semânticas de finalizadores e referências fracas, e por conseguinte, da própria linguagem de programação.

Esse problema é acentuado ainda pela própria complexidade inerente a tais abstrações, decorrente sobretudo de alguns fatores comuns relacionados ao processo de coleta de lixo:

- Nem sempre é possível, através de uma análise estática, determinar com precisão quando um objeto torna-se efetivamente coletável. De acordo com a especificação da linguagem Java [43],

  > "otimizações efetuadas no código do programa podem fazer com que o número de objetos considerados acessíveis seja menor do que o esperado. Por exemplo, o compilador, ou o gerador de código, pode decidir atribuir o valor `null` a uma variável ou a um parâmetro que não vai mais ser usado, fazendo assim com que a memória associada ao objeto seja reciclada mais rapidamente."

  Em alguns casos um objeto pode tornar-se coletável mesmo enquanto um de seus métodos ainda está sendo executado. Apesar da referência para o próprio objeto ser um parâmetro implícito na chamada de métodos ordinários, a referência correspondente no registro de ativação pode ser eliminada como consequência de uma otimização de tail-call, ou simplesmente nem ser criada caso o método seja executado inline. Portanto se a partir

de um ponto qualquer do método não existirem mais referências explícitas para o objeto, este pode ser coletado.

- Como já mencionamos, coletores de lixo baseados em rastreamento determinam se um objeto é coletável, e eventualmente invocam ou executam rotinas associadas a tal evento, de forma não determinística.

- A execução de finalizadores e de callbacks associados a referências fracas introduz linhas de processamento concorrente, mesmo em sistemas que não oferecem suporte a mecanismos básicos de concorrência e sincronização.

- Tanto finalizadores quanto referências fracas permitem ao programa cliente ter acesso a objetos coletáveis, e em algumas raras implementações, até mesmo a objetos já coletados (por exemplo, Multilisp [34]). Além das dificuldades práticas associadas a esta dinâmica (ressurreição), do ponto de vista abstrato também surgem algumas questões importantes. Como devem ser tratados objetos finalizados que foram ressuscitados? O sistema precisa diferenciar estes objetos, e se positivo, esta diferenciação deve ser visível para o programa cliente?

- Como o comportamento de finalizadores e referências fracas depende, ainda que parcialmente, da implementação do coletor de lixo, podem surgir problemas de portabilidade entre diferentes versões de uma mesma linguagem que empregam diferentes rotinas de gerenciamento de memória.

Assim, apesar de finalizadores e referências fracas serem conceitos relativamente simples, existem inúmeras questões semânticas e de implementação que devem ser cuidadosamente analisadas antes de definir como introduzir suporte a estas abstrações em uma linguagem de programação qualquer. Neste capítulo identificamos e discutimos estas questões, bem como as suas respectivas implicações. Antes porém iremos apresentar exemplos que ilustram algumas das dificuldades básicas associadas à implementação e ao uso de finalizadores e referências fracas.

4.1 Dificuldades de Uso e Implementação

O processo de coleta de lixo geralmente precisa ser adaptado para atender aos requisitos semânticos de finalizadores e referências fracas, o que implica em mudanças na implementação das rotinas de gerenciamento de memória. Em linguagens orientadas a objetos, por exemplo, a desalocação da memória ocupada

por um objeto finalizável não pode ocorrer até que o finalizador correspondente seja executado. Esta restrição atrasa o processo de reciclagem de memória, e em alguns casos, pode interferir de forma significativa no desempenho da aplicação.

O programa seguinte ilustra esse problema no ambiente Java[1]. Cinco threads são usados para instanciar objetos finalizáveis que tornam-se inacessíveis imediatamente após a sua criação. Como a prioridade e o número de threads utilizados pela máquina virtual para executar os finalizadores são inadequados para esta aplicação, a velocidade de criação de objetos supera a velocidade de coleta. Como resultado desta dinâmica, a memória disponível se esgota rapidamente, e o sistema lança uma exceção.

```
/*
 * Esta é uma classe finalizável. Instâncias desta classe armazenam
 * um número (double), e antes de serem desalocadas, imprimem o
 * número armazenado.
 */
class FinalizableDouble {

  // o número armazenado
  double data;

  FinalizableDouble(double data){
    this.data = data;
  }

  protected void finalize(){
    System.out.println(data);
  }
}

/*
 * Esta classe cria um número pré-definido de threads. Cada thread
 * cria um número pré-definido de objetos da classe
 * FinalizableDouble, mas não mantém referências para os mesmos.
 * A princípio a memória ocupada por cada objeto poderia ser
```

[1]Os exemplos em Java apresentados aqui foram implementados usando a versão 1.4.1 desta linguagem. A execução foi feita em um computador dotado de um Pentium 4 de 2.8 GHz e 512 Mb de memória RAM. Os resultados obtidos usando Linux (Red Hat 9.0) e Windows XP foram idênticos.

```
 * imediatamente reclamada.
 */
public class MemoryTest extends Thread {

  // o código a ser executado por cada thread
  public void run() {
    // número de objetos criados por cada thread
    int iter = 100000000;
    for (int i=0; i<iter; i++)
      new FinalizableDouble(i);
  }

  public static void main(String args[]) {
    // número de threads
    int nThreads = 5;
    MemoryTest thread[] = new MemoryTest[nThreads];
    // cria os threads e inicia sua execução
    for(int i =0; i< nThreads; i++){
      thread[i] = new MemoryTest();
      thread[i].start();
    }
  }
}
```

O próximo exemplo, também em Java, ilustra dois problemas típicos de concorrência associados à implementação e ao uso de finalizadores. Java adota como mecanismo básico de sincronização um modelo semelhante a monitores. Cada instância de uma classe possui um único lock, que deve ser compartilhado entre diferentes threads que desejam executar um ou mais métodos sincronizados. Este lock é recursivo, isto é, o thread que detém o lock de um objeto pode invocar seguidamente qualquer um de seus métodos (sincronizados ou não).

No programa seguinte uma coleta de lixo é disparada dentro de um monitor. Como o thread principal já detém um lock, um novo thread é escalonado para executar o finalizador (a especificação da linguagem Java permite que qualquer thread seja usado para a execução de finalizadores, desde que o mesmo não esteja de posse de um lock visível para o programa cliente). Este thread entretanto precisa obter o lock associado ao monitor para conseguir completar a execução do finalizador. Porém, como o thread principal está bloqueado dentro do monitor

aguardando o fim da coleta, a aplicação entra em deadlock.

```
/*
 * Esta é uma classe auxiliar usada para ilustrar o problema de
 * sincronização de finalizadores. Ela contém dois métodos
 * sincronizados, um dos quais cria objetos não finalizáveis.
 */
class Monitor {

  // Variável compartilhada
  private int sharedData = 0;

  // As duas próximas rotinas, por alguma razão não contemplada aqui,
  // precisam ser sincronizadas.

  public synchronized void callA() {
    ++sharedData;
  }

  public synchronized void callB() {
    --sharedData;
    // aloca memória para forçar uma coleta e a invocação
    // de finalizadores
    int size = 100000;
    Integer v[];
    for (int i = 0; i < size*size; i++)
       v  = new Integer[size];
  }
}

/*
 * Esta é uma classe finalizável. O finalizador invoca um método
 * sincronizado de um objeto da classe Monitor.
 */
class FinalizableClass {

  // objeto usado pelo finalizador
  static Monitor monitor = null;
```

```
  public void finalize() {
    System.out.println("Executando finalizador no thread " +
    Thread.currentThread().getName());
    monitor.callA();
  }
}

/*
 * Esta classe executa o teste.
 */
public class MonitorTest{

  public static void main (String args[]){
    Monitor monitor = new Monitor();
    FinalizableClass.monitor = monitor;
    new FinalizableClass();
    System.out.println("Invocando callB do thread principal");
    monitor.callB();
  }
}
```

Se a especificação da linguagem Java não restringisse a execução de finalizadores em threads que não detém um lock, durante a execução do programa acima o finalizador conseguiria acessar o monitor enquanto o thread principal estivesse bloqueado dentro do mesmo. Em um certo sentido, isto violaria o conceito de monitor. Algumas implementações da máquina virtual Java, inclusive da própria Sun (Java 1.0), permitem esta violação. Um exemplo específico em que isto ocorre é descrito em [1].

Conforme já observamos, teoricamente um objeto pode ser finalizado enquanto um de seus métodos está sendo executado. Para isso basta que não seja mais necessário acessar a referência para o próprio objeto (this) a partir de um determinado ponto durante a execução do método. Boehm [18] discute um exemplo em que esta dinâmica pode levar a erros na execução de um programa. Considere o seguinte pseudo-código:

```
/*
 * Essa classe implementa um contador simples.
 */
class Counter {
```

```
  int counter = 0;
}

/*
 * Esta é uma classe finalizável. Instâncias desta classe
 * armazenam o número de invocações de um de seus métodos
 * em um array estático. Cada instância possui um índice
 * próprio para acessar o array.
 */
class FinalizableClass2 {

  // número total de objetos desta classe que foram criados
  private static int index = 0;

  // número máximo de objetos desta classe que podem ser criados
  // por um aplicação
  final static int MAX_ELEM = 1000000;

  // array usado por todos os objetos desta classe
  private static Counter counter[] = new Counter[MAX_ELEM];

  // índice deste objeto
  private int myIndex;

  // construtor - aloca um objeto no array counter
  FinalizableClass2(){
    myIndex = index++;
    if (myIndex==MAX_ELEM)
      System.exit(-1);
    counter[myIndex] = new Counter();
  }

  // este método conta quantas vezes ele próprio foi invocado
  void foo(){
    ++counter[myIndex].counter;
  }

  // finalizador - limpa a entrada correspondente
```

```
  // ao objeto no array counter
  void finalize(){
    counter[myIndex] = null;
  }
}
```

e suponha que um objeto `obj` do tipo `FinalizableClass2` seja acessado pela última vez por uma aplicação qualquer no seguinte trecho de código:

```
for (int i =0; i < 1000000; i++) {
  obj.foo();
}
```

O compilador pode transformar a chamada do método `foo` em um código inline, e manter o endereço de `obj.myIndex` em um registrador (nesta rotina a referência para `obj.myIndex` é invariante), eliminando assim qualquer referência para `obj`. Caso o finalizador deste objeto seja invocado enquanto o loop ainda estiver executando, a próxima iteração resultará em um erro (null pointer). O ambiente .Net permite evitar problemas como este através de uma chamada específica (`GC.keepAlive`) que impede a coleta e finalização prematura de objetos.

O manual da linguagem ML [5] apresenta dois exemplos interessantes que ilustram algumas das dificuldades inerentes à semântica de referências fracas. Considere a expressão

```
let
  val (b', w') = let
    val a = (1, 2)
    val b = (1, 2)
    val w = weak(a)
  in (b,w) end
in (b, strong w) end
```

onde a função `weak(a)` cria uma referência fraca; `strong w` retorna `NONE` caso o objeto referenciado por `w` já tenha sido coletado, ou `SOME(a)` caso contrário; e `NONE` e `SOME` representam os possíveis valores de uma opção.

Após a avaliação desta expressão, como a variável `a` está tanto estaticamente quanto dinamicamente morta, o valor esperado de `(b', w')` a princípio seria `((1,2),NONE)`. Mas como referências fracas só são implicitamente modificadas quando o coletor de lixo é executado, o resultado desta expressão poderia ser

`((1,2),SOME(1,2))`. Por outro lado se o compilador efetuar a eliminação de subexpressões comuns, de tal forma que `a` e `b` referem-se ao mesmo `(1,2)`, o valor de `w` pode continuar acessível mesmo após uma coleta, desde que `b` continue acessível.

Já na expressão

```
let
  val a = (1,2)
  val w = weak(a)
in (a,w) end
```

se o compilador usar um registrador para representar o valor de `a`, a atribuição `val w = weak(a)` iria implicar na alocação de uma nova cópia de `(1,2)` na memória heap. Ainda que `a` continue acessível, `w` pode ser imediatamente limpa após a atribuição, já que não existem referências fortes para o seu conteúdo.

Referências fracas não precisam necessariamente ser implementadas através de um tipo específico, e muitas linguagens suportam mecanismos que permitem marcar referências ordinárias como fracas (por exemplo, Perl e Smalltalk). Ainda que esta semântica aparentemente seja atraente, ela pode facilmente induzir a erros de programação. Considere o seguinte pseudo-código:

```
Object o;
Vector v = new Vector();
v.weaken(); // marca v como uma referência fraca
...
if (v!=null)
  v.add(o);
```

O teste na cláusula condicional não garante que `v` é diferente de `null` quando o método `add` é invocado, podendo assim levar a um erro crítico de execução (null pointer).

4.2 Finalizadores

Nesta seção exploramos os pontos que de acordo com a nossa visão são mais relevantes para a definição da semântica de finalizadores.

4.2.1 Desacoplamento e Ressurreição

Finalizadores baseados em classes geralmente têm acesso irrestrito aos objetos aos quais estão associados. Esse acoplamento permite que todas as informações usadas pela rotina de finalização fiquem armazenadas no próprio objeto finalizável, preservando portanto o encapsulamento de dados. A princípio esta semântica impõe duas restrições:

- A coleta do objeto finalizável precisa ser adiada até que a sua finalização esteja completa.
- Tanto a coleta quanto a finalização de quaisquer objetos referenciados direta ou indiretamente por um objeto finalizável precisam ser adiadas. Sem isso a rotina de finalização pode ter acesso a objetos finalizados, o que pode constituir um erro semântico, ou ainda pior, ter acesso a objetos que já foram coletados.

A preservação do objeto finalizado, bem como de todos os objetos acessíveis através do mesmo, é essencial para garantir a correção da linguagem e do mecanismo de finalização (no caso em que o finalizador tem acesso ao objeto finalizado). Por outro lado, a invocação ordenada de finalizadores é necessária apenas sob a perspectiva da semântica de aplicações específicas. Como essa ordenação pode aumentar significativamente o atraso no processo de coleta de lixo (ver a próxima seção), poucas linguagens optam por oferecer tal facilidade.

Se a rotina de finalização recebe como parâmetro o objeto a ser finalizado, a sua execução pode fazer com que este objeto torne-se novamente acessível ao programa cliente, isto é, ressuscite. A *ressurreição* ocorre quando a execução do finalizador faz com que uma referência para o objeto finalizado seja armazenada em uma variável global, ou em um outro objeto qualquer que permanece acessível após a execução do finalizador.

A ressurreição de objetos através de finalizadores dá origem a uma dinâmica incomum. Um objeto finalizável não pode interagir com o programa cliente entre o momento em que torna-se desconexo, e o instante em que é efetivamente finalizado. Não obstante, durante este intervalo de tempo o objeto retém o potencial de influenciar o comportamento futuro da aplicação. Esticando um pouco mais a analogia de ressurreição, é como se objetos finalizáveis passassem por um estado morto-vivo (ou zumbi) antes de ressuscitar. Mesmo quando necessário, um pro-

grama usualmente não tem como evitar a finalização de um objeto zumbi, nem as consequências imediatas desta finalização.

Para impedir a ressurreição através de finalizadores é preciso restringir totalmente o acesso do finalizador ao objeto finalizado. Contudo, na maioria das vezes não faz muito sentido definir um finalizador genérico que seja totalmente independente do objeto finalizado. É preciso usar então algum mecanismo ou estrutura de dados externa para armazenar as informações necessárias à finalização específica de cada objeto. Uma solução elegante é usar um fechamento, como faz a linguagem Ruby. Uma outra alternativa, viável apenas se o mecanismo de finalização for acoplado a referências fracas, é armazenar os dados a serem usados pela rotina de finalização diretamente em um array associativo. Como o objeto finalizado não é acessível durante a finalização, a própria referência fraca deve ser usada como chave de busca. Apesar da quebra de encapsulamento inerente a estas soluções, elas evitam que blocos de memória sejam preservados desnecessariamente, e não atrasam o processo de coleta.

Uma última questão relacionada à ressurreição diz respeito a finalização de objetos ressuscitados. O que acontece quando um objeto ressuscitado torna-se novamente coletável? Na solução mais comum o finalizador não é invocado novamente, mas algumas linguagens, como C# e Smalltalk, permitem reabilitá-lo de forma explícita. Em Java, como já discutimos, um finalizador só pode ser invocado uma única vez.

4.2.2 Ordem de Invocação

Em algumas aplicações a ordem de finalização dos objetos é importante. Considere o exemplo tradicional em que uma classe que representa um arquivo buferizado é implementada como uma agregação de dois objetos: um buffer e um descritor de arquivos. O finalizador do objeto composto deve ser executado antes dos finalizadores associados aos objetos componentes, garantindo assim que o buffer seja esvaziado antes do arquivo ser fechado ou os recursos associados ao buffer serem liberados.

Existem basicamente quatro alternativas para ordenar a invocação de finalizadores:

- A ordem cronológica em que finalizadores são registrados ou habilitados, como adotado por exemplo em Glasgow Haskell.

- A ordem de instanciação de objetos finalizáveis, como adotado por exemplo em Lua.

- O sentido das referências entre objetos finalizáveis, isto é, se o objeto `x` contém uma referência para o objeto `y`, e ambos devem ser finalizados, o finalizador de `x` é sempre invocado antes do finalizador de `y`. Essa solução é adotada por exemplo no coletor de C++ desenvolvido por Boehm (ver Seção 3.1.2), em Modula-3 e em Dolphin Smalltalk. Em linguagens que empregam contagem de referências os finalizadores são invocados na ordem em que os objetos tornam-se desconexos (na ordem em que a contagem de referências atinge zero), seguindo naturalmente o sentido das referências entre objetos finalizáveis.

- Uma ordem especificada explicitamente pelo programador. A linguagem pode oferecer algum tipo de mecanismo que permite definir uma prioridade para cada objeto finalizável. Em cada ciclo, o coletor invoca os finalizadores seguindo a ordem de prioridade dos respectivos objetos. Apesar de interessante, não conhecemos nenhuma linguagem que oferece este tipo de facilidade.

Do ponto de vista da semântica de aplicações, a ordem de invocação que segue o sentido das referências entre objetos geralmente é a mais correta. No entanto ela gera um impasse sempre que objetos finalizáveis formam referências cíclicas. Como visto no Capítulo 3, algumas linguagens que empregam coletores baseados em rastreamento, e respeitam a dinâmica acima, não são capazes de finalizar objetos que formam ciclos. Depois que estes objetos tornam-se inacessíveis não há mais como eliminar as suas referências cíclicas. Neste caso a restrição de finalização impede a sua coleta, gerando um vazamento de memória.

A invocação de finalizadores seguindo a ordem de referências introduz também um outro problema. Para garantir a ordem correta de finalização, em cada ciclo o coletor deve invocar inicialmente apenas os finalizadores de objetos inacessíveis que não são referenciados por outros objetos finalizáveis. Como o grafo de referências entre os objetos pode ser modificado durante a execução de um finalizador, o coletor precisa recalcular este grafo após a execução dos finalizadores inicialmente invocados. Essa tarefa entretanto consiste em um novo rastreamento, que por razões de eficiência, usualmente é adiado até o ciclo seguinte. Assim, a finalização de um conjunto de n objetos, no pior caso (uma lista encadeada por exemplo) pode levar até n ciclos de coleta.

Esses dois problemas não acontecem quando a invocação de finalizadores segue a ordem cronológica de habilitação, ou a ordem de elaboração dos objetos. Em ambas as alternativas a ordem relativa de finalização não é afetada pela execução de finalizadores, e o coletor de lixo é capaz de finalizar todos os objetos inacessíveis em um único ciclo.

Independentemente da garantia oferecida pela linguagem, normalmente não é difícil controlar a ordem de finalização dos objetos de forma explícita. Para isso pode-se usar uma estrutura de dados estática. Por exemplo, o construtor de um objeto definido como uma agregação (um objeto composto) insere uma referência para cada um de seus objetos componentes em uma tabela global. Essas referências são explicitamente destruídas pelo finalizador do objeto composto, garantindo assim que os finalizadores dos objetos componentes sejam invocados na ordem desejada.

O uso de filas, incluindo mecanismos como *guardians* (a ser discutido na Seção 4.3.1), constitui uma alternativa mais elaborada de finalização que permite uma ordenação conforme a lógica específica de cada aplicação.

Em linguagens que não oferecem garantias de ordenação não faz muito sentido exigir que objetos sejam finalizados ao término da aplicação, já que os objetos necessários a qualquer finalização podem ter sido previamente finalizados. Mesmo a solução explícita discutida acima não é suficiente para controlar a invocação de finalizadores, porque até mesmo os objetos referenciados estaticamente podem ter sido finalizados pelo coletor de lixo. Sobretudo por este motivo, algumas linguagens não garantem a invocação de finalizadores nessa fase final de processamento.

Por outro lado, a invocação ordenada de finalizadores no fim da aplicação impõe a necessidade de um ou mais rastreamentos adicionais. Tal demanda pode atrasar de forma significativa o término efetivo da aplicação.

4.2.3 Concorrência

Existem ao menos três alternativas distintas para a execução implícita de finalizadores (e de callbacks associados a referências fracas):

- Execução no próprio thread (ou threads) responsável pela execução do programa cliente. Neste caso o coletor de lixo também é executado por este thread, e a sua ativação acontece quando um novo objeto é explicitamente alocado.

- Execução no thread (ou threads) dedicado à execução do coletor de lixo.
- Execução em um ou mais threads dedicados exclusivamente a finalização.

Qualquer que seja a opção escolhida, a execução de finalizadores introduz um grau de concorrência na execução do programa cliente, mesmo em linguagens que não oferecem nenhum mecanismo básico de concorrência ou sincronização. Considerando-se ainda que um sistema pode optar pelo uso de múltiplos threads para a execução de finalizadores, o número de threads ativos em uma aplicação simples pode ser bem maior do que o esperado pelo programador[2]. Certamente essa dinâmica nem sempre é apropriada ou desejável, podendo levar a condições de corrida totalmente imprevistas. Considere por exemplo o código abaixo, adaptado de [18].

```
class X {
  Y mine;

  //construtor
  public X(){
    mine = new Y();
    ...
  }

  public foo(){
    ...
    mine.bar();
  }

  //finalizador
  public void finalize(){
    mine.baz();
  }
}
```

Suponha que em um programa qualquer, uma instância `x` da classe `X` é criada, e que o último acesso a `x` se dá através da chamada `x.foo()`. Se `x` for finalizado enquanto `x.foo` estiver executando, uma possibilidade que já discutimos, *mine.bar* e *mine.baz* serão executados simultaneamente, eventualmente acessando de forma concorrente uma estrutura de dados compartilhada.

[2]Em algumas linguagens é possível que finalizadores associados a instâncias diferentes de uma mesma classe sejam executados simultaneamente. Java constitui um exemplo.

Assim, o uso de mecanismos de sincronização associados a finalizadores é importante e mesmo inevitável em muitas situações, sobretudo quando finalizadores acessam variáveis globais. Infelizmente o uso de mecanismos de sincronização tende a deteriorar o desempenho da aplicação, e é uma fonte constante de erros de programação, inclusive deadlocks. Algumas linguagens nem sequer oferecem suporte básico a este tipo de mecanismo, o que contudo, pode dificultar bastante a correta implementação de programas que usam finalizadores.

A complexidade inerente à concorrência intrínseca e a esta eventual necessidade de sincronização pode levar até mesmo a erros de implementação da linguagem. Conforme discutimos na Seção 4.1, diferentes implementações da máquina virtual de Java permitem que o mecanismo de monitores seja violado. Isso pode ocorrer quando o finalizador é executado em um thread suspenso dentro de um método sincronizado, e precisa do lock associado ao objeto para completar a execução. A especificação da linguagem Java ao menos trata explicitamente desta questão, o que não acontece por exemplo em C#. O mecanismo básico de sincronização desta linguagem é semelhante ao de Java, com um lock por objeto. Ainda assim a sua especificação não impõe qualquer restrição ao thread a ser usado para executar um finalizador, e portanto, é perfeitamente legal que a execução de um finalizador viole o mecanismo de sincronização[3].

Um outro caso problemático acontece em linguagens que permitem a invocação de finalizadores ao término da aplicação. Finalizadores podem ser invocados enquanto threads de menor prioridade (basicamente daemon threads, que não impedem o término da aplicação) estão manipulando o objeto. Especificamente por esta razão Java aboliu a invocação de finalizadores ao término de aplicações (que podia ser habilitada através do método `System.runFinalizersOnExit`) [62].

4.2.4 Sincronismo de Invocação

O tipo de coletor usado na implementação da linguagem quase sempre determina o quão rapidamente objetos inacessíveis são considerados coletáveis, e quando os respectivos finalizadores são invocados. Em linguagens com coletores baseados em contagem de referências, como Perl e Python, a invocação de finalizadores geralmente é feita de forma síncrona (o finalizador é invocado imediatamente após a contagem de referências atingir zero)[4]. Em sistemas com coletores que em-

[3]Na implementação padrão de .Net isso aparentemente não acontece.

[4]Em aplicações multithread a execução do finalizador pode ser postergada em função de uma troca de contextos. Ainda assim, o thread cuja execução fez com que a contagem de referências

pregam técnicas baseadas em rastreamento, a invocação implícita de finalizadores é sempre não-determinística.

Como recursos externos muitas vezes são representados através de proxies, é natural que a sua aquisição seja feita através de construtores, e a desalocação através de finalizadores ou destrutores (esse padrão é conhecido em C++ como RAII: *resource acquisition is initialization* [60]). Quanto mais rápida é a desalocação destes recursos, mais eficiente é o seu uso. Logo, quando os recursos são escassos, ou possuem um custo não desprezível, a dinâmica de aquisição e desalocação pode tornar-se crítica para o desempenho global da aplicação.

Arquivos representam um exemplo típico. O número total de descritores de arquivo disponibilizados pelo sistema operacional é limitado, mas relativamente alto, e portanto raramente constitui uma restrição efetiva para a implementação de uma aplicação. Por outro lado, como operações de atualização podem requerer acesso exclusivo a um arquivo, em aplicações concorrentes pode ser importante que os descritores de arquivo sejam fechados agressivamente.

Conexões de banco de dados constituem um outro exemplo interessante. O número de conexões simultâneas que um gerenciador de banco de dados (DBM) pode fazer por vezes é limitado pela licença do software (o preço da licença varia com o número de conexões concorrentes). Se o número de conexões disponíveis for pequeno, a aplicação deve ser capaz de fechar a conexão tão logo quanto possível.

Infelizmente a invocação assíncrona de finalizadores introduz atrasos que podem criar gargalos críticos, limitando, ou mesmo impedindo o uso deste dispositivo para a liberação implícita de recursos.

Em linguagens cujo coletor controla a conectividade dos objetos através da execução de uma rotina de rastreamento, o atraso na execução de finalizadores possui dois componentes distintos:

- O hiato entre os instantes em que a última referência para o objeto desaparece e o coletor de lixo efetivamente determina que o objeto é finalizável.

- O hiato entre os instantes em que o coletor de lixo determina que o objeto é finalizável e o finalizador é efetivamente invocado.

Normalmente a frequência de invocação do coletor de lixo é inversamente proporcional a disponibilidade de memória do sistema[5]. Além disso, de forma geral o

chegasse a zero irá executar o finalizador imediatamente após ser reescalonado.

[5]No ambiente .Net coletas de lixo são disparadas dependendo do nível de utilização da memória e do número de handlers de janelas usados pela aplicação.

nível de utilização de um recurso externo possui uma baixa correlação com o uso da memória. Assim, mesmo que o programa cliente tenha implicitamente liberado todos os proxies correspondentes ao recurso externo, pode-se atingir uma situação em que o coletor nunca é invocado, não sendo mais possível a alocação e o uso do recurso em questão.

Esse problema pode ser parcialmente contornado em sistemas que permitem a invocação explícita do coletor de lixo. Para isso basta que o programa cliente invoque o coletor sempre que a utilização do recurso externo atingir níveis críticos.

O segundo componente do atraso na execução de finalizadores é intrínseco a implementação do sistema, e só pode ser reduzido pelo programa cliente se o sistema suportar chamadas que forçam a invocação de finalizadores agendados para execução, como é o caso tanto de Java quanto de C#.

Em sistemas que oferecem suporte a essas chamadas, finalizadores podem ser usados na liberação de recursos externos escassos desde que o custo de execuções adicionais do coletor de lixo não seja relevante para a aplicação considerada. A alocação de proxies que representam recursos externos deve ser realizada através de um método estático, que antes de efetuar a alocação, verifica a disponibilidade do recurso. Caso o nível de disponibilidade esteja abaixo de um limite crítico pré-estabelecido, o coletor de lixo é invocado explicitamente, e os finalizadores agendados são executados antecipadamente. Como a execução de finalizadores pode fazer com que novos objetos tornem-se coletáveis e finalizáveis, a invocação explícita do coletor deve acontecer através de um loop. O coletor deve ser invocado até que o nível de disponibilidade de recursos atinja um nível satisfatório, ou que a coleta não tenha nenhum efeito (para descobrir se uma coleta teve algum efeito basta monitorar a memória livre).

Durante o desenvolvimento do ambiente .Net, para tentar contornar o indeterminismo associado a coletores baseados em rastreamento, os autores desse sistema consideraram a possibilidade de introduzir classes que fossem automaticamente coletadas usando contagem de referências [33]. Duas dificuldades entretanto impediram a adoção desta solução:

- Sempre que um objeto controlado através de contagem de referências é referenciado por um objeto comum (coletado usando rastreamento), a sua coleta deixa de ser determinística. O objeto referenciado só vai ter a sua contagem de referências decrementada quando o objeto que contém a referência for coletado (de forma assíncrona).

- A hierarquia de classes e o mecanismo de polimorfismo complicam muito o processo de contagem de referências. Como o compilador deve lidar com um objeto controlado através de contagem de referências que é referenciado no código através de um supertipo comum (por exemplo `System.Object`, a classe raiz no ambiente .Net)? Nem sempre é possível determinar estaticamente se o objeto precisa ser controlado através de contagens de referências, e instruções extras podem tornar-se sempre necessárias, impondo um custo de processamento adicional proibitivo.

Como esta arquitetura demonstrou ser inviável, a alternativa encontrada foi a introdução da interface `IDisposable` e da expressão `using`, ambas discutidas no Capítulo 3.

Apesar da invocação assíncrona ser vista como um problema ou uma limitação inerente a coletores baseados em rastreamento, alguns autores defendem que finalizadores devem ser sempre invocados desta forma (por exemplo, Boehm [18]). A justificativa básica para essa posição está relacionada ao problema de sincronização discutido na seção anterior. Em linguagens que adotam um modelo de sincronização baseado em monitores, como Java ou C#, para preservar o modelo de sincronização é necessário que finalizadores sejam executados em um thread que não detém um lock visível. Caso estas linguagens efetuassem a coleta de lixo através de contagem de referências, duas alternativas para a execução de finalizadores poderiam ser consideradas:

- A solução mais natural seria executar um finalizador no mesmo thread que fez com que a contagem de referências chegasse a zero. Neste caso duas linhas de execução distintas poderiam estar simultaneamente ativas dentro do monitor, ou então finalizadores não poderiam ser sincronizados e nem invocar métodos sincronizados.

- Se um novo thread fosse disparado para executar o finalizador, a sua execução efetiva não seria mais síncrona, já que somente após a liberação do lock associado ao objeto seria possível executar o finalizador.

O problema de sincronização de finalizadores invocados de forma determinística torna-se ainda mais complexo devido à forma como referências são decrementadas. Qualquer fechamento de escopo ou operação de atribuição pode levar a destruição de referências e a invocação de um finalizador. Na hora de definir os requisitos de sincronização de uma aplicação essa dinâmica precisa ser

cuidadosamente considerada, o que nem sempre representa uma tarefa das mais simples.

4.3 Referências Fracas

De forma análoga ao que fizemos com finalizadores, nesta seção exploramos os pontos que consideramos mais relevantes na definição da semântica de referências fracas.

4.3.1 Coleta do Objeto Referenciado

Em sua definição original, referências fracas não impedem a coleta do objeto referenciado. Para evitar o problema de ponteiros quebrados, assim que um objeto fracamente referenciado é coletado (ou considerado coletável), o coletor de lixo limpa as referências fracas que apontavam para este objeto (substituem o endereço do objeto por um valor padrão como `nil` ou `null`).

Ainda que a princípio referências fracas sejam ignoradas pelo coletor de lixo, em algumas situações existem vantagens em modificar tal comportamento. Caches de memória geralmente serão mais efetivos se a coleta do objeto referenciado for adiada até que a memória se torne realmente escassa. Claro que este comportamento nem sempre é apropriado, podendo até mesmo prejudicar o desempenho da aplicação se recursos forem liberados em um ritmo muito lento. Uma alternativa interessante para atender a estes requisitos distintos é implementar referências fracas com diferentes graus de fraqueza, cada qual associada a uma prioridade diferente de reciclagem. Como discutido no Capítulo 3, esta é a abordagem adotada por Java. O modelo usado em Eiffel é ainda mais flexível, e permite ao programador definir explicitamente quando um objeto fracamente referenciado deve ser coletado.

O ambiente .Net suporta apenas um tipo de referência fraca. Porém é possível simular referências suaves em função de uma particularidade do seu mecanismo de gerenciamento de memória. O Microsoft Common Language Runtime (CLR) usa um coletor generacional que promove objetos com finalizadores sempre que eles tornam-se finalizáveis [56]. Por conseguinte, para implementar um cache mais efetivo basta encapsular os objetos com uma classe proxy que implementa finalizadores, e referenciá-las com referências fracas longas. Com isso o objeto só será considerado em uma coleta quando a disponibilidade de memória for menor.

No outro extremo, muitas vezes pode ser interessante bloquear completamente a coleta de um objeto referenciado enquanto a referência correspondente não for explicitamente limpa. Guardians [27] constituem um mecanismo criado para controlar a reciclagem e finalização de objetos (incluindo a ordem de invocação dos finalizadores) semelhante a referências fracas. Objetos registrados pelo programa cliente são automaticamente inseridos em uma fila logo após tornarem-se finalizáveis. Os finalizadores correspondentes são implicitamente invocados somente após os objetos serem retirados da fila pelo programa cliente[6]. As referências fantasmas de Java constituem um outro exemplo de referências fracas que demandam liberação explícita.

4.3.2 Desacoplamento e Ressurreição

Se uma linguagem suporta finalizadores, o coletor de lixo deve determinar quando um objeto se torna finalizável e quando o objeto foi finalizado (usualmente a memória ocupada por um objeto só é liberada após a execução do finalizador correspondente). Referências fracas podem ser limpas antes ou depois da finalização. No ambiente .Net, como já vimos, o programador precisa especificar explicitamente quando cada referência fraca deve ser limpa.

As regras para a limpeza de referências fracas são particularmente importantes quando referências fracas são usadas para a finalização de objetos. É importante definir, por exemplo, se o callback terá acesso ao objeto referenciado, podendo eventualmente ressuscitá-lo, como é comum na semântica de finalizadores baseados em classes. Modula-3 constitui um caso raro de uma linguagem com finalização baseada em referências fracas que mantém o acoplamento entre o objeto e o respectivo finalizador.

A ressurreição através de referências fracas pode acontecer através de callbacks que dão acesso ao objeto, exatamente como acontece com finalizadores baseados em classes, ou através da dereferenciação simples de um objeto fracamente conexo. Referências fracas entretanto evitam completamente a dinâmica de objetos zumbis, já que objetos que podem ressuscitar permanecem acessíveis ao programa cliente durante todo o tempo.

[6]Este mecanismo é semelhante a uma fila de notificação que insere os objetos fracamente referenciados na fila, ao invés das referências fracas.

4.3.3 Mecanismos de Notificação

Como vimos no capítulo anterior, a interface de referências fracas pode ser estendida com um mecanismo de notificação. Existem duas alternativas básicas para a implementação desta facilidade:

- No mecanismo de filas, assim que o coletor de lixo determina que um objeto fracamente referenciado é coletável, ou que a sua conectividade mudou, ele insere a referência associada na fila. Como solução alternativa, mas bem menos comum, o próprio objeto referenciado pode ser inserido na fila. Para obter informações sobre mudanças de conectividade de objetos o programa cliente precisa checar explicitamente o conteúdo da fila[7].

- No mecanismo de callback o coletor de lixo ao invés de inserir um objeto em uma fila, invoca o callback correspondente passando como parâmetro a referência fraca ou o próprio objeto referenciado. A referência fraca geralmente é limpa antes de ser passada como parâmetro, e pode ser usada como chave de busca em uma tabela externa que armazena informações associadas ao objeto coletado, ou ainda, como condição de execução de uma rotina qualquer.

Referências fracas estendidas com um mecanismo de callback permitem a definição de finalizadores por objetos, ao invés de por classes, e constituem o exemplo mais comum de finalização baseada em coleções. A execução destas rotinas contudo envolve as mesmas dificuldades associadas a finalizadores tradicionais, as quais já foram discutidas na Seção 4.2:

- Em linguagens que empregam coletores de lixo baseados em rastreamento, a invocação dos callbacks acontece de forma não-determinística.

- A execução de callbacks introduz linhas de execução concorrentes na aplicação.

- Geralmente não existem garantias quanto à ordem de invocação de callbacks.

[7]Observe que esta informação poderia ser obtida simplesmente checando-se todas as referências fracas no grupo de interesse. Isto no entanto implica em um custo de processamento bem mais alto.

A principal diferença entre finalizadores e callbacks associados a referências fracas é o grau de acoplamento existente entre o objeto a ser finalizado e a respectiva rotina de finalização. Com raras exceções, o callback não tem acesso ao objeto finalizado, o que evita atrasos no processo de coleta.

Filas de notificação permitem a implementação de mecanismos de finalização mais flexíveis, mas cuja execução deve ser escalonada de forma explícita. O programa cliente pode esperar por condições específicas, incluindo a finalização prévia de outros objetos, para só então invocar explicitamente a rotina de finalização de um objeto particular. Esta dinâmica evita os principais problemas de sincronização associados a callbacks e finalizadores, além de não atrasar desnecessariamente o processo de coleta.

Vale observar ainda que qualquer que seja o mecanismo de notificação adotado, a velocidade da notificação depende da implementação do coletor de lixo. Conforme já discutimos, mecanismos assíncronos podem ser inadequados para a implementação de certas funcionalidades.

Uma última questão associada a callbacks, importante sobretudo quando referências fracas são usadas para finalização de objetos, envolve a dinâmica de coleta de referências fracas para as quais foram registrados callbacks. O que acontece se a referência fraca torna-se inacessível antes do objeto referenciado ser coletado? As especificações de várias linguagens são vagas a esse respeito. Em Glasgow Haskell a associação de um callback com uma referência fraca impede a coleta da referência fraca até que o callback seja executado.

Para garantir que referências fracas com callbacks ativos sejam preservadas, basta que o sistema utilize uma estrutura de dados estática, de preferência visível para o programa cliente[8]. Sempre que um callback é registrado, a referência fraca correspondente é inserida na estrutura considerada. Se esta estrutura for visível, para cancelar ou modificar o callback associado a qualquer objeto o programa cliente deve simplesmente acessar a referência fraca correspondente.

4.3.4 Tipos Fracos

Referências fracas não precisam necessariamente ser implementadas através de um tipo específico, e muitas linguagens dão suporte a mecanismos que permitem marcar referências ordinárias como fracas (por exemplo, Perl e GNU Smalltalk),

[8]Se as referências fracas não puderem ser acessadas pelo programa cliente, essa solução cria um problema de objetos zumbis idêntico ao que acontece com finalizadores baseados em classes.

ou oferecem tipos que são implicitamente dereferenciados (Python). Apesar desta semântica ser atraente em um certo sentido, ela pode induzir facilmente a erros críticos de programação, como ilustramos na Seção 4.1. Para garantir a correção de um programa acaba sendo necessário modificar o seu código de tal forma que este torna-se bem menos inteligível.

4.3.5 Coleções Fracas

Qualquer uma das estruturas de dados tipicamente encontradas em linguagens de programação pode ser estendida com uma versão fraca. Porém, tabelas e conjuntos fracos são as extensões mais comuns, e requerem a colaboração explícita do coletor de lixo para serem implementadas de forma mais eficiente.

Conjuntos fracos não interferem na conectividade de seus elementos, os quais são armazenados apenas enquanto o coletor não tiver determinado que eles são coletáveis. Pelo menos dois métodos devem ser disponibilizados para manipular conjuntos fracos: um para adicionar elementos, e outro para remover todos os elementos.

Tabelas fracas podem ser definidas com qualquer combinação de chaves fracas e valores fracos, mas tabelas nas quais a chave é fraca, e os valores são mantidos por referências ordinárias, constituem-se na versão mais comum. Geralmente as entradas de uma tabela fraca são implicitamente removidas quando uma das suas referências fracas é limpa.

Idealmente, tabelas fracas deveriam evitar a coleta de objetos apenas enquanto estes pudessem ser acessados normalmente, ou seja, usando diretamente chaves de busca (outros mecanismos de acesso a tabelas, como por exemplo, iteradores, poderiam ser considerados como secundários, não impedindo a coleta de elementos). Contudo, a maioria das implementações não se comporta desta forma. Sempre que existe um ciclo interno à tabela (como quando um elemento tem como chave um objeto referenciado pelo valor do outro elemento e vice-versa), ou um ciclo entre elementos de diferentes tabelas fracas, os elementos que compõem o ciclo tornam-se não coletáveis.

Considere a tabela fraca apresentada na figura seguinte, onde os objetos A e B não são acessíveis a partir do conjunto-raiz. Suponha ainda que a única forma de acessar os elementos desta tabela é através de uma função que recebe como parâmetro a chave de busca. Em função do ciclo interno existente entre os elementos 1 e 3, os objetos referenciados por estes elementos não serão coletados.

	chave	valor
1	*A*	*B*
2	C	D
3	*B*	*A*
4	D	E

Figura 4.1: Uma tabela fraca com valores fortes.

Ainda que os objetos fracamente referenciados não possam ser considerados desconexos no sentido tradicional, na prática eles são, e portanto seria conveniente se o coletor de lixo fosse capaz de lidar adequadamente com tal tipo de situação.

Mesmo quando não existem ciclos, a remoção de elementos com campos nulos (que foram limpos) pode levar mais tempo do que o esperado. Considere uma tabela com chaves fracas em que um elemento possui como valor um objeto que é usado como chave de um segundo elemento, como os elementos 2 e 4 na Figura 4.1. Geralmente o coletor de lixo só vai ser capaz de remover este segundo elemento em um ciclo de processamento posterior aquele em que foi removido o primeiro elemento (no nosso exemplo, o elemento 4 só vai ser removido depois que o elemento 2 for removido). Uma tabela fraca com um encadeamento de n elementos levará pelo menos n ciclos para ser completamente limpa. Isto pode até ser útil para criar um cache, mas por outro lado pode levar ao esgotamento da memória apesar de ainda existirem objetos coletáveis que não foram coletados. Tabelas Lua por exemplo comportam-se desta maneira. Em Java a situação é ainda mais complicada. O coletor demanda no mínimo n ciclos para efetuar a coleta, mas além disso é necessário que entre cada ciclo de coleta a tabela seja explicitamente acessada pelo programa cliente (lembre que a implementação de tabelas fracas em Java usa uma fila de notificação).

Uma solução interessante para esses problemas, apresentada originalmente por Hayes [35], é a substituição de elementos em tabelas fracas por pares (chave/valor) especiais denominados *ephemerons*. O valor de um ephemeron pode ser visto como uma referência semi-fraca. Ele se comporta como uma referência fraca quando refere-se a objetos que são referenciados por outros elementos de uma tabela fraca, e comporta-se como uma referência forte quando refere-se a objetos fora da tabela. Assim qualquer objeto que não puder ser alcançado de fora da tabela pode ser eventualmente coletado.

Glasgow Haskell, conforme discutido no Capítulo 3, define uma semântica para referências fracas baseada em pares chave/valor que é semelhante a ephemerons,

evitando também o problema de referências internas cíclicas. A principal diferença entre esta implementação e a definição original de ephemerons diz respeito a como estes pares são coletados. Em Haskell o valor é considerado acessível se a chave for acessível, mas a conectividade da chave não tem qualquer relação com a conectividade do par chave/valor. Já o valor referenciado por um ephemeron, segundo a definição original apresentada por Hayes, só é acessível se o ephemeron for acessível a partir do conjunto-raiz, e a chave também.

4.4 Referências Fracas e Finalização

Filas de notificação e callbacks associados a referências fracas constituem um mecanismo alternativo de finalização. Callbacks em particular, quando recebem como único parâmetro a referência fraca recém-limpa, apresentam duas vantagens importantes em relação a finalizadores baseados em classe:

- O objeto finalizável é desacoplado da rotina de finalização, evitando atrasos na reciclagem da memória.
- O objeto finalizável não deixa de ser acessível antes de ser finalizado, garantindo um maior controle da aplicação.

Filas de notificação oferecem estes mesmos benefícios, e ainda evitam os principais problemas de sincronização associados à execução implícita de finalizadores.

Apesar dessas vantagens, referências fracas simples, estendidas com mecanismos de notificação com desacoplamento, não são suficientemente expressivas para tornar o suporte a finalizadores tradicionais completamente desnecessário. Essa limitação decorre de duas razões principais:

- Não faz muito sentido definir um finalizador que seja totalmente independente do estado do objeto a ser finalizado. Desse modo rotinas de finalização normalmente dependem de informações armazenadas pelos objetos finalizados. A finalização através de referências fracas implica porém no uso de uma estrutura externa ao objeto para armazenar as informações necessárias a sua finalização, o que em linguagens orientadas a objetos, fere o princípio básico de encapsulamento de dados.
- Em alguns casos é essencial que o finalizador tenha acesso ao objeto finalizado. Isso acontece quando a finalização é intrínseca ao objeto (por exemplo, a destruição explícita de ciclos de referências), ou quando o finalizador tem

como uma de suas responsabilidades definir se o objeto finalizado precisa ser coletado ou não (objetos que não devem ser coletados são ressuscitados)[9].

O problema de quebra de encapsulamento pode ser atenuado, mas não tem como ser inteiramente evitado. Contudo, em linguagens não orientadas a objetos esse problema não representa uma limitação tão relevante, e a finalização baseada em coleções, mesmo que com desacoplamento, pode ser considerada como um mecanismo natural para gerenciar o ciclo de vida de objetos.

Conforme discutimos na Seção 1.2, finalizadores proporcionam uma solução elegante para controlar a coleta de objetos. Referências fracas ordinárias a princípio não permitem replicar esse tipo de dinâmica, mas algumas extensões simples tornam possível tratar os casos mais comuns. Se o controle da coleta de objetos for baseado na disponibilidade de memória, basta introduzir diferentes gradações de referências fracas. Caches implementados com referências suaves atendem perfeitamente a este tipo de requisito: preservam o objeto referenciado apenas enquanto a memória disponível for abundante. Se o controle da coleta de objetos for baseado em um critério mais específico, a linguagem pode ser estendida para permitir a parametrização do critério de limpeza de referências fracas. SmartEiffel por exemplo oferece suporte a um tipo de referência fraca cuja limpeza é determinada através de uma função especificada dinamicamente pelo programador (ver a Seção 3.2.2).

Quando o mecanismo de notificação mantém o acoplamento com o objeto fracamente referenciado, as diferenças em relação a finalizadores tradicionais praticamente desaparecem. Não é difícil perceber que callbacks que recebem como parâmetro o objeto fracamente referenciado são equivalentes a finalizadores tradicionais (iremos explorar esse ponto formalmente na Seção 5.5), e incorrem nos mesmos problemas inerentes a estes. Filas que permitem acesso ao objeto referenciado, apesar de também manterem um maior acoplamento, constituem uma alternativa interessante ao garantir ao programador um controle maior sobre a dinâmica de coleta e finalização. Neste caso porém é necessário um cuidado especial para evitar um escalonamento inadequado das rotinas de finalização, o que pode levar a um esgotamento prematuro da memória disponível.

Um último ponto que precisamos discutir aqui diz respeito a interação de finalizadores e referências fracas em sistemas que oferecem suporte a estas duas

[9] A finalização intrínseca a estrutura de objetos é um caso patológico relacionado a coletores que utilizam contagem de referências. Não conhecemos nenhum outro exemplo que dependa deste tipo de solução.

abstrações. Quando deve ser efetuada a limpeza de uma referência fraca associada a um objeto finalizável[10]? Na verdade não existe nenhuma grande vantagem em limpar uma referência fraca antes ou depois da finalização, e a diferença entre estas duas opções é sutil. Como um objeto só pode ser coletado se não estiver associado a um finalizador habilitado, a limpeza após a finalização garante acesso ao objeto através da referência fraca até ele tornar-se efetivamente coletável. Por outro lado, essa opção permite a ressurreição de objetos finalizados, algo que nem sempre é adequado. C#, como vimos, oferece uma solução interessante para essa questão: permite ao programador definir dinamicamente qualquer uma das alternativas mencionadas.

4.5 Considerações Finais

Conforme discutimos, alguns dos problemas típicos associados à finalização automática podem ser evitados através do uso de um mecanismo de notificação acoplado a referências fracas. Esta facilidade constitui uma alternativa interessante de finalização sobretudo em aplicações nas quais um alto nível de desempenho é um requisito fundamental. Filas de notificação, em especial, evitam problemas de sincronização, mas podem ser facilmente implementadas usando callbacks. O inverso só é verdadeiro em linguagens que oferecem suporte a mecanismos de concorrência suficientemente expressivos.

A representação de referências fracas através de tipos específicos, além de evitar erros associados a mudanças na conectividade de objetos, oferece uma interface mais simples, que pode ser naturalmente estendida para implementar referências com diferentes gradações de força. Considerando-se que a implementação de caches de memória constitui um problema relativamente comum, que pode ser facilmente tratado com referências suaves, acreditamos que linguagens de programação devem oferecer suporte a referências fracas com ao menos duas gradações distintas de força. Adicionalmente, referências fracas cuja limpeza pode ser controlada explicitamente pelo programador (vide Eiffel), também representam uma facilidade interessante, principalmente como alternativa a finalizadores tradicionais.

Para completar o suporte a referências fracas é recomendável que a linguagem ofereça tanto conjuntos quanto tabelas fracas. As demais coleções fracas são menos úteis na prática, e podem ser facilmente implementadas usando referências fracas

[10] A invocação do callback associado a referência fraca usualmente acontece apenas após o processo de limpeza.

simples.

Para permitir a finalização automática de objetos sem ferir o princípio básico de encapsulamento, o suporte a finalizadores tradicionais faz-se necessário mesmo em linguagens orientadas a objetos que contam com um amplo suporte a referências fracas. Para minimizar os problemas inerentes à execução concorrente de finalizadores, alguns cuidados devem ser observados. Como em muitos casos a execução dessas rotinas precisa ser sincronizada, a linguagem deve prover mecanismos adequados para tal fim. Além disso, para reduzir a necessidade de sincronização e minimizar os problemas decorrentes deste processo, finalizadores devem ser executados em threads dedicados, que não detenham locks, e a execução de métodos sincronizados deve impedir a coleta (ou ao menos a finalização) prematura de objetos (o que evita condições de corrida inesperadas).

Durante a execução de uma aplicação, quando necessário, o programa cliente pode controlar a ordem de invocação de finalizadores diretamente através do uso de uma estrutura de dados estática. Portanto o suporte automático à invocação ordenada de finalizadores geralmente introduz um custo de processamento adicional desnecessário. Uma exceção importante acontece em sistemas que garantem a execução de finalizadores ao término da aplicação. Não faz muito sentido invocar um finalizador se todos os objetos que representam recursos externos já tiverem sido coletados ou finalizados. Assim é fundamental que ao término da aplicação os finalizadores sejam sempre executados em uma ordem pré-determinada. Por outro lado, a invocação de finalizadores nesta fase da aplicação é uma facilidade problemática, cuja ausência não restringe o uso geral da linguagem.