2
Fundamentos

Para um melhor entendimento do nosso trabalho, esse capítulo descreve os principais conceitos abordados. A seção 2.1 revisa brevemente as técnicas usadas na implementação de coletores de lixo. Uma descrição mais detalhada dessas técnicas e dos algoritmos de coleta de lixo pode ser encontrada nos trabalhos de Jones e Lins (Jones96) e Wilson (Wilson92). A seção 2.1 também descreve os mecanismos de finalizadores e referências fracas. A seguir, na seção 2.2 discutimos o uso de mecanismos de notificação acoplados a referências fracas para implementar um mecanismo alternativo de finalização. E por fim, na seção 2.3, apresentamos uma estrutura de dados denominada tabela fraca e o problema de ciclos inerente à mesma.

2.1
Coleta de Lixo

Coleta de lixo (Jones96, Wilson92) é uma forma automática de realizar gerenciamento de memória. Com este recurso, é possível recuperar zonas de memória que o programa não utiliza mais. Enquanto em muitos sistemas os programadores devem gerenciar explicitamente a memória, utilizando, por exemplo, as funções `malloc` e `free` na linguagem C (Kernighan88) ou os operadores `NEW` e `DISPOSE` em Pascal (Jensen91), sistemas com coletores de lixo livram o programador desse fardo. Uma porção contígua de memória, onde um determinado conjunto de dados está alocado, é chamada de *nó*, *célula* ou *objeto*[1]. A função do coletor de lixo é achar objetos que não são mais usados e tornar seu espaço de armazenamento disponível para reuso pelo programa em execução. Na maioria dos casos, um objeto é considerado lixo quando não existem mais referências para ele. O coletor de lixo irá então marcar para desalocação a área de memória onde reside tal objeto. Objetos *vivos* (que podem ser acessados pelo programa) são preservados pelo coletor, assegurando que o programa nunca terá uma referência para um objeto desalocado. O funcionamento básico de um coletor de lixo consiste de duas fases[2]:

[1]Esse último é mais usado num contexto orientados a objetos. Optamos por adotar esse termo ao longo do trabalho.

[2]Na prática, essas duas fases podem ser intercaladas.

1. Distinguir de alguma forma o lixo dos objetos vivos (*detecção de lixo*);

2. Desalocar a memória ocupada pelos objetos considerados lixo para que o programa possa reusá-la (*desalocação de memória*).

A coleta de lixo automática elimina a grande quantidade de erros que podem ser introduzidos pelo programador quando esse deve gerenciar explicitamente a memória. Além disso, permite uma programação mais modular, já que um módulo não precisa saber se um objeto está sendo referenciado por outros módulos para que possa liberar, ou não, a memória ocupada por ele explicitamente. Devido ao custo de programação inerente ao gerenciamento explícito de memória, sistemas sem coleta de lixo automática são mais difíceis de desenvolver e manter. Não desalocar a memória no momento correto pode acarretar erros como *memory leak* (vazamento de memória) ou *dangling pointers*. Vazamento de memória ocorre quando objetos não mais necessários ao programa em execução são acumulados na memória, acarretando um desperdício de espaço. Já dangling pointers são caracterizados por uma coleta prematura do objeto, ou seja, um objeto que ainda pode ser acessado pelo programa é removido. Isso acaba por tornar a referência para esse objeto um dangling pointer, um ponteiro direcionado a uma posição inválida.

O gerenciamento automático de memória aumenta o nível de abstração de uma linguagem de programação sem necessariamente modificar a sua semântica básica. Entretanto, por vezes pode ser interessante fornecer ao programa cliente uma interface com o coletor de lixo a fim de que o primeiro possa fornecer informações ao coletor, alterando seu comportamento, ou simplesmente obter informações geradas a partir da execução do último. Buscando, portanto, uma maior flexibilidade, muitas linguagens oferecem mecanismos que possibilitam a interação dinâmica entre programas clientes e o coletor de lixo. Tipicamente, esses mecanismos incluem *finalizadores*, *referências fracas*, descritos em mais detalhes nas sessões 2.1.2 e 2.1.3 respectivamente, e métodos para a invocação explícita e parametrização dinâmica do coletor de lixo.

2.1.1 Técnicas de Coleta de Lixo

Na prática, a detecção de objetos coletáveis pode ser feita através de duas técnicas: *contagem de referências* ou *rastreamento*. Em sistemas que utilizam o técnica de contagem de referências (Collins60), cada objeto alocado na memória possui associado a si um contador de referências (ou ponteiros). Cada vez que uma referência para um objeto é criada, o contador desse objeto é incrementado. Quando uma referência existente é eliminada, o contador é

decrementado. A memória ocupada por um objeto pode ser reciclada quando seu contador for igual a zero, já que isso indica que não existem referências para esse objeto e o programa não pode mais acessá-lo. A coleta de um objeto pode levar a transitivos decrementos em contadores de outros objetos e, consequentemente, a novas remoções, pois referências vindas de um objeto considerado lixo não devem ser contadas como referências. O ajuste e a verificação dos contadores correspondem à fase de detecção de lixo. A fase de desalocação de memória ocorre quando o contador atinge zero. O algoritmo de coleta de lixo que utiliza essa técnica também é chamado de contagem de referências (Collins60). Variações e otimizações podem ser encontradas nos trabalhos de Deutsch (Deutsch76) e Wise (Wise77).

Wilson argumenta em seu trabalho (Wilson92) que a técnica de contagem de referências possui uma natureza incremental, pois as fases da coleta são intercaladas com a execução do programa, já que ocorrem sempre que uma referência é criada ou removida. E, devido a isso, a técnica de contagem de referências pode ser usada em sistemas de tempo real flexível (*soft real-time* (Liu00)). Contudo, essa técnica possui um grande problema: ela não é capaz de detectar ciclos. Se os ponteiros de um grupo de objeto criam um ciclo direcionado, então seus contadores nunca serão reduzidos a zero, mesmo que não exista um caminho do programa para os objetos.

Na técnica de rastreamento o grafo das relações entre os objetos (formado pelas referências e ponteiros entre os objetos) é percorrido, normalmente via busca em largura ou busca em profundidade. A varredura parte do que é chamado *conjunto raiz* que pode incluir variáveis globais alocadas estaticamente, variáveis em registradores, etc. Os objetos alcançados são marcados, seja alterando alguns bits nos próprios objetos ou gravando os mesmos em uma tabela especial. O algoritmo *mark-sweep* (McCarthy60) utiliza a técnica de rastreamento para detectar os objetos vivos. Uma vez que os objetos vivos se tornaram distinguíveis, inicia-se a fase de desalocação de memória. Nessa fase, a memória é percorrida a fim de achar todos os objetos não marcados e liberar o espaço que ocupam para reuso. Outros algoritmos que utilizam a técnica de rastreamento são *mark-compact* (Cohen83) e *stop-and-copy* (Cheney70, Fenichel69).

Devido ao uso da técnica de rastreamento, o algoritmo mark-sweep consegue eliminar o problema dos ciclos que existe no algoritmo de contagem de referências. Porém, um problema ainda persiste. Para ser realizada a coleta, o programa pára sua execução de tempos em tempos e inicia a fase de detecção de lixo. Somente quando a segunda fase da coleta estiver completa o programa volta à sua execução normal. Algoritmos de coleta de

lixo que apresentam essa característica são muitas vezes chamados de *stop-the-world* (Jones96, Boehm91) e não é possível utilizá-los em sistemas de tempo real flexível.

A coleta de lixo pode se beneficiar de algoritmos *incrementais* (Dijkstra78) para contornar esse problema. Os algoritmos incrementais permitem que a coleta de lixo seja realizada a curtos passos, intercalada com a execução da aplicação. Esses algoritmos podem reduzir o impacto da execução da coleta de lixo, já que o programa não precisa parar e esperar que a coleta termine. Isso possibilita o uso de coletores de lixo em sistemas de tempo real flexível. Os algoritmos incrementais também podem ser generalizados a fim de realizar coletas concorrentes que são executadas em outro processador e em paralelo com a execução do programa.

O Coletor de Lua e o Algoritmo Tricolor Marking

O coletor de lixo da linguagem Lua implementa um algoritmo incremental, *tricolor marking* (Dijkstra78). Esse algoritmo utiliza a técnica de rastreamento na fase de detecção de lixo, intercalando-a com a execução do programa.

No geral, o coletor de lixo pode ser descrito como um processo que percorre o grafo de objetos colorindo-os. No início da coleta, todos os objetos estão coloridos de branco. À medida que o coletor percorre o grafo de referência, ele vai colorindo os objetos encontrados de preto. Ao final da coleta, todos os objetos acessíveis ao programa devem estar coloridos de preto e os objetos brancos são removidos. No entanto, em algoritmos incrementais, as fases intermediárias do rastreamento são importantes, pois a execução do programa é intercalada com a execução do coletor. Além disso, o programa não pode modificar o grafo de referências de modo a tornar impossível para o coletor achar todos os objetos vivos. Para prevenir esse problema, o algoritmo de tricolor marking introduz uma terceira cor, cinza, para representar objetos que foram alcançados pelo rastreamento, mas seus descendentes ainda não foram percorridos. Ou seja, à medida que o rastreamento ocorre, os objetos são inicialmente coloridos de cinza. Quando as referências para seus descendentes são rastreadas, eles são coloridos de preto e os descendentes de cinza. A coleta termina quando não existirem mais objetos cinza. O coletor então remove todos os objetos coloridos de branco.

Na prática, existem operações em coletores de lixo incrementais que não podem ser intercaladas com a execução do programa. Em Lua, a coleta de lixo é dividida em quatro fases. A primeira é a fase de rastreamento, onde o coletor percorre o grafo de referências marcando os objetos. Essa fase é intercalada com a execução do programa. A segunda fase é a fase atômica, onde um conjunto

de operações de coleta deve ser executado em um único passo. A terceira, também incremental, é a fase de desalocação de memória, onde os objetos não marcados são removidos e a memória associada a eles é liberada. Por fim, na quarta fase, ocorre a invocação dos finalizadores. Assim como a fase anterior, essa última também é intercalada com a execução do programa.

2.1.2 Finalizadores

No conjunto das linguagens de programação que oferecem suporte a gerenciamento automático de memória, finalizadores são rotinas executadas automaticamente antes de um objeto ser coletado. Um conceito muitas vezes confundido com finalizadores é o de *destructors* (destrutores), típicos da linguagem C++ (Stroustrup97). Ao contrário dos finalizadores, destrutores são rotinas executadas de forma síncrona e chamadas quando o programa libera explicitamente um objeto. Como finalizadores são chamados automaticamente pelo coletor, não é possível determinar quando um finalizador específico será executado.

Finalizadores vêm sendo utilizados em várias atividades, incluindo no gerenciamento de caches de objetos e na liberação de recursos providos por servidores ou outros programas. Hayes (Hayes92) provê uma visão geral de vários mecanismos de finalização em diferentes linguagens.

Existem duas caracterizações básicas para os mecanismos de finalização associadas aos tipos abstratos de dados (Hayes97). Na *finalização baseada em classes*, com amplo suporte em linguagens orientadas a objetos, todas as instâncias de uma classe tornam-se finalizáveis (possuem uma rotina de finalização a ser executada antes de sua coleta) se esta classe implementar a interface padrão de finalização. Tal interface, geralmente descrita pela especificação da linguagem, define um método que é invocado automaticamente pelo coletor de lixo. Na *finalização baseada em coleções* (containers), objetos tornam-se finalizáveis quando são inseridos em alguma estrutura de dados especial reconhecida pelo coletor de lixo. Neste caso, a finalização não é inerente à classe ou ao tipo do objeto; finalizadores podem ser associados dinamicamente a instâncias específicas, o que permite que vários objetos, por vezes de tipos distintos, compartilhem um mesmo finalizador.

2.1.3 Referências Fracas

Referências fracas, também conhecidas como *ponteiros fracos* ou *soft pointers* (Hayes97), são um tipo especial de referência que não impede

que um objeto seja coletado pelo coletor de lixo. De acordo com Brownbridge (Brownbridge85), uma das motivações iniciais para o desenvolvimento e uso desse mecanismo foi a dificuldade associada à coleta de referências cíclicas em coletores de lixo que usam exclusivamente contagem de referências para detectar objetos não mais alcançáveis pelo programa. Nesses sistemas, o ciclo de referências pode ser quebrado substituindo-se algumas referências ordinárias por referências fracas de tal forma que qualquer ciclo de referências seja composto por pelo menos uma referência fraca. Mais recentemente porém, com a ampla adoção de coletores de lixo baseados em rastreamento, o emprego de referências fracas passou a ser motivado sobretudo por constituir-se em uma opção elegante para que aplicações exerçam um nível maior de controle sobre a dinâmica de desalocação de memória.

Para que o programa possa obter mais informações sobre mudanças de conectividade de objetos, é possível estender a interface de referências fracas com um mecanismo de notificação. Existem duas alternativas para a implementação desta facilidade:

- Na notificação passiva, cada notificação é de alguma forma armazenada e o programa cliente precisa checar explicitamente as notificações existentes. Essa alternativa é implementada, na maioria dos casos, através de um 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. Para obter informações sobre mudanças de conectividade de objetos, o programa cliente verifica explicitamente o conteúdo da fila.
- Na notificação ativa, ao invés de inserir um objeto em uma fila, o coletor de lixo associa a cada referência fraca um callback, que será invocado assim que o coletor determinar que o objeto referenciado tornou-se inacessível. As linguagens Python (Rossum06) e Modula-3 (Homing93, Modula07) implementam callbacks que oferecem um suporte alternativo à finalização baseada em coleções. Ao invocar um callback, pode-se passar como parâmetro um objeto que represente a referência fraca ou o próprio objeto referenciado.

2.2 Referências Fracas e Finalizadores

Referências fracas simples podem ser estendidas a fim de constituir um mecanismo alternativo de finalização. Elas podem ser utilizadas para informar ao programa cliente que um objeto foi coletado, eventualmente disparando

automaticamente rotinas associadas a tal evento. Essa dinâmica constitui um exemplo do mecanismo de finalização baseado em coleções, e evita alguns problemas associados a finalização baseada em classes. Como finalizadores baseados em classes geralmente têm acesso irrestrito aos objetos aos quais estão associados, as informações usadas pela rotina de finalização podem estar armazenadas no próprio objeto finalizável. Isso ocasiona atrasos na reciclagem de memória, pois o objeto finalizável deve permanecer armazenado até que sua finalização esteja completa. Outro problema ocorre no intervalo de tempo entre o objeto ser marcado para finalização e ele ser efetivamente coletado. Nesse intervalo, o objeto torna-se inacessível, contudo, ainda pode influenciar no comportamento da aplicação.

Callbacks, quando recebem a referência fraca recém limpa como parâmetro, apresentam vantagens importantes em relação a finalizadores baseados em classes. Como o objeto finalizável é desacoplado da rotina de finalização, os atrasos na reciclagem de memória são evitados. Além disso, o objeto finalizável continua acessível antes de ser efetivamente removido, fornecendo à aplicação um maior controle sobre a coleta. Apesar dessas vantagens, callbacks apresentam algumas das dificuldades também associadas a finalizadores tradicionais:

- Em sistemas que utilizam coletores de lixo baseados em rastreamento, a invocação de callbacks acontece de forma não-determinística.
- Em algumas implementações, não existem garantias quanto à ordem de invocação dos callbacks.
- A execução de callbacks pode introduzir linhas de execução concorrentes na aplicação.

Filas de notificação, por outro lado, podem ser usadas para implementar mecanismos de finalização bem mais complexos e flexíveis. No entanto, por se tratar de um tipo de notificação passiva, sua execução deve ser escalonada de forma explícita, ou seja, a rotina de finalização não é automaticamente executada. O programa cliente pode esperar por condições específicas para só então invocar a rotina de finalização de um objeto em particular. Esta dinâmica evita os principais problemas de concorrência e sincronização relacionados a callbacks e finalizadores.

De acordo com a literatura mais recente (Leal05), referências fracas simples, estendidas com mecanismos de notificação com desacoplamento, não são suficientemente expressivas para tornar o suporte a finalizadores tradicionais completamente desnecessário. A decisão sobre como estender o mecanismo de referências fracas a fim de dar suporte a finalização depende do paradigma de

programação utilizado, do modo como é feita a detecção de objetos coletáveis e de como se dá o controle da coleta (por exemplo, baseado na disponibilidade de memória).

Neste trabalho, mostramos como é possível substituir finalizadores por referências fracas. Mais especificamente, discutimos o uso dos mecanismos de notificação passiva e notificação ativa e mostramos as vantagens e desvantagens de cada mecanismo nos diferentes contextos. Como ponto de partida, realizamos um amplo levantamento dos usos típicos de finalizadores e referências fracas, buscando identificar os problemas abordados por finalizadores e resolvendo cada um desses problemas através de referências fracas. Procuramos utilizar notificação passiva sempre que possível, pois acreditamos ser uma opção mais adequada que a notificação ativa. Iremos discutir isso em mais detalhes no Capítulo 3. Também estudamos em detalhes as implementações de finalizadores baseados em referências fracas a fim de apontar as vantagens e desvantagens de cada implementação, sempre levando em conta o contexto no qual está inserida a aplicação.

2.3 Tabelas Fracas

Outro problema interessante que tratamos neste trabalho diz respeito a ciclos em tabelas fracas. Tabelas fracas são constituídas de *pares fracos* (weak pairs) onde o primeiro elemento do par, a chave, é mantido por uma referência fraca e o segundo elemento, o valor, é mantido por uma referência comum. Na linguagem de programação Lua, uma tabela fraca pode ser criada como mostra a Listagem 1. Na implementação de Lua, também é possível criar tabelas fracas contendo apenas valores fracos ou contendo tanto chaves quanto valores fracos. Para isso, basta modificar o campo `__mode` da metatabela da tabela fraca: caso a string possua a letra 'k', as chaves são fracas; caso possua a letra 'v', os valores são fracos.

Listagem 1 Criando uma tabela fraca em Lua

```
a = {} -- tabela fraca
b = {}
setmetatable(a, b)
b.__mode = "k"
```

Tabelas fracas podem ser usadas para adicionar propriedades arbitrárias a um objeto qualquer, o que chamamos de *Tabelas de Propriedades*. O benefício de se utilizar uma tabela fraca para representar uma tabela de propriedades está no fato de que, na maioria dos casos, a adição de uma propriedade a

um objeto não irá modificar o instante em que ele deve ser coletado. Como exemplo, considere a tabela na Figura 2.1. Nessa tabela, cada chave possui uma referência fraca para um objeto e as propriedades extras desse objeto são referenciadas pelo respectivo valore através de uma referência ordinária. Caso as chaves não fossem fracas, o simples fato de inserir um par chave/valor na tabela iria garantir que ele nunca seria coletado. Quando as chaves são mantidas por referências fracas, uma entrada poderá ser coletada assim que o objeto referenciado pela chave não for mais usado pelo programa.

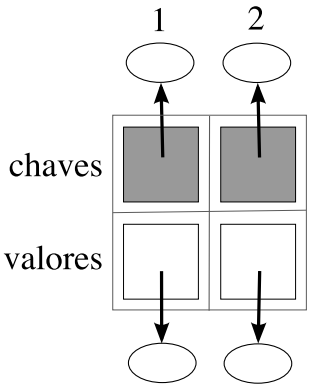

Figura 2.1: Uma tabela de propriedades

Idealmente, um objeto em uma tabela fraca, ou um par numa tabela de propriedades, deve ser mantido apenas enquanto sua chave puder ser acessada de algum lugar externo à tabela. Contudo, a maioria das implementações não se comporta desta forma. O problema ocorre quando o valor de alguma entrada da tabela contém uma referência direta ou indireta para uma chave, formando um ciclo interno à tabela ou um ciclo entre elementos de diferentes tabelas fracas. A tabela na figura 2.2 demonstra esse problema. Suponha 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 da auto-referência do elemento 1 (valor apontando para a própria chave) e do ciclo interno existente entre os elementos 2 e 3, os objetos referenciados por estes elementos não serão coletados. Mesmo quando não existem ciclos, a remoção de elementos pode levar mais tempo do que o esperado. Considere os elemento 4 e 5, onde o valor do elemento 4 contém uma referência para a chave do elemento 5. Geralmente, o coletor de lixo só vai ser capaz de remover este segundo elemento em um ciclo de processamento posterior àquele em que foi removido o elemento 4. Uma tabela fraca com um encadeamento de n elementos levará pelo menos n ciclos para ser completamente limpa. Tornar os valores fracos também não irá ajudar. Caso um objeto exista apenas como valor de uma tabela de propriedades, a

tabela deve manter a entrada, pois a chave correspondente pode não ter sido coletada e a busca pelo valor através da chave ainda é possível.

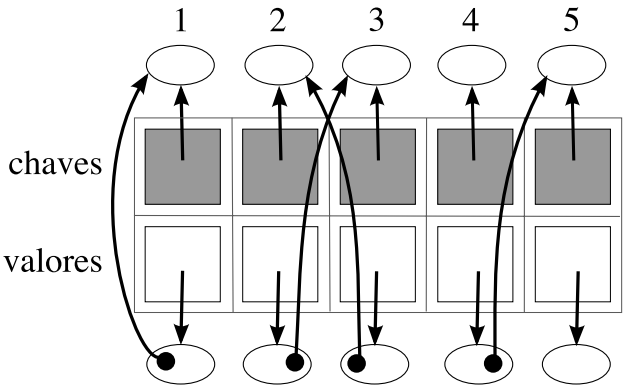

Figura 2.2: Alguns problemas com pares fracos

A implementação de tabelas fracas da linguagem Lua apresenta o problema de ciclos em tabelas fracas. Como exemplo, considere a Listagem 2. Como a primeira entrada da tabela `a` referencia a segunda e vice versa, nenhuma das entradas será coletada, mesmo depois de se tornarem inacessíveis.

Listagem 2 Criando um ciclo em uma tabela fraca de Lua

```
a = {} -- tabela fraca
b = {}
setmetatable(a, b)
b.__mode = "k"
c = {}
d = {}
a[c] = d
a[d] = c
c = nil
d = nil
```


2.3.1 Ephemerons

Uma solução interessante para esse problema, apresentada originalmente por Hayes (Hayes97), é a utilização de *ephemerons* ao invés de pares fracos. Ephemerons são um refinamento dos pares fracos chave/valor onde nem a chave nem o valor podem ser classificados como fraco ou forte. A conectividade da chave determina a conectividade do valor, porém, a conectividade do valor não influencia na conectividade da chave.

De acordo com Hayes, quando a coleta de lixo fornece suporte a ephemerons, ela ocorre em três fases ao invés de duas. A primeira fase percorre

o grafo das relações entre os objetos até encontrar um ephemeron. Quando isso ocorre, o coletor, no lugar de percorrer imediatamente os campos do ephemeron, o insere em uma lista para que possa ser processado futuramente. Os ephemerons dessa lista podem conter entradas acessíveis ou não.

Na segunda fase, o coletor percorre a lista de ephemerons. Qualquer ephemeron que possua uma chave que já tenho sido marcada como acessível pelo programa mantém o valor - se a chave é acessível então alguma parte do programa pode requisitar o valor. Qualquer ephemeron cuja chave não tenha sido marcada pode ou não conter entradas acessíveis. Esses ephemerons são recolocados na fila para inspeção futura. O primeiro grupo de ephemerons, os que possuem entradas acessíveis, são agora percorridos como qualquer outro objeto. Como a chave já foi marcada, e percorrida, somente o valor precisa ser percorrido. Porém, os valores podem conter referências para as chaves de ephemerons que continuam na fila, o que tornará as entradas nesses ephemerons alcançáveis. Sendo assim, eles precisam ser inspecionados novamente. Além do mais, outros ephemerons podem ser descobertos. Nesse caso eles são adicionados à fila.

Esse procedimento se repete até que a fila contenha apenas ephemerons cujas chaves não tenham sido marcadas. Esse conjunto de ephemerons armazena apenas entradas inacessíveis. O coletor pode então remover esses ephemerons, liberando a memória ocupada por eles. E finalmente, na terceira fase, o coletor percorre os objetos restantes. Os ephemerons encontrados nessa fase são tratados como objetos comuns e todos os campos são percorridos.

A linguagem Haskell define uma semântica para referências fracas baseada em pares chave/valor que é semelhante a ephemerons. O nosso trabalho mostra que também é possível incorporar ephemorons à implementação atual da linguagem Lua, bastando modificar um pouco o algoritmo de coleta de lixo. O Capítulo 5 mostra como ephemerons foram incorporados ao coletor de lixo de Lua, assim como analisa e compara seu comportamento para ephemerons e tabelas fracas.