Alexandra Barreto Assad de Barros

Finalizadores e Ciclos em Tabelas Fracas

DISSERTAÇÃO DE MESTRADO

DEPARTAMENTO DE INFORMÁTICA

Programa de Pós-Graduação em

Informática

Rio de Janeiro
Abril de 2007

Pontifícia Universidade Católica
do Rio de Janeiro

Alexandra Barreto Assad de Barros

Finalizadores e Ciclos em Tabelas Fracas

Dissertação de Mestrado

Dissertação apresentada como requisito parcial para obtenção do grau de Mestre pelo Programa de Pós–graduação em Mestrado em Informática do Departamento de Informática da PUC–Rio

Orientador: Prof. Roberto Ierusalimschy

Rio de Janeiro
Abril de 2007

Pontifícia Universidade Católica
do Rio de Janeiro

Alexandra Barreto Assad de Barros

Finalizadores e Ciclos em Tabelas Fracas

Dissertação apresentada como requisito parcial para obtenção do grau de Mestre pelo Programa de Pós–graduação em Mestrado em Informática do Departamento de Informática do Centro Técnico Científico da PUC–Rio. Aprovada pela Comissão Examinadora abaixo assinada.

Prof. Roberto Ierusalimschy
Orientador
Departamento de Informática — PUC–Rio

Prof. Noemi de la Rocque Rodriguez
Departamento de Informática — PUC–Rio

Prof. Renato Fontoura de Gusmão Cerqueira
Departamento de Informática — PUC–Rio

Prof. Luiz Henrique de Figueiredo
IMPA

Prof. José Eugenio Leal
Coordenador Setorial do Centro Técnico Científico — PUC–Rio

Rio de Janeiro, 13 de Abril de 2007

Alexandra Barreto Assad de Barros

Graduou–se em Ciência da Computação na Universidade Federal de Pernambuco.

Ficha Catalográfica

Barros, Alexandra Barreto Assad de

Finalizadores e Ciclos em Tabelas Fracas / Alexandra Barreto Assad de Barros; orientador: Roberto Ierusalimschy. — Rio de Janeiro : PUC–Rio, Departamento de Informática, 2007.

v., 75 f: il. ; 29,7 cm

1. Dissertação (mestrado) - Pontifícia Universidade Católica do Rio de Janeiro, Departamento de Informática.

Inclui referências bibliográficas.

1. Informática – Tese. 2. Linguagens de Programação. 3. Coleta de Lixo. 4. Finalizadores. 5. Referências Fracas. I. Ierusalimschy, Roberto. II. Pontifícia Universidade Católica do Rio de Janeiro. Departamento de Informática. III. Título.

CDD: 004

Agradecimentos

Aos meus pais e à minha irmã pelo constante apoio, carinho e compreensão e por todo amor e dedicação que recebi durante toda a vida.

Ao meu orientador Professor Roberto Ierusalimschy por ter me proporcionado um incrível aprendizado, pela confiança, pelas repreensões e por toda sua paciência, dedicação e sabedoria, e aos professores do Departamento de Informática com quem tive a honra de estudar, pelos seus ensinamentos.

À CAPES, à FAPERJ e à PUC–Rio, pelos auxílios concedidos, sem os quais este trabalho não poderia ter sido realizado.

Aos meus colegas da PUC–Rio, em especial aos amigos do LabLua, Fábio, Hisham e Sérgio, pelo companherismo e suporte técnico.

Aos meus queridos amigos e colegas de apartamento, Viviane, Ives, Ivana e Börje pelo apoio e pelas longas conversas.

Meus sinceros agradecimentos a todos que, direta ou indiretamente, contribuíram para este trabalho.

Resumo

Barros, Alexandra Barreto Assad de; Ierusalimschy, Roberto. **Finalizadores e Ciclos em Tabelas Fracas**. Rio de Janeiro, 2007. 75p. Dissertação de Mestrado — Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro.

Referências fracas e finalizadores constituem uma alternativa elegante para se obter controle sobre a interação entre a aplicação e o coletor de lixo. No entanto, em alguns contextos, finalizadores são desnecessários, pois é possível estender o mecanismo de referências fracas a fim de dar suporte a finalização. Neste trabalho, realizamos um estudo detalhado sobre os usos desses mecanismos e mostramos como é possível substituir finalizadores por referências fracas propondo uma implementação baseada em referências fraca para cada uso de finalizadores. Baseado nesse estudo, desenvolvemos um mecanismo de finalização via referências fracas para a linguagem Lua. Motivados por nossa proposta de uma maior exploração do mecanismo de referências, desenvolvemos um algoritmo para um importante problema relacionado a ciclos em tabelas fracas, uma estrutura criada a partir de referências fracas. A existência de referências cíclicas entre chaves e valores impede que os elementos que compõem o ciclo sejam coletados, mesmo que eles não sejam mais utilizados pelo programa. Isso acaba dificultando o uso de tabelas fracas em determinadas aplicações. A linguagem Haskell resolveu esse problema através de uma adaptação do mecanismo de ephemerons ao seu coletor de lixo. Partindo desse fato, modificamos a implementação do coletor de lixo de Lua para que este oferecesse suporte ao mecanismo de ephemerons. Dessa forma, pudemos eliminar o problema de ciclos em tabelas fracas nessa linguagem.

Palavras–chave

Linguagens de Programação. Coleta de Lixo. Finalizadores. Referências Fracas.

Abstract

Barros, Alexandra Barreto Assad de; Ierusalimschy, Roberto. **Finalizers and Cycles in Weak Tables**. Rio de Janeiro, 2007. 75p. MsC Thesis — Department of Informática, Pontifícia Universidade Católica do Rio de Janeiro.

Weak References and finalizers constitute an elegant alternative to obtain control over the interaction between the application and the garbage collector. However, in some contexts, finalizers are not necessary because it's possible to extend the weak reference mechanism in order to give support to finalization. In this work, we present a survey of the most common uses of these mechanisms. We also show how weak references can replace finalizers proposing a weak reference based implementation for each finalizer use. Based on this survey, we developed a finalization mechanism based on weak references for the Lua programming language.

Motivated by our proposal of a better exploration of the weak reference mechanism, we developed a solution for an important problem related to cycles on weak tables, an structure created using weak references. Cyclic references between keys and values prevents the elements inside the cycle from being collected, even if they are no more reachable. This ends up bringing difficulties to the use of weak tables in certain kinds of applications. The Haskell programming language solved this problem implementing an adaptation of a mechanism called ephemerons. Based on this fact, we modified the Lua garbage collector in order to offer support to ephemerons. As a result, we were able to solve the problem of cycles on weak tables in Lua.

Keywords

Programming Languages. Garbage Collection. Finalizers. Weak References.

Sumário

Lista de figuras

Lista de tabelas

Um dia virá em que todo meu movimento será criação, nascimento. Eu romperei todos os nãos que existem dentro de mim, provarei a mim mesma que nada há a temer, que tudo o que eu for será sempre onde haja uma mulher com meu princípio, erguerei dentro de mim o que sou um dia, a um gesto meu minhas vagas se levantarão poderosas, água pura submergindo a dúvida, a consciência, eu serei forte como a alma de um animal e quando eu falar serão palavras não pensadas e lentas, não levemente sentidas, não cheias de vontade de humanidade, não o passado corroendo o futuro! O que eu disser soará fatal e inteiro. Não haverá nenhum espaço dentro de mim para eu saber que existe o tempo, os homens, as dimensões, não haverá nenhum espaço dentro de mim para notar sequer que estarei criando instante por instante, não instante por instante; sempre fundido, porque então viverei, só então serei maior que na infância, serei brutal e mal feita como uma pedra, serei leve e vaga como o que se sente e não se entende, me ultrapassarei em ondas. Ah, Deus, e que tudo venha e caia sobre mim, até a compreensão de mim mesma em certos momentos brancos porque basta me cumprir e então nada impedirá o meu caminho até a morte sem medo de qualquer luta ou descanso me levantarei forte e bela como um cavalo novo.

Clarice Lispector, *Perto do Coração Selvagem.*

1
Introdução

Muitas linguagens de programação fornecem coleta de lixo a fim de desalocar automaticamente objetos inacessíveis. Os coletores de lixo livram o programador da responsabilidade de desalocar a memória, simplificando assim a implementação de algoritmos complexos e eliminando erros por falta de memória que são difíceis de detectar e corrigir. Ainda hoje, o gerenciamento dinâmico e automático de memória é uma área bastante ativa dentro da Ciência da Computação (Leal05). Essa constante busca por melhorias nos coletores de lixo se deve não só à necessidade de administrar a memória de forma mais eficiente, mas também à demanda de um desenvolvimento mais acelerado de sistemas de computação tradicionais.

Duas das ferramentas mais poderosas da engenharia de software são abstração e modularidade. O gerenciamento explícito de memória vai fortemente contra esses princípios (Jones96). Além disso, o trabalho realizado por Rovner (Rovner85) sugere que uma porção considerável do tempo de desenvolvimento pode ser gasta para resolver problemas relacionados ao gerenciamento explícito de memória, sobretudo à correção de erros. Ele estimou que 40% do tempo despendido no desenvolvimento do sistema Mesa foi dedicado à solução desses problemas. No entanto, coletores de lixo fornecem ao programador um aumento na abstração, ou seja, o programador não deve se preocupar com detalhes de gerenciamento de memória e podem focar sua atenção na implementação da real funcionalidade da aplicação. Além disso, coletores permitem 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.

Coletores de lixo realizam o trabalho de gerenciamento automático de memória nos bastidores, ou seja, o programa não toma conhecimento sobre as ações executadas pelo coletor. Dizemos então que o coletor de lixo age de forma transparente. No entanto, em várias situações as aplicações requerem um maior grau de flexibilidade, que não é possível em coletores totalmente transparentes. Nesse caso, é interessante permitir que o programa escrito pelo usuário interaja diretamente com o coletor de lixo, quer seja alterando o

seu comportamento, fornecendo a ele informações extras ou ainda obtendo informações geradas a partir de sua execução. Sendo assim, muitas linguagens fornecem funções e mecanismos auxiliares a fim de obter essa interação, ou seja, elas fornecem uma interface com o coletor de lixo. Tais interfaces são tipicamente representadas por *finalizadores* (Atkins88, Boehm03, Dybvig93, Hayes92) e *referências fracas* (Leal05).

Finalizadores são rotinas executadas automaticamente pelo coletor de lixo antes de liberar a memória ocupada por um objeto. Eles vêm sendo utilizados em várias atividades, incluindo no gerenciamento de caches de objetos e na liberação de recursos providos por servidores ou outros programas. Veremos na Seção 2.1.2 que finalizadores possuem uma natureza assíncrona. Devido a isso, Bloch (Bloch01) chega a dizer em seu trabalho que finalizadores são imprevisíveis, frequentemente perigosos e desnecessários e afetam negativamente o desempenho do programa. No entanto, Boehm (Boehm03) argumenta que o uso de finalizadores é essencial em alguns casos e seu assincronismo não deve necessariamente levar a programas não confiáveis.

Referências fracas são um tipo especial de referência que não impede que um objeto seja coletado pelo coletor de lixo. Muitas implementações de linguagens que utilizam coleta de lixo, ao menos desde a década de 80, apresentam algum suporte a referências fracas (Xerox85, Rees84). Dentre outras aplicações, elas podem ser empregadas como um mecanismo de finalização, mas evitando diversas dificuldades associadas a finalizadores tradicionais. Dependendo da linguagem de programação e da forma como é feito o controle da coleta de lixo, o suporte a finalizadores torna-se completamente desnecessário.

O trabalho apresentado por Leal (Leal05) procurou explorar os conceitos de finalizadores e referências fracas, suprindo a ausência de uma especificação clara e abrangente. Tomando como ponto de partida esse trabalho, apresentamos um estudo detalhado sobre como implementar mecanismos de finalização através de referências fracas. Com isso, podemos substituir finalizadores tradicionais por referências fracas e assim simplificar a implementação das linguagens de programação.

Com essa proposta de implementar finalizadores via referências fracas, estamos sugerindo um uso maior desse último mecanismo. Isso se deve ao fato de que iremos usar referências fracas não só nos usos típicos conhecidos desse mecanismo (apresentados no Capítulo 3), como também iremos usar referências fracas em cada um dos usos típicos de finalizadores encontrados. Dessa forma, estamos explorando mais as implementações de referências fracas presentes nas linguagens de programação. Contudo, a maioria das implementações de referências fracas apresenta o problema de ciclos em tabelas fracas.

Em muitos casos, queremos adicionar dinâmicamente propriedades a um objeto independente dos atributos de sua classe. Para isso, uma opção bastante comum é utilizar uma *tabela de propriedades* onde uma referência para o objeto é inserida como chave de busca e o respectivo valor armazena as propriedades extras. No entanto, se todas as referências na tabela de propriedades forem comuns, o simples fato inserir um novo par chave/valor garante que o objeto referenciado pela chave nunca será coletado. Para resolver esse problema o ideal é utilizar uma estrutura de dados chamada *tabelas fracas*, implementada através de referências fracas. Essa estrutura é constituída 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. Dessa forma, a adição de propriedades a um objeto não irá modificar o instante em que ele deve ser coletado

Porém, um grande problema com tabelas fracas ainda persiste na maioria das linguagens. A existência de referências cíclicas entre chaves e valores impede que os elementos que compõem o ciclo sejam coletados, mesmo que eles não sejam mais utilizados pelo programa. Isso acaba ocasionando um grande desperdício de memória, impossibilitando o uso de tabelas fracas. As linguagens Java (SUN04) e Lua (Ierusalimschy06), por exemplo, apresentam esse problema e, através da lista de discussão da linguagem Lua, pudemos observar que é motivo de reclamações constantes por parte dos programadores. Uma solução para o problema de ciclos em tabelas fracas foi encontrada pela linguagem Haskell (Haskell07). A implementação do mecanismo de referências fracas dessa linguagem possui uma adaptação do mecanismo de *ephemerons* apresentado por Hayes (Hayes97). Baseados no sucesso obtido em Haskell, realizamos uma adaptação do mesmo mecanismo para Lua, resolvendo o problema nessa linguagem.

1.1 Objetivos

As contribuições desta pesquisa se situam na área de Linguagens de Programação, mais especificamente no desenvolvimento de sistemas de coleta de lixo que fornecem suporte a finalizadores e referências fracas a fim de prover uma maior flexibilidade às aplicações. Inúmeras linguagens de programação fornecem algum tipo de suporte a ambos os mecanismos. Contudo, é possível utilizar referências fracas como um mecanismo alternativo de finalização, tornando desnecessário, em determinados contextos, o suporte a finalizadores e simplificando a linguagem.

Assim, o um dos objetivos deste trabalho é substituir finalizadores

tradicionais por um mecanismo de finalização baseado em referências fracas, seja via notificação passiva ou ativa. Realizamos um estudo detalhado de quais extensões são necessárias ao mecanismo de referências fracas a fim de dar suporte a finalização em diferentes contextos (linguagem de programação, modo de controle da coleta de lixo, etc). Com isso, analisamos as vantagens e desvantagens de utilizar referências fracas no lugar de finalizadores. Discutimos também a possibilidade de utilizar apenas notificação passiva e quais as vantagens desse mecanismo se comparado a notificação ativa. Além disso, implementamos um mecanismo de finalização baseado em referências fracas para a linguagem Lua (Ierusalimschy06).

Outro objetivo deste trabalho diz respeito ao problema de ciclos em tabelas fracas. Como ponto de partida, estudamos detalhadamente o funcionamento do mecanismo de ephemerons. Em seguida, estudamos o funcionamento do coletor de lixo de Lua a fim de estabelecer a melhor adaptação do mecanismo de ephemerons a ser implementada. Posteriormente, incorporamos esse mecanismo ao algoritmo de coleta de lixo de Lua, resolvendo assim o problema de ciclos nessa linguagem.

1.2 Organização

Esta dissertação está organizada da seguinte forma. Inicialmente, no Capítulo 2, descrevemos os conceitos necessários para um melhor entendimento do nosso trabalho. Com o intuito de identificar os problemas abordados por finalizadores e referências fracas, apresentamos no Capítulo 3 uma descrição dos principais usos desses mecanismos. Nesse capítulo, para cada uso típico de finalizadores encontrado, descrevemos uma solução baseada em referências fracas e discutimos a notificação passiva como a melhor alternativa em vários casos. No Capítulo 4, descrevemos um mecanismo de finalização baseado em referências fracas para a linguagem Lua. Também apresentamos nesse capítulo algumas linguagens de programação onde o mecanismo de finalização é implementado através de referências fracas e discutimos cada implementação. No Capítulo 5, mostramos como ephemerons foram adaptados à implementação do coletor de lixo de Lua para resolver o problema de ciclos em tabelas fracas. Por fim, no Capítulo 6, apresentamos as conclusões do nosso trabalho.

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.

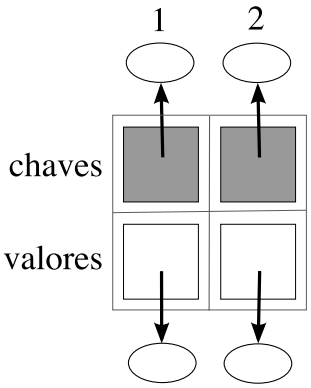

Figura 2.1: Uma tabela de propriedades

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.

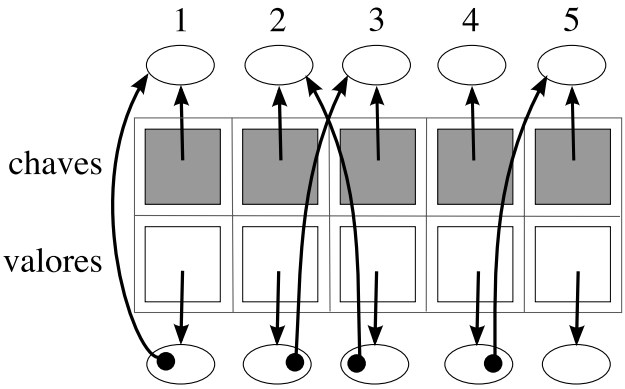

Figura 2.2: Alguns problemas com pares fracos

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.

3
Principais Usos de Referências Fracas e Finalizadores

Finalizadores e referências fracas recebem amplo suporte de algumas das principais linguagens de programação, entre elas C# (CLS07, Richter02), Java, Lua, Python, Perl (Schwartz05) e Haskell. No entanto, muitas vezes não precisamos de finalizadores quando a linguagem oferece suporte a referências fracas. Partindo desses fatos, decidimos investigar quais os usos típicos desses mecanismos através de uma pesquisa informal. Utilizamos como fonte de pesquisa a literatura, listas de discussões e dois mecanismos de busca em código na internet, Krugle (Krugle07) e Google Code Search (GCS07). Além disso, enviamos um questionário por e-mail a programadores tanto da área acadêmica quanto da indústria. Mais especificamente, enviamos o questionário para a lista de discussão da Linguagem Lua (LML07), que possui mais de 1.200 assinantes, e para a lista de discussão do laboratório Tecgraf da PUC-Rio (TCG07). Enviamos também mensagens individuais para 18 programadores da área acadêmica e 17 programadores da indústria. Uma parte do resultado de nossa pesquisa encontra-se na Tabela 3.1 e a outra parte refere-se aos usos encontrados para finalizadores e referências fracas que serão discutidos nas seções seguintes.

Ao analisar o resultado mostrado na Tabela 3.1, foi possível observar rapidamente que o mecanismo de referências fracas é pouco usado e pouco conhecido pela maioria dos programadores. Apenas 11% dos programadores que responderam a pesquisa já usaram referências fracas e menos da metade conhece esse mecanismo. Os número são mais generosos com relação a final-

	Lua	**Tecgraf**	**Academia**	**Indústria**
Total de programadores	> 1.200	63	18	17
Respostas obtidas	4	5	18	17
Conhecem finalizadores	4	5	18	17
Já usaram finalizadores	2	1	18	17
Conhecem referências fracas	4	5	4	5
Já usaram referências fracas	2	3	0	0

Tabela 3.1: Resultado da pesquisa informal.

izadores: 86% já usaram e todos conhecem esse mecanismo.

Com o objetivo de mostrar o alcance do mecanismo de referências fracas, realizamos algumas consultas no Google Code Search e no Krugle. Procuramos por programas Java que contivessem a palavra "String" a fim ter uma idéia do tamanho da base. Foram encontrados 1.430.000 resultados na primeira ferramenta e 1.805.649 na segunda. Ao buscar por "WeakReference" obtivemos 4.000 resultados na primeira e 2.727 na segunda, ou seja, apenas 0,28% e 0,15% da base respectivamente. Podemos então observar quão pouco esse mecanismo é utilizado. Já no caso de finalizadores, efetuamos uma busca por "finalize" em cada ferramenta, pois para implementar um finalizador em Java é necessário sobrescrever o método `finalize` da classe `java.land.Object`, da qual todas as classes são derivadas. Note que, como esse identificador não é uma palavra reservada nem um tipo específico da linguagem, ele pode ser usado em outros contextos que não dizem respeito a finalização de objetos. Devido a isso, retiramos uma amostra de 5% dos resultados para verificar onde a palavra "finalize" era usada. Pudemos então observar que em ambas as amostras, "finalize" se referia unicamente à implementação do método `finalize`. Voltando então à busca, obtivemos 14.500 resultados no Google Code Search e 13.328 no Krugle. Isso nos mostra que finalizadores têm um alcance quase quatro vezes maior que referências fracas de acordo com a primeira ferramenta e quase cinco vezes maior de acordo com a segunda.

Na próxima seção descrevemos os principais usos de referência fracas encontrados. A seguir, na Seção 3.2, além de descrever cada um dos usos típicos encontrados para finalizadores, mostramos como eles podem ser implementados através de referências fracas. Após catalogar as respostas dos programadores ao questionário enviado e realizar uma pesquisa bibliográfica em busca de usos de finalizadores e referências fracas, observamos que todos os usos encontrados em nossa pesquisa estavam descritos brevemente no trabalho de Leal (Leal05). No nosso trabalho, detalhamos mais cada um dos usos e as respectivas implementações. Além disso, apresentamos novos exemplos e casos especiais.

3.1 Referências Fracas

Nesta seção, descrevemos os usos encontrados para o mecanismo de referências fracas, são eles:

- **Coleta de referências cíclicas** – Encontrado na literatura, nos trabalhos de Brownbridge (Brownbridge85) e Leal (Leal05).
- **Cache** – Dentre os 5 programadores que disseram já ter usado referências fracas, dois o fizeram para implementar uma cache de objetos. Também

encontramos uma descrição desse uso no trabalho de Leal (Leal05).

- **Tabelas de propriedades** – Encontramos descrições desse uso nos trabalhos de Hayes (Hayes97) e Leal (Leal05).
- **Conjuntos fracos** – Dois programadores, dentre os 5 que disseram já ter usado referências fracas, disseram ter usado o mecanismo para implementar conjuntos fracos. O trabalho de Leal (Leal05) também apresenta uma descrição desse uso.
- **Finalização** – As linguagens Modula-3, Python e Haskell utilizam referências fracas na implementação de finalizadores. Também encontramos esse uso através do questionário (um programador disse já ter usado referências fracas como mecanismo alternativo de finalização, para desalocar recursos externos).

3.1.1 Coleta de Referências Cíclicas

Coletores de lixo que usam exclusivamente contagem de referências para detectar objetos não mais alcançáveis pelo programa não são capazes de coletar referências cíclicas, ou seja, não são capazes de coletar objetos que se referenciam mutuamente, gerando vazamento de memória. De acordo com Brownbridge (Brownbridge85), essa deficiência foi uma das motivações iniciais para o desenvolvimento e uso do mecanismo de referências fracas. A solução para o problema consiste em quebrar o ciclo de referências através da substituição de algumas referências ordinárias por referências fracas, de tal forma que qualquer ciclo de referências seja composto por pelo menos uma referência fraca. É responsabilidade do programador garantir essa quebra. Um exemplo disso encontra-se na Figura 3.1. O número ao lado de cada objeto corresponde ao seu contador de referências. O objeto A é referenciado de um ponto fora do ciclo e também pelo objeto C. Como essa última referência é ordinária, mesmo que a primeira referência seja removida o contador de A continuará maior que zero e nenhum objeto no ciclo será coletado. Ao substituir a referência ordinária de C para A por uma referência fraca, o contador de A chega a zero assim que a referência vinda de fora do ciclo é removida. Sendo assim, A será coletado e, consequentemente, os outros objetos.

3.1.2 Cache

Uma maneira interessante de gerenciar recursos externos é manter os objetos proxy que os representam na memória, ao invés de removê-los sempre

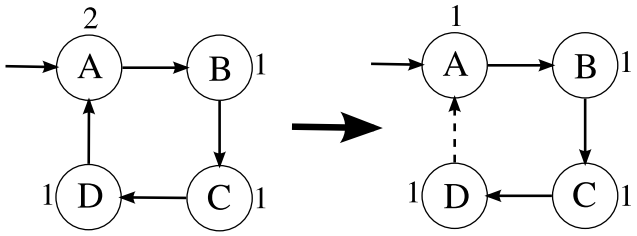

Figura 3.1: Coletando ciclos através de referências fracas

que não são mais necessários. Contudo, a existência de muitos recursos externos pode acabar sobrecarregando a memória, tornando interessante remover instâncias não utilizadas por um longo período. Além disso, pode ser necessário impedir que diferentes instâncias sejam mapeadas ao mesmo recurso externo, do contrário estaríamos criando uma dificuldade extra para manter uma sincronização entre essas instâncias. Essas funcionalidades podem ser oferecidas através de uma cache de instâncias e de uma função que retorne a instância do recurso desejado, ou crie uma nova caso ainda não exista uma instância para o recurso.

Podemos também utilizar uma cache de objetos em aplicações que fazem uso de estruturas de dados de tamanho significativo. Tais aplicações podem melhorar sensivelmente seu desempenho ao manter essas estruturas residentes na memória. Essa política contudo pode levar a um rápido esgotamento da memória livre. Uma cache pode ser usada para preservar os objetos apenas enquanto o nível de utilização da memória não for restritivo.

A dificuldade nessas duas abordagens está no fato de que uma entre duas coisas deve ser tolerada: ou a cache pode crescer indefinidamente ou existe alguma forma explícita de gerenciamento da cache na aplicação. Essa última pode ser bastante tediosa e levar a mais código do que é realmente necessário para resolver o problema. A primeira é inaceitável em processos com um longo tempo de execução ou mesmo em processos que fazem uso substancial da memória.

O uso de referências fracas possibilita a implementação de caches gerenciadas automaticamente. Considere como exemplo uma aplicação que utiliza uma tabela (um *result set*) com milhares de dados provenientes de um banco de dados. Após carregar os dados, a aplicação remove todas as referências para a tabela, exceto por uma referência fraca. Sempre que for necessário acessar algum dado na tabela, a aplicação primeiro verifica se a mesma ainda está acessível. Caso a tabela tenha sido coletada a aplicação recarrega os dados diretamente do banco de dados através de uma nova consulta.

Uma outra aplicação interessante de caches são funções memorizadas

(*memorizing functions*) (Ierusalimschy06, Jones00). O esforço computacional necessário para atender à invocação de uma função complexa pode ser reduzido salvando-se o resultado de cada invocação. Caso a função seja chamada novamente com o mesmo argumento, ela retorna o valor salvo. A utilização de um cache evita que a memorização dos resultados leve a um esgotamento da memória, preservando os resultados acessados mais recentemente (e possivelmente mais frequentemente).

Vale observar que em sistemas com coletores de lixo que usam contagem de referência os objetos são coletados imediatamente após sua contagem atingir zero, o que impede a implementação de caches usando-se apenas o mecanismo básico de referências fracas. Uma solução seria criar um segundo tipo de referência fraca que impediria a coleta do objeto enquanto a memória não atingisse níveis críticos.

3.1.3 Tabelas de Propriedades

Por várias razões, pode ser necessário associar atributos adicionais a um objeto sem modificar a sua estrutura interna. Em aplicações orientadas a objetos por exemplo, podemos querer associar dinamicamente um ou mais atributos a um objeto independente dos atributos de sua classe. Essa associação pode ser feita via uma estrutura de dados externa, como uma tabela associativa, que nesse caso é mais comumente chamada de tabela de propriedades. O objeto em questão deve então ser usado como chave nessa tabela, mapeando em seus atributos extras. Contudo, isso irá acarretar um problema: a referência na tabela impedirá que o objeto seja coletado. Vimos na Seção 2.3 que tabelas fracas podem ser usadas para representar uma tabela de propriedades, corrigindo o problema. Isso ocorre porque, ao invés de usar como chave de busca o próprio objeto, usamos uma referência fraca. Assim, quando o objeto não for mais usado pelo programa, a referência fraca poderá ser removida e o objeto poderá ser coletado.

Na linguagem Lua referências fracas são usadas através de tabelas fracas. O programador pode criar uma tabela onde apenas a chave é fraca, apenas o valor é fraco, ou ambos são fracos. Isso torna a implementação de tabelas de propriedades bastante simples.

3.1.4 Conjuntos Fracos

Conjuntos fracos (weak sets) podem ser entendidos como um conjunto de objetos cuja pertinência a esse conjunto não acarrete uma referência para o

objeto. Ou seja, sempre que objetos tem que ser processados como um grupo, mas sem interferir no ciclo de vida de cada um, podemos usar weak sets como uma solução de implementação.

Um exemplo bastante comum é o padrão de projeto *Observer* (Gamma95) que define uma dependência de um-para-vários entre um objeto observado e os objetos observadores. Quando o estado de um objeto observado se modifica, todos os seus observadores são notificados e tem o seu estado atualizado automaticamente. Geralmente, essa notificação é feita pelo objeto observado, que portanto precisa conhecer seus observadores, mantendo referências para os mesmos. O uso de referências fracas para mapear o conjunto de observadores permite manter um acoplamento mínimo entre os objetos e não impede que os observadores sejam coletados.

Note que, para implementar conjuntos fracos, caches e tabelas de propriedades, precisamos construir uma estrutura formada de referências fracas. No caso de conjuntos fracos, temos um conjunto de referências fracas que apontam para os observadores. A implementação de uma cache requer um conjunto de referências fracas para manter os objetos na memória. Por fim, no caso da tabela de propriedades, construímos uma tabela onde as chaves são mantidas por referências fracas. Caso a linguagem utilizada já forneça suporte a essas estruturas, a implementação desses usos é bastante simplificada. A linguagem Lua, por exemplo, fornece referências fracas através de tabelas fraca, como vimos na Seção 2.3. O programador pode criar uma tabela onde apenas a chave é fraca para representar uma tabela de propriedades, ou pode criar uma tabela com índices numéricos onde apenas os valores são fracos para representar um conjunto fraco ou uma cache.

3.1.5 Finalização

Referências fracas, acrescidas de um mecanismo de notificação, também podem ser utilizadas para informar o programa cliente que um objeto foi coletado, eventualmente disparando automaticamente rotinas associadas a tal evento. Como já foi visto, essa dinâmica constitui um exemplo do mecanismo de finalização baseado em coleções, e evita alguns dos problemas associados a finalizadores baseados em classes. Esse importante uso de referências fracas será abordado na próxima seção.

3.2

Finalizadores

De acordo com nossa pesquisa, o uso de finalizadores é bastante comum se comparado ao uso de referências fracas. Nesta seção, mostramos os usos encontrados para finalizadores tradicionais e fornecemos implementações alternativas que utilizam referências fracas. Os usos encontrados para finalizadores são:

- **Desalocação de Memória** – Esse uso foi primeiramente encontrado através do questionário enviado aos programadores. Em particular, quando programas em Lua fazem referência a bibliotecas em C, vimos que é bastante comum utilizar finalizadores para reciclar a memória alocada por essas bibliotecas. Posteriormente, constatamos que alguns trabalhos (Boehm03, Leal05) já haviam descrito esse uso. Boehm (Boehm03) o chama de *Legacy Libraries* e diz que talvez seja o uso mais comum de finalizadores. Nossa pesquisa também chegou a essa conclusão, pois 39% dos programadores que disseram já ter usado finalizadores o fizeram para desalocar memória.
- **Desalocação de Recursos Externos** – Esse uso foi encontrado através do questionário enviado aos programadores. Entre os programadores que disseram já ter usado finalizadores, 14% responderam que o fizeram para desalocar recursos externos. Existem dois casos particulares desse uso que foram encontrados no trabalho de Boehm (Boehm03), são eles: o uso de finalizadores para remoção de arquivos temporários e para obtenção de informações de conectividade[1].
- **Coleta de Referências Cíclicas** – A linguagem Perl usa finalizadores para coletar referências cíclicas (Christiansen03). Também encontramos uma descrição desse uso no trabalho de Leal (Leal05).
- **Fallback** – Encontrado apenas no trabalho de Leal (Leal05).
- **Reciclagem de Objetos** – Esse uso de finalizadores também foi encontrado apenas no trabalho de Leal (Leal05).

Como um de nossos objetivos, mostramos que finalizadores e callbacks possuem características semelhantes. Além disso, comparamos esses mecanismos com o mecanismo de notificação passiva. Contudo, antes de iniciar qualquer discussão sobre as implementações, gostaríamos de identificar características importantes da linguagem utilizada. Essas características podem influenciar na escolha da melhor abordagem.

[1]Como a memória pode ser vista como um recursos externo, o uso de finalizadores na desalocação de memória também pode ser considerado um caso particular de desalocação de recursos externos.

Uma das características mais importantes diz respeito aos mecanismos de concorrência e sincronização oferecidos pela linguagem. Como vimos na Seção 2.2, a execução de callbacks e finalizadores pode introduzir linhas de execução concorrentes na aplicação. O uso de mecanismos de sincronização associados a callbacks e finalizadores é importante e mesmo inevitável em muitas situações, sobretudo quando esses últimos acessam variáveis globais. Infelizmente, o uso de mecanismos de sincronização tende a deteriorar o desempenho da aplicação. Esse uso também é uma fonte constante de erros de programação, incluindo deadlocks. A falta de mecanismos de concorrência e sincronização suficientemente expressivos dificulta bastante a implementação de programas que usam finalizadores e callbacks. Além disso, a complexidade inerente à concorrência e a sincronização desses mecanismos pode levar até mesmo a erros de implementação da linguagem. Para evitar esse problema, é preferível usar um mecanismo de notificação passiva, onde o próprio programa é responsável por invocar a rotina de finalização de um objeto em particular. Outra característica importante do contexto refere-se ao algoritmo de coleta de lixo utilizado pelo coletor da linguagem. Existem duas abordagem básicas que devemos levar em consideração: contagem de referências e rastreamento. A primeira abordagem recicla a memória alocada para um objeto assim que esse não pode mais ser acessado pelo programa. A segunda abordagem não é determinística e pode haver uma demora até o objeto ser removido. Dependendo do algoritmo, devemos considerar extensões do mecanismo de referências fracas para não inviabilizar a implementação.

Outra característica a ser considerada é o paradigma da linguagem de programação utilizada. Independente do tipo de notificação, passiva ou ativa, a implementação de finalizadores através de referências fracas constitui uma finalização baseada em coleções onde a rotina de finalização é totalmente desacoplada do objeto. No entanto, finalizadores totalmente independentes do estado do objeto a ser finalizado não são muito úteis, pois as rotinas de finalização normalmente dependem de informações armazenadas pelos objetos finalizados. Para ter acesso a essas informações, uma implementação de finalizadores via referências fracas pode armazená-las numa estrutura de dados alternativa. Contudo, em linguagens orientadas a objetos isso significa uma quebra no encapsulamento. Um modo de atenuar esse problema é passar o objeto como parâmetro do callback, no caso da notificação ativa, ou armazenar o objeto na fila de notificações, no caso da notificação passiva. Porém, ainda assim dependemos da visibilidade das informações e da existência de métodos de acesso e controle. Por fim, a última característica que devemos considerar refere-se à garantia de execução dos finalizadores. Algumas linguagens de pro-

gramação não garantem que todos os finalizadores serão executados antes do término da aplicação. No caso de referência fracas, essa garantia pode não existir quanto à execução dos callbacks que implementam rotinas de finalização. Essa falta de garantia pode dificultar o uso de callbacks ou finalizadores para liberar recursos que não são liberados pelo sistema operacional ao término da aplicação, como veremos no caso de remoção de arquivos temporários.

3.2.1 Desalocação de Memória

Por vezes, é necessário que programas escritos em diferentes linguagens interajam. O trabalho de Muhammad (Muhammad06) discute uma série de questões de projeto ligadas à interoperabilidade de linguages de programação. Dentre as questões apresentadas, uma particularmente interessante é a diferença entre os modelos de gerenciamento de memória das linguagens de programação, mais especificamente, a interação entre linguagens com gerenciamento automático de memória e linguagens sem esse tipo de suporte. Um exemplo dessa interação ocorre quando programas escritos em Java (SUN04) fazem referência a bibliotecas escritas na linguagem C++, que não possui gerenciamento automático de memória. As funções dessas bibliotecas podem acabar alocando regiões de memória que não poderão ser liberadas automaticamente pelo coletor de Java. A solução é utilizar finalizadores para liberar a memória alocada com uma rotina nativa como `malloc`, ampliando assim a região de memória gerenciada pelo coletor de lixo.

A reciclagem da memória alocada por uma rotina nativa, escrita numa linguagem que não possui coleta de lixo automática, também pode ser feita através de um mecanismo de referências fracas que ofereça suporte a notificação ativa ou passiva. Para isso, o programa deve referenciar os objetos criados através das bibliotecas nativas via uma referência fraca. No caso de optarmos por notificação ativa, a rotina de finalização deve ser inserida em um callback associado à referência fraca. Essa implementação é bastante semelhante a finalizadores tradicionais, pois os callbacks também são executados automaticamente pelo coletor de lixo. Contudo, como vimos anteriormente, existem dois problemas quando utilizamos finalizadores ou callbacks para desalocar memória. O primeiro deles refere-se à possível introdução de linhas de execução concorrentes na aplicação. Caso a linguagem utilizada não ofereça um suporte adequado a concorrência e sincronização, o uso desses mecanismos pode levar a inconsistências durante a execução do programa. O segundo problema refere-se à impossibilidade de determinar quando finalizadores ou callbacks serão executados em coletores de lixo baseados em rastreamento.

Atrasos na liberação da memória podem ser gerados caso esses mecanismos não sejam executados no momento correto e esses atrasos podem afetar o desempenho da aplicação.

A outra abordagem que pode ser utilizada é a notificação passiva. Nesse caso, quando o coletor identifica que existem apenas referências fracas para o objeto que representa uma biblioteca em C ou C++, ele insere esse objeto em uma fila de notificações. Quando o programa achar adequado, por exemplo, quando o nível de memória ultrapassar certos limites, ele pode acessar explicitamente a fila de notificações e, a seguir, o programa pode executar a rotina de finalização adequada para cada objeto encontrado na fila. Apesar de perder um pouco de automação, o uso de notificação passiva elimina os problemas de concorrência. O indeterminismo ainda permanece, mas em menor escala. Isso se deve ao fato de que o coletor ainda é responsável por inserir o objeto na fila de notificações, após determinar que não existem mais referências ordinárias para ele. Contudo, ao contrário do que ocorre com finalizadores e callbacks, o próprio programa é responsável por invocar as rotinas de finalização e não o coletor de lixo.

3.2.2 Desalocação de Recursos Externos

Determinados recursos externos, como descritores de arquivos e conexões, são liberados pelo sistema operacional assim que o programa termina sua execução. Contudo, tais recursos não são infinitos e caso o programa não os libere a medida que não são mais necessários, pode haver um esgotamento dos mesmos. Geralmente, recursos externos são representados através de um objeto proxy, cujo ciclo de vida segue o padrão de utilização do recurso correspondente. Sendo assim, podemos alocar e desalocar automaticamente o recurso associado ao proxy através do uso de construtores e finalizadores. Para isso, basta inserir a rotina para alocação do recurso dentro do construtor e a rotina de desalocação dentro do finalizador referentes ao proxy, estendendo assim a capacidade de gerenciamento do coletor de lixo. Como sabemos, a rotina de finalização também pode ser implementada através de um callback associado a uma referência fraca para o proxy.

Assim como na seção anterior, finalizadores e callbacks nem sempre são adequados quando a linguagem utilizada possui um coletor de lixo baseado em rastreamento. Nesse caso, não é possível determinar quando a coleta será iniciada, consequentemente, também não é possível determinar quando os finalizadores serão executados e quando os recursos serão liberados. Atrasos na liberação dos recursos podem afetar negativamente o desempenho da aplicação.

Esses atrasos podem ser gerados quando os finalizadores não são executados no momento correto. Em alguns casos porém, é possível atenuar esse problema forçando explicitamente a execução do coletor de lixo, por exemplo, chamando o método `System.gc()` em Java. O coletor de lixo deve ser invocado sempre que a disponibilidade de recursos externos cair abaixo de níveis pré-definidos.

O problema de desalocação de recursos externos também pode ser abordado via um mecanismo de notificação passiva. Para isso, associamos uma referência fraca a cada proxy e quando a memória atingir níveis críticos o programa acessa então a fila de notificações. Contudo, como essa fila é preenchida automaticamente pelo coletor, é possível que nem todos os objetos finalizáveis estejam na fila de notificações. Sendo assim, caso a finalização dos objetos na fila não seja suficiente, pode ser necessário forçar explicitamente a execução do coletor para que os objetos finalizáveis que não se encontram na fila sejam inseridos nela. Essa abordagem pode ser vantajosa caso o acesso à fila de notificações seja feito logo após uma coleta, pois assim, todos os objetos finalizáveis estarão na fila. Porém, caso exista um longo intervalo de tempo entre uma coleta e um posterior acesso à fila de notificações, a utilização de finalizadores ou callbacks pode ser mais eficaz, pois os recursos já terão sido liberados enquanto que na notificação passiva o programa ainda precisará acessar a fila de notificações. Não devemos nos esquecer, no entanto, que o mecanismo de notificação passiva, ao contrário de finalizadores e callbacks, não apresenta problemas de concorrência e sincronização, o que o torna de fato mais vantajoso.

Remoção de Arquivos Temporários

Existem alguns recursos externos, como arquivos temporários criados pelo programa, que não são liberados pelo sistema operacional assim que o programa termina sua execução. Nesse caso, o próprio programa deve ser responsável pela remoção dos arquivos. Durante sua execução, o programa pode remover alguns arquivos temporários que não serão mais utilizados, porém, por vezes é necessário prover uma remoção mais confiável desses arquivos antes do término do programa. Finalizadores ou callbacks podem ser usados para tal fim, porém com algumas ressalvas. Dependendo da linguagem, não existe garantia de que esses mecanismos serão executados ao final do programa, consequentemente, alguns arquivos temporários poderão ainda permanecer.

A solução mais simples é garantir a execução dos finalizadores, ou dos callbacks, forçando uma coleta de lixo especial ao final do programa. Outra solução, descrita no trabalho de Boehm (Boehm03), sugere a implementação de uma rotina alternativa que deve ser chamada ao final da execução do programa,

enquanto finalizadores removem arquivos temporários durante a execução. No caso de descritores de arquivos e conexões, é irrelevante se a linguagem executa ou não os finalizadores ou os callbacks ao final do programa, pois o sistema operacional se encarrega de liberar os recursos.

Assim como ocorre quando usamos simplesmente finalizadores ou callbacks, o mecanismo de notificação passiva também não garante que todos os arquivos serão removidos ao final do programa. Sempre que necessário, podemos acessar a fila de notificações durante a execução do programa e remover os arquivos referentes aos objetos encontrados. Porém, quando acessamos a fila ao final de execução, o coletor pode não ter inserido todos os proxys que representam arquivos temporários na fila de notificações. Isso irá impossibilitar que todos os arquivos temporários sejam removidos.

Existe uma solução bastante simples, via referência fracas, que não requer a invocação de uma coleta de lixo especial ao final do programa. Essa solução consiste em manter os proxys em um conjunto fraco e utilizar notificação passiva ou ativa para remover os arquivos durante a execução do programa. À medida que os proxys são finalizados, suas respectivas referências contidas no conjunto fraco são removidas. Ao final da execução do programa, restam no conjunto fraco apenas os proxys referentes aos arquivos temporários que não foram removidos. Podemos então acessar esse conjunto e remover os arquivos temporários explicitamente.

Obtenção de Informações de Conectividade

Em alguns casos, a desalocação explícita de recursos externos pode ser uma tarefa bastante complexa e ineficiente. Contudo, podemos nos beneficiar das informações sobre a conectividade dos objetos fornecidas pelo coletor de lixo a fim de facilitar o trabalho de liberação de recursos.

O trabalho de Boehm (Boehm03) descreve uma aplicação que usa grafos acíclicos direcionados (DAGs) contendo descritores de arquivos em seus nós terminais. Em função do tamanho e complexidade das estruturas de dados utilizadas, bem como do padrão de acesso da aplicação, é extremamente difícil rastrear explicitamente todas as referências para cada descritor de arquivos, a fim de fechá-los explicitamente quando necessário. A solução encontrada foi associar finalizadores aos nós terminais, fechando automaticamente os arquivos — como sabemos, isso também pode ser feito através de callbacks. Obviamente, essa solução pode levar os arquivos a ficarem abertos mais tempo do que o necessário, devido ao indeterminismo da coleta de lixo em linguagens com coletores baseados na técnica de rastreamento.

Da mesma forma que nos usos anteriores, podemos utilizar notificação

passiva. Nesse caso, o coletor irá inserir o descritor do arquivo na fila de notificações quando não existirem mais referências ordinárias para ele. Sempre que necessário, o programa pode acessar a fila e fechar os arquivos. Atrasos no fechamento dos arquivos ainda podem ocorrer, pois para fechar um arquivo o proxy que o representa já deve ter sido inserido na fila pelo coletor. Contudo, os problemas de concorrência e sincronização são evitados.

3.2.3 Coleta de Referências Cíclicas

Coletores de lixo que utilizam exclusivamente contagem de referências são incapazes de liberar a memória ocupada por estruturas de dados cíclicas. Na linguagem Perl, o suporte a detecção de ciclos deve ser feito através de finalizadores, como mostram Christiansen e Torkington (Christiansen03). A solução proposta consiste em primeiro definir um *container* (um classe de objetos cujo propósito é conter outros objetos) para cada estrutura de dados recursiva ou potencialmente cíclica, como anéis, grafos e listas duplamente encadeadas. Todo acesso à estrutura deve ser feito através do seu container. A seguir, deve-se implementar um finalizador para cada container definido. A responsabilidade do finalizador é examinar a estrutura potencialmente cíclica armazenada em seu container a fim detectar e eliminar explicitamente os ciclos existentes. O finalizador é invocado quando a contagem de referências de seu respectivo container atinge zero. Isso garante que todos os objetos de uma estrutura potencialmente cíclica são coletados imediatamente após a última referência para seu container ser destruída.

A dificuldade associada a coleta de referência cíclicas em coletores que usam apenas contagem de referências foi uma das motivações iniciais para o desenvolvimento e uso de referências fracas (Brownbridge85). É bem mais intuitivo e adequado usar referências fracas para resolver esse problema do que finalizadores. A solução é simples: basta substituir referências ordinárias por referências fracas de tal forma que qualquer ciclo de referências seja composto por ao menos uma referência fraca, assim como visto na Seção 3.1.1.

3.2.4 Fallback

Aplicações que fazem uso intensivo de recursos computacionais limitados precisam liberar tais recursos à medida em que eles não são mais utilizados. Vimos na Seção 3.2.2 que finalizadores nem sempre são uma solução adequada para liberar recursos externos, pois coletores de lixo baseados em rastreamento podem atrasar a realização dessa operação. Foi sugerido nessa seção que a

coleta de lixo fosse explicitamente forçada sempre que disponibilidade de recursos caísse abaixo de níveis pré-estabelecidos. No entanto, ao invés de forçar toda a coleta de lixo, o ideal seria prover um método para realizar apenas a liberação do recurso. Como erros nesse processo de desalocação são bastante comuns, finalizadores seriam utilizados como um mecanismo de segurança (*fallback*) para liberar os recursos externos que deveriam ter sido liberados explicitamente. Ou seja, o finalizador verifica se o recurso já foi liberado e, caso ainda não tenho sido, executa uma rotina para desalocá-lo. Ainda que não existam garantias sobre quando ou mesmo se o finalizador vai ser invocado, essa pequena redundância é uma solução melhor do que depender apenas da liberação explícita dos recursos, que podem acabar não sendo liberado caso o programador cometa algum erro. Algumas classes da biblioteca padrão da linguagem Java, como `LogFileManager`, implementam essa solução (Venners98).

No geral, fallbacks são mecanismos utilizados para dar continuidade à execução do programa mesmo após a ocorrência de falhas. Uma facilidade bastante comum nas linguagens de programação e que caracteriza bem um fallback é o bloco `try-finally` (Sebesta02). Essa construção é utilizada basicamente para o tratamento de exceções; porém, ela também pode ser vista como um mecanismo de finalização baseado na estrutura sintática do programa. A Listagem 3 mostra a estrutura geral do bloco `try-finally`. Após sair do escopo definido pela cláusula `try`, mesmo que existam exceções ainda não tratadas, o fluxo de execução do programa é imediatamente transferido para a cláusula `finally`.

Listagem 3 Bloco `try-finally` com tratadores.

```
try {
...
}
catch(...) {
...
}
... //Outros tratadores
finally{
...
}
```

Essa estrutura pode ser adaptada para funcionar como um finalizador no caso da desalocação de recursos externos tratado no primeiro parágrafo. Para isso, a cláusula `try` deve conter o bloco de código referente à criação e à utilização do recurso e a cláusula `finally`, por sua vez, deve assegurar

que o recurso será liberado. Mesmo que uma exceção interrompa prematuramente a execução do bloco `try`, o fluxo será transferido para a cláusula `finally`, garantindo a liberação do recurso. Um exemplo disso se encontra na Listagem 4.

Listagem 4 Liberando recursos com o bloco `try-finally`.

```
Proxy p;
try {
    p = new Proxy();
    foo(p);
}
finally{
    p.freeResources();
}
```

Infelizmente, o uso do bloco `try-finally` como um mecanismo de finalização para liberar recursos externos possui algumas desvantagens em relação a finalizadores tradicionais. Em primeiro lugar, é mais simples liberar um recurso através do finalizador do proxy que o representa do que inserir blocos `try-finally` em cada região do programa em que o proxy é criado. Em segundo lugar, a necessidade de finalização está quase sempre associada a tipos abstratos de dados, e não à estrutura sintática do programa (Hayes92, Schwartz81).

Pode parecer que no tratamento de exceções é vantajoso usar o bloco `try-finally` como finalizador. Porém, quando a implementação da linguagem está programada para executar um coleta de lixo sempre que ocorrer uma exceção, basta definir um finalizador para liberar o recurso, o que elimina a necessidade do bloco `try-finally`. Mesmo que uma exceção impeça o recurso de ser liberado explicitamente, a execução do coletor e, consequentemente, do finalizador irá garantir a liberação do recurso.

3.2.5 Reciclagem de Objetos

Determinados objetos possuem um custo de criação relativamente alto devido a sua complexidade. Uma aplicação pode obter ganhos de desempenho significativos se, ao invés de sempre criar esses objetos, ela reciclar os que não são mais utilizados. Uma forma de efetuar essa reciclagem é através da definição de finalizadores para todos os objetos recicláveis. A rotina de finalização deve decidir se o objeto correspondente será coletado ou reciclado, o que vai depender de alguns parâmetros relevantes, como o número de objetos

recicláveis disponíveis no sistema ou a quantidade de memória livre. Novos objetos só são criados quando não existirem objetos recicláveis livres.

Algumas linguagem, como Java, não permitem que o finalizador associado a um objeto seja executado mais de uma vez. Nesse caso, geralmente não é muito útil implementar esse tipo de solução, já que que um objeto poderá ser reciclado uma única vez.

A reciclagem de objetos com finalizadores dá origem ao que chamamos de *objetos ressucitáveis*. Um objeto ressucitável é gerado quando a aplicação não pode mais acessar o objeto a partir do momento em que este se torna desconexo, porém o objeto pode influenciar no comportamento futuro da aplicação. No mecanismo de notificação ativa isso ocorre apenas quando o coletor de lixo limpa a referência fraca antes de executar o callback associado a ela. No caso da notificação passiva, não há meios de existirem objetos ressucitáveis.

Para uma implementação via um mecanismo de notificação passiva, todos os objetos candidatos a serem reciclados devem possuir uma referência fraca. O coletor irá inserir o objeto candidato à reciclagem em uma fila de notificações quando não existirem mais referências ordinárias para ele. O programa é então responsável por acessar a fila e reciclar os objetos necessários.

Outra forma de implementar essa reciclagem é inserindo o objeto em uma tabela fraca, de forma que a tabela mantenha o objeto através de uma referência fraca. Nesse caso, quando não mais existirem referências ordinárias para o objeto, ele permanecerá um tempo armazenado na tabela fraca até ser coletado. Nesse tempo, ele pode ser reciclado se necessário. Porém, essa abordagem tráz limitações óbvias. Caso o coletor de lixo seja baseado exclusivamente em contagem de referências, o objeto é coletado imediatamente após a última referência ordinária ser removida. Nesse caso, nunca haverá tempo para ele ser reciclado. No entanto, isso pode ser contornado com a criação de um tipo especial de referências fracas que impede que o objeto seja coletado enquanto a memória não atingir níveis críticos.

3.3 Conclusões

Mesmo sendo pouco conhecido e pouco utilizado, o mecanismo de referências fracas consitui-se numa alternativa elegante para a implementação de um mecanismo de finalização. Vimos que callbacks e finalizadores tradicionais possuem características semelhantes; mais especificamente, ambos possuem o mesmo nível de automação na execução das rotinas de finalização e ambos apresentam problemas de concorrência, sincronização e indeterminismo. No

entanto, o mecanismo de notificação passiva apresenta algumas vantagens em relação a callbacks e finalizadores, pois elimina os problemas de concorrência e sincronização e diminui o indeterminismo. Porém, um pouco de automação é perdida, já que o programador é responsável por chamar as rotinas de finalização. Dependendo da aplicação, isso pode acabar trazendo algumas desvantagens. No entanto, para alguns dos usos de finalizadores encontrados é até mais adequado e intuitivo utilizar notificação passiva que callbacks ou finalizadores. Por esses motivos, acreditamos que toda implementação de referências fracas deveria fornecer suporte a um mecanismo de notificação passiva, mesmo que a implementação já ofereça suporte a finalizadores ou callbacks.

Com base no que foi discutido neste capítulo, decidimos implementar um mecanismo de notificação passiva para a linguagem Lua. Esse mecanismo será descrito no Capítulo 4.

4
Mecanismos de Finalização Baseados em Referências Fracas

Este capítulo descreve um mecanismo de finalização baseado em referências fracas para a linguagem Lua. Além disso, apresentamos algumas linguagem de programação cujo mecanismo de referências fracas pode ser utilizado na construção de finalizadores. As Seções 4.1, 4.2 e 4.3, discutem respectivamente os mecanismo de finalização de Modula-3, Python e Haskell. Nosso objetivo é mostrar que em algunas linguagens o suporte a finalizadores já pode ser considerado desnecessário, pois referências fracas podem ser usadas para implementar a finalização de objetos. No entanto, a linguagem Python mostrada aqui ainda oferece finalizadores tradicionais. Ao contrário do que consideramos ideal para a implementação de finalizadores, todas as linguagens descritas utilizam um mecanismo de notificação ativa para implementar as rotinas de finalização.

Na Seção 4.4, descrevemos nossa implementação de um mecanismo de finalização baseado em referências fracas para a linguagem Lua. Esse mecanismo foi adaptado à implementação atual do coletor de lixo de Lua. Ao contrário dos outros mecanismos descritos, optamos por utilizar a notificação passiva, pois evita os problemas de concorrência e sincronização tratados na Seção 3.2.

4.1
Modula-3

Modula-3 segue o paradigma imperativo e possui suporte para tipos parametrizados, multithreading, tratamento de exceções e coleta de lixo. A fim de fornecer um maior controle sobre a coleta de lixo, Modula-3 possui um mecanismo de referências fracas que pode ser utilizado através da interface `WeakRef`.

Referências fracas são criadas através do método `FromRef`. Esse método recebe como parâmetro uma referência ordinária para um objeto qualquer alocado na memória heap e retorna uma referência fraca para esse objeto. Uma referência fraca é removida quando o coletor de lixo determina que o objeto não é mais alcançável pelo programa. A definição de um objeto alcançável

para Modula-3 será dada mais adiante. Uma vez que uma referência fraca é removida, ela não pode ser ressuscitada, mesmo que o objeto ao qual ela se refere se torne alcançável novamente. Um callback pode ser associado a uma referência fraca através de um parâmetro opcional do método `FromRef`. Caso o callback não seja nulo, o coletor de lixo irá executá-lo quando determinar que o objeto referenciado tornou-se inacessível, e após remover a referência fraca. O callback sempre é executado antes do objeto associado ser efetivamente desalocado e recebe como parâmetro uma referência ordinária para o próprio objeto, podendo então ressuscitá-lo. Mais de um callback pode ser associado a uma referência fraca, mas a especificação da linguagem não oferece qualquer garantia quanto à ordem de execução (Modula07).

De acordo com a definição da linguagem, um objeto é alcançável se existe um caminho de referências para ele a partir de variáveis globais, valores nos registros de ativação ativos, ou ainda a partir de um nó fracamente referenciado que possua um callback não nulo. Sendo assim, uma referência fraca para um objeto A não o torna alcançável, porém, se ele possui um callback, os objetos referenciados por A serão alcançáveis. Dessa forma, agregações de objetos são sempre finalizadas seguindo uma ordem correta, ou seja, seguindo o sentido das referências entre os objetos. Por outro lado, referências cíclicas impedem a finalização (e a coleta) de objetos finalizáveis.

A Listagem 5 foi retirada do trabalho de Leal (Leal05) e ilustra a implementação de um rotina de finalização em Modula-3 (o símbolo "..." foi colocado onde o código não é relevante para o exemplo). Em Modula-3, `WeakRef.T` é uma estrutura de dados que referencia um objeto sem impedir que ele seja coletado, ou seja, representa uma referência fraca. Além disso, `REFANY` é um tipo da linguagem que representa todas as referências que podem ser rastreadas. Voltando ao exemplo, temos que `MyStream` representa uma classe qualquer que envia dados através de um stream e utiliza um buffer. Ao tornar-se inacessível, um objeto dessa classe deve esvaziar o buffer e fechar o stream correspondente. O método `New` cria um novo stream que é armazenado na variável `res` e, a seguir, cria uma referência fraca para o novo stream através do método `FromRef`. Nesse exemplo, como a referência fraca não é utilizada o resultado da função é ignorado, o que justifica o uso de `EVAL`. Porém, o coletor continua tendo acesso a essa referência, pois todas as referências fracas criadas por um programa são armazenadas pelo coletor numa estrutura interna. O método `Cleanup` implementa a rotina de finalização. O parâmetro `self` desse método representa a referência fraca que tem `Cleanup` como rotina de finalização. O parâmetro `ref` representa uma referência ordinária para o objeto a ser finalizado. A implementação de `FromRef` se encarrega de associar

a referência fraca criada ao parâmetro `self` e o objeto ao parâmetro `ref`.

Listagem 5 Implementação de uma rotina de finalização em Modula-3

```
MODULE MyStream; IMPORT WeakRef, Wr, ...;

  PROCEDURE New(...): Stream =
    (* Inicializa res como sendo um Stream *)
    VAR res := NEW (Stream);
    BEGIN
      ...
      (* Cria uma referência fraca e associa um callback a ela *)
      EVAL WeakRef.FromRef(res, Cleanup);
      RETURN res
  END New;
  ...

  (* Callback representando uma rotina de finalização *)
  PROCEDURE Cleanup(READONLY self: WeakRef.T; ref: REFANY) =
    VAR wr: Stream := ref;
    BEGIN
      IF NOT Wr.Closed(wr) THEN
        (* esvazia o buffer e fecha o stream *)
        Wr.Flush(wr);
        Wr.Close(wr);
      END
  END Cleanup;

END MyStream.
```


4.2 Python

O módulo `weakref` implementa o mecanismo de referências fracas de Python. Para criar uma referência fraca, basta utilizar a função `ref` passando o objeto a ser referenciado como parâmetro. Essa função retorna um objeto que representa uma referência fraca para o objeto original. A função `ref` também aceita um segundo parâmetro opcional, um callback que é invocado quando o objeto referenciado torna-se inalcançável (mas antes de ser coletado, ou mesmo finalizado). O objeto que representa a referência fraca será passado como único parâmetro do callback. Se mais de um callback estiver associado a um objeto, os callbacks serão executados na ordem inversa em que foram registrados.

Além de oferecer um mecanismo de finalização baseado em referências fracas, finalizadores em Python também podem ser implementados através do método `__del__`. Dessa forma, o finalizador é acoplado à classe, seguindo a

semântica típica de lingagens orientadas a objeto. Em Python, é possível implementar tanto finalização baseada em coleções, utilizando callbacks, quanto finalização baseada em classes, através do método `__del__`. A principal diferença é que callbacks podem ser associados dinamicamente às instâncias de uma classe. Além disso, pode-se associar múltiplos callbacks a um objeto.

4.3 Haskell

O Glasgow Haskell Compiler (GHC) implementa referências fracas através de pares chave/valor, onde o valor é considerado acessível se a chave for acessível, mas a conectividade do valor não influencia na conectividade da chave. Durante a coleta de lixo, o campo referente ao valor de uma referência fraca não é rastreado a não ser que a chave seja alcançável. Esse tipo de referência fraca é uma generalização das referências fracas comuns usadas na criação de mapeamentos fracos com dinâmicas de coleta complexas, como *memo tables* (Jones00).

O GHC permite que sejam associadas ações[1] (basicamente callbacks, que em GHC são denominados finalizadores) a referências fracas, as quais são executadas após as chaves serem limpas. Se múltiplos finalizadores forem associados à mesma chave, estas serão executadas em uma ordem arbitrária, ou mesmo de forma concorrente. O modo mais simples de criar uma referência fraca é através da função `mkWeakPtr` que recebe um valor de um tipo qualquer e um finalizador do tipo `IO()` e retorna uma referência fraca do tipo `Weak a` referente ao valor.

A documentação do GHC (GHC07) especifica a semântica de referências fracas baseadas nos pares chave/valor. Um objeto é coletável se: (a) ele é referenciado diretamente por um objeto alcançável, ao invés de uma referência fraca, (b) é uma referência fraca cuja chave é alcançável ou (c) é o valor ou finalizador de uma referência fraca cuja chave é alcançável. Note que um ponteiro do valor ou do finalizador para a chave associada não torna a chave alcançável. No entanto, se a chave puder ser alcançada de outra forma, então o valor e o finalizador são alcançáveis e consequentemente qualquer outra chave que seja referenciada por eles direta ou indiretamente. O GHC garante ainda que finalizadores registrados serão executados uma única vez, quer seja quando a chave correspondente for limpa, ou através de uma invocação explícita, ou ainda ao final da execução da aplicação. Essa especificação de referências fracas é semelhante à semântica de ephemerons, diferindo em alguns detalhes,

[1] Em linguagens puramente funcionais, efeitos colaterais e estados globais podem ser representados através de *monads*. Mais detalhes sobre esse conceito podem ser encontrados em nos trabalhos de Wandler (Wandler95, Wandler90).

como no tratamento de finalizadores. Uma comparação mais detalhada sobre o mecanismo de ephemerons e a implementação de referências fracas de Haskell pode ser encontrado no trabalho de Jones (Jones00).

4.4 Mecanismo de Notificação Passiva para Lua

A implementação atual da linguagem Lua oferece suporte a um mecanismo de finalização que pode ser usado exclusivamente com um tipo específico, `userdata`. Para tornar um `userdata` finalizável, deve-se registrar um finalizador para esse objeto (através da definição do metamétodo `__gc`). Após o coletor de lixo determinar que um objeto não pode ser mais acessado pelo programa, ele insere o finalizador correspondente a esse objeto em uma fila. Ao final do ciclo de coleta de lixo, os finalizadores são invocados, recebendo como parâmetro o próprio objeto. Lua garante que todos os finalizadores serão invocados antes do término da aplicação.

Vimos na Seção 2.1.1 que o coletor de lixo de Lua é baseado na técnica de rastreamento. Sendo assim, não é possível determinar quando os finalizadores serão executados. No Capítulo 3, vimos que esse indeterminismo pode afetar negativamente o desempenho da aplicação. O mesmo ocorre caso o mecanismo de finalização seja baseado em notificação ativa, usando callbacks. Nesse caso, o coletor de lixo também irá decidir quando executar os callbacks associados às referências fracas, e não é possível determinar quando isso irá ocorrer. Além disso, vimos que a execução de callbacks e finalizadores pode introduzir linhas de execução concorrentes na aplicação e, caso a linguagem utilizada não ofereça um suporte adequado à concorrência e sincronização, o uso desses mecanismos pode levar a inconsistências durante a execução do programa.

A fim de fornecer ao programador um maior controle sobre a coleta através de um mecanismo de finalização simples, decidimos incorporar à implementação atual do coletor de lixo de Lua um mecanismo de notificação passiva. Esse mecanismo, ao contrário dos finalizadores existentes na implementação atual da linguagem, pode ser usado com qualquer tipo de Lua, e não apenas com `userdata`. Mesmo perdendo um pouco da automação, pois o programador é responsável por acessar a fila de notificações, acreditamos que esse mecanismo de notificação passiva pode trazer mais vantagens que o mecanismo de finalizadores atual. Isso é justificado pelo fato de que a notificação passiva elimina o problema de concorrência e sincronização e atenua o problema do indeterminismo. Com esse mecanismo, o programa pode esperar por condições específicas para executar as ações associadas à finalização de um objeto. Além disso, o uso de notificação passiva elimina o problema de objetos ressucitáveis,

pois o objeto a ser finalizado continua acessível para o programa, ao contrário do que o ocorre com finalizadores.

A linguagem Lua implementa referências fracas através de sua principal estrutura de dados, os arrays associativos também conhecidos como *tabelas Lua*. Na Seção 2.3, vimos que, modificando o campo `__mode` de uma metatabela, o programador pode criar tabelas onde apenas a chave é fraca, apenas o valor é fraco, ou ambos são fracos. Essas tabelas são chamadas de tabelas fracas. O mecanismo de notificação passiva que implementamos está aclopado a essas tabelas de forma que a cada tabela fraca pode ser associada uma *tabela de notificações*[2]. Cabe ao programador estabelecer essa associação.

O programador pode optar por usar ou não uma tabela de notificações para uma determinada tabela fraca. Para isso, estabelecemos um novo campo para as metatabelas de Lua, o campo `__notify`. Caso o programador queira habilitar o uso de uma tabela de notificações para uma tabela fraca, ele deve atribuir a esse campo uma tabela vazia. Dessa forma, sempre que o coletor de lixo remover uma entrada da tabela fraca, ele irá copiar essa entrada para a tabela de notificações da metatabela, ou seja, para a tabela atribuída ao campo `__notify`. Caso o campo `__notify` seja nulo, ou caso o programador tenha atribuído a esse campo outro valor que não seja uma tabela, o coletor irá simplesmente remover a entrada da tabela fraca.

A Listagem 6 exemplifica de maneira simples o uso de uma tabela de notificações. Inicialmente, criamos uma tabela com chaves e valores fracos, atribuindo a string `"kv"` ao campo `__mode` da metatabela. Vale notar que uma tabela de notificações pode ser associada a qualquer tabela fraca, tenha ela apenas chaves fracas, apenas valores fracos, ou tanto chaves quanto valores fracos. Voltando ao exemplo, após criar a tabela fraca, atribuímos uma nova tabela ao campo `__notify` da metatabela. Em seguida, duas entradas são inseridas na tabela fraca. A referência do programa para a chave da primeira entrada é perdida quando atribuímos uma nova tabela a variável `key`. Em seguida, forçamos uma coleta chamando a função `collectgarbage` a fim de coletar a primeira entrada. Como definimos uma tabela de notificações para a tabela fraca, o coletor irá copiar a entrada para a tabela de notificações após removê-la da tabela fraca. O primeiro loop `for` irá imprimir a string `"2"`, pois apenas a segunda entrada permanece na tabela. O segundo loop, por sua vez, irá imprimir `"1"`, pois a primeira entrada foi movida para a tabela de notificações.

Diferentes tabelas fracas podem possuir uma mesma tabela de noti-

[2]Como em Lua a fila de notificações é representada através de uma tabela, iremos nos referir a ela como tabela de notificações.

Listagem 6 Criando uma tabela de notificações para uma tabela fraca.

```
wt = {} -- tabela fraca
mt = {} -- metatabela
setmetatable(wt, mt)
mt.__mode = "kv" -- Define os valores e as chaves como fracos
mt.__notify = {} -- Cria a tabela de notificacao

key={}
wt[key] = 1
key = {}
wt[key] = 2

collectgarbage()

for k, v in pairs(wt) do
print(v)
end
for k, v in pairs(mt.__notify) do
print(v)
end
```


ficações. Isso pode ocorrer de duas maneiras. A primeira delas é quando uma mesma metatabela é associada a diferentes tabelas fracas, e o valor do campo `__notify` da metatabela é uma tabela. Um exemplo desse caso é mostrado na Listagem 7, onde as tabelas fracas `a` e `b` foram associadas à metatabela `mt`. Duas entradas foram adicionadas à tabela `a` e à tabela `b`. Como as chaves são fracas e não existem referências para elas fora das tabelas `a` e `b`, todas as chaves serão coletadas e copiadas para a tabela de notificações. Como `a` e `b` possuem a mesma tabela de notificações, o loop `for` irá imprimir as strings `"1"`, `"2"`, `"3"` e `"4"`.

O segundo modo de construir uma mesma tabela de notificações para tabelas fracas diferentes é atribuindo uma mesma tabela a campos `__notify` de diferentes metatabelas, que por sua vez estão associadas às tabelas fracas. Um exemplo é mostrado na Listagem 8, onde a tabela `n` foi atribuída à tabela de notificações da tabela fraca `a` e à tabela de notificações da tabela fraca `b`. Nesse caso, todas as entradas removidas das tabelas fracas `a` e `b` serão copiadas para suas respectivas tabelas de notificações. Como essas tabelas referenciam a mesma tabela, a tabela de notificações de `a` irá conter as entradas removidas da tabela `b` e vice-versa, ou seja, elas serão a mesma tabela.

Existe um outro caso especial que devemos levar em consideração: quando um mesmo objeto é inserido em diferentes tabelas fracas. Digamos que duas tabelas fracas, A e B, possuam o objeto k como chave (o valor pode também

Listagem 7 Associando duas tabelas fracas a uma mesma metatabela.

```
a = {} -- tabela fraca
b = {}
mt = {} -- metatabela
setmetatable(a, mt)
setmetatable(b, mt)
mt.__mode = "k" -- Define os valores e as chaves como fracos
mt.__notify = {} -- Cria a tabela de notificacao

a[{}] = 1
a[{}] = 2
b[{}] = 3
b[{}] = 4
collectgarbage()

for k, v in pairs(mt.__notify) do
print(v)
end
```


referenciar um mesmo objeto ou objetos diferentes). Suponha que as chaves de A e B são mantidas por referências fracas e que não existem referências ordinárias para o objeto k, ou seja, k pode ser coletado. Sendo assim, caso as tabelas A e B possuam diferentes tabelas de notificações, k será copiado duas vezes: uma vez ao ser inserido na tabela de notificações de B e uma vez ao ser inserido na tabela de notificações de A. Contudo, caso as tabelas A e B possuam a mesma tabela de notificações, o coletor irá copiar k para essa tabela de notificações quando removê-lo da primeira tabela e, em seguida, irá copiar uma segunda vez ao remover da segunda, sobrepondo a primeira cópia. Consequentemente, apenas uma cópia de k pode ser encontrada na tabela de notificações.

Vamos mostrar agora como implementar finalização através do nosso mecanismo de notificação passiva. De modo geral, a tabela de notificações contém todos os objetos que podem ser finalizados. Sendo assim, o programador deve acessar essa tabela e executar para cada objeto encontrado, a rotina de finalização adequada. Quando a tabela de notificações não for mais útil ao programa, este também deve explicitamente limpar essa tabela, por exemplo, atribuindo `nil`, para que o coletor de lixo seja capaz de reciclar a memória associada a ela.

A Listagem 9 apresenta um uso real em Lua que utiliza uma implementação em C para arrays. Nesse caso, a variável `path` é uma String contendo o caminho para a biblioteca em C. As funções `new` e `clean` são respectivamente

Listagem 8 As tabelas de notificações de `a` e `b` referenciam a mesma tabela.

```
a = {}
b = {}
mta = {} -- metatabela de a
mtb = {} -- metatabela de b
setmetatable(a, mta)
setmetatable(b, mtb)
mta.__mode = "k"
mtb.__mode = "k"
n = {}
mta.__notify = n
mtb.__notify = n
```

funções da biblioteca para criar um novo array e limpar a memória ocupada pelo programa em C. Após criar um novo array e atribuí-lo à variável `a`, criamos uma tabela fraca com valores fracos e inserimos `a` como valor dessa tabela. Dessa forma, a tabela `wt` não impedirá que `a` seja coletado. Em seguida, criamos uma tabela de notificações para a tabela fraca `wt` e atribuímos `nil` à variável `a` para que o userdata antes armazenado seja coletado. Logo após, forçamos uma coleta de lixo. Por fim, o programa acessa explicitamente a tabela de notificações e, para cada objeto encontrado, executa a rotina `array.clean`, passando como parâmetro o próprio objeto.

A implementação atual desse mecanismo de notificação possui um problema crítico. Para entender esse problema, precisamos discutir alguns detalhes do funcionamento do coletor de lixo de Lua. Vimos, na Seção 2.1.1 que o coletor de Lua possui quatro fases: uma para rastrear os objetos, uma fase atômica, onde um conjunto de operações é executado em um único passo, uma fase para desalocar a memória ocupada pelos objetos não marcados e uma fase para invocar os finalizadores. Durante a fase de rastreamento, todas as tabelas fracas encontradas são colocadas em uma lista. Ao final dessa fase, e durante a fase atômica, o coletor percorre a lista de tabelas fracas e remove todos os pares que possuem chaves ou valores fracos que não foram alcançados pelo rastreamento, ou seja, não são referenciados fora da tabela fraca. Isso ocorre em uma função chamada `cleartable`, que não pode ser intercalada com a execução do programa e por isso está inserida na fase atômica do coletor. Em seguida, o coletor prossegue para as fases de desalocação de memória e execução dos finalizadores.

Neste capítulo, vimos que a tabela de notificações deve ser preenchida enquanto o coletor remove as entradas da tabela fraca. Sendo assim, adaptamos a função `cleartable` para que esse comportamento fosse obtido, ou seja,

Listagem 9 Exemplo de uso de notificação passiva para finalização de objetos.

```
array_Init, err1 = package.loadlib(path, "_luaopen_array")

if not array_Init then -- se a biblioteca não pode ser aberta
error(err1)
end

array_Init()

a = array.new(1000000)
wt = {a}
mt = {}
setmetatable(wt, mt)
mt.__mode = "v"
mt.__notify = {}

a = nil

collectgarbage()

for k, v in pairs(mt.notify) do
array.clean(v)
end
```


dentro dessa função preenchemos a tabela de notificações. Isso significa que durante a fase atômica, na execução de `cleartable`, o coletor precisa alocar memória para construir a tabela. O coletor de Lua executa um controle rígido sempre que precisa alocar memória. Esse controle abrange toda a implementação da coleta de lixo e é extremamente necessário para garantir que erros de memória não irão ocorrer durante a coleta. Contudo, na nossa implementação, não foi possível executar esse controle, pois toda a construção da tabela de notificações está contida dentro de `cleartable` que por sua vez é executada em um único passo. Caso não haja memória suficiente para executar essa construção, a própria coleta pode gerar um erro por falta de memória. Para impedir esse erro, teríamos que trabalhar de forma mais abrangente no coletor de Lua, não apenas modificando a função `cleartable`, mas sim realizando adaptações em todo o algoritmo. Esse trabalho é extremamente complexo e não foi possível elaborar uma solução. Consequentemente, não é aconselhável utilizar nossa implementação corrente do mecanismo de notificação passiva numa futura versão da linguagem Lua.

5
Eliminando Ciclos

Tabelas fracas são ideais para a construção de tabelas de propriedades, pois não interferem na conectividade do objeto ao qual se deseja adicionar propriedades dinamicamente. Porém, um grande problema com tabelas fracas ainda persiste na maioria das linguagens. A existência de referências cíclicas entre chaves e valores impede que os elementos que compõem o ciclo sejam coletados, mesmo que eles não sejam mais utilizados pelo programa. Além disso, como foi visto na Seção 2.3 uma tabela fraca com um encadeamento de n elementos levará pelo menos n ciclos para ser completamente limpa.

Como vimos na Seção 2.3.1, 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 tem nenhuma relação com a conectividade da chave. Esse mecanismo foi adaptado com sucesso à implementação do coletor de lixo de Haskell. Baseado nisso, desenvolvemos uma adaptação também para a linguagem Lua, que na sua implementação atual apresenta o problema de ciclos em tabelas fracas.

Para adaptar o mecanismo de ephemerons ao coletor de lixo de Lua, estudamos detalhadamente o algoritmo apresentado por Hayes (Hayes97) e a implementação do coletor. Para nossa surpresa, essa adaptação foi bastante simples e os detalhes estão descritos na Seção 5.1. Porém, antes de discutirmos nossa implementação, vamos mostrar como a linguagem Lua pode fornecer suporte ao mecanismo de ephemerons.

Sabemos que ephemerons são pares chave/valor que não estão necessariamente armazenados em uma tabela. Porém, como Lua fornece suporte a referências fracas através de tabelas fracas, o suporte a ephemerons é feito através de *tabelas de ephemerons*. A criação de uma tabela de ephemerons é semelhante à criação de uma tabela fraca, diferindo apenas na configuração do campo `__mode` da metatabela. A Listagem 10 mostra como criar uma tabela de ephemerons em Lua. Após criar a tabela `et` e associá-la a metatabela `mt`,

devemos atribuir a string `"e"` ao campo `__mode` da metatabela. Dessa forma, classificamos `et` como sendo uma tabela de ephemerons. Mesmo que o programador atribua `"ev"` ao campo `__mode`, esperando tornar os valores fracos, o coletor irá ignorar o caractere `'v'`. Na verdade, basta a string possuir o caractere `'e'` que o ephemeron será criado e todos os outros caracteres serão ignorados. Em seguida, a listagem mostra a criação de um ciclo na tabela de ephemerons, onde o valor da primeira entrada referencia a chave da segunda e o valor da segunda entrada referencia a chave da primeira. Esse ciclo não seria coletado se estivéssemos usando uma tabela fraca com chaves fracas (comum na criação de tabelas de propriedades) ao invés de uma tabela de ephemerons. Após a criação do ciclo, forçamos uma coleta para que ele seja coletado. Sendo assim, o valor da variável `count` ao final da execução do loop será igual a zero, pois não restam entradas na tabela de ephemerons.

Listagem 10 Criando um ephemeron em Lua.

```
et = {}
mt = {}
setmetatable(et, mt)
mt.__mode = "e" -- define a tabela como sendo um ephemeron

-- Criando um ciclo
a = {}
b = {}
et[a] = b
et[b] = a
a = nil
b = nil

collectgarbage()

count = 0
for k,v in pairs(et) do
count = count + 1
end
print(count) -- 0
```

5.1 Implementação

Antes de iniciar qualquer implementação, estudamos detalhadamente o algoritmo de coleta de lixo que oferece suporte a ephemerons, apresentado por Hayes (Hayes97) e a implementação do coletor de lixo de Lua. Como vimos na Seção 5.1, inicialmente a coleta de lixo percorre o grafo das relações

entre os objetos até encontrar um ephemeron. Quando isso ocorre, no lugar de percorrer imediatamente os campos do ephemeron, a coleta insere esse ephemeron em uma lista para que possa ser processado futuramente. No nosso caso, precisávamos de uma estrutura de dados para armazenar as tabelas de ephemerons encontradas durante a fase de rastreamento do coletor de Lua. Ao nos depararmos com essa situação, achamos que acabaríamos com o mesmo problema da implementação do mecanismo de notificação passiva para Lua, descrito na Seção 4.4. A versão atual desse mecanismo ainda precisa ser melhorada para que o coletor passe a exercer um controle mais rígido sobre a construção da tabela de notificações, a fim de evitar erros de memória. Para que o mesmo não ocorresse ao construir a lista de tabelas de ephemerons, precisávamos que sua criação fosse monitorada pelo coletor, o que levaria a uma mudança complexa em toda a coleta de lixo. Porém, ao estudar detalhadamente o coletor de Lua, vimos que este já possui uma estrutura cuja criação é monitorada e que serve perfeitamente para armazenar as tabelas de ephemerons. Esse estrutura é a lista de tabelas fracas, chamada `weak`. Vimos que durante a fase de rastreamento, sempre que o coletor encontra uma tabela fraca, ele o insere nessa lista. O que fizemos foi também inserir nessa lista as tabelas de ephemerons encontradas. Como o coletor já se encarrega de gerenciar essa lista, não tivemos que nos preocupar com erros por falta de memória que poderiam ocorrer durante sua construção. Isso facilitou bastante a implementação do mecanismo de ephemerons para Lua.

Algumas ações foram acrescentadas às fases do coletor de lixo de Lua para que este passasse a fornecer suporte ao mecanismo de ephemerons. Inicialmente, na fase de rastreamento, sempre que o coletor encontra uma tabela de ephemerons ele a insere na lista `weak`, onde também são inseridas as tabelas fracas. As entradas da tabela de ephemerons não são percorridas, nem as chaves nem os valores. Isso é tudo o que foi acrescentado à primeira fase.

Em seguida, o coletor entra na fase atômica, onde várias operações são executadas em um único passo. Duas novas funções foram inseridas na fase atômica: `traverseephemerons`, que percorre a lista `weak` a procura de tabelas de ephemerons e `convergeephemerons`, que chama a primeira função. Um pseudo-código dessas funções encontra-se na Listagem 11. A função `traverseephemerons` percorre a lista `weak` de tabelas fracas e tabelas de ephemerons. Quando essa função acha uma tabela de ephemerons, ou seja, quando o resultado do teste na linha 3 de `traverseephemerons` é verdadeiro, essa função irá percorrer todos os campos da tabela de ephemerons encontrada. Caso a chave de algum campo tenha sido marcada, o valor correspondente é marcado. A função `traverseephemerons` retorna um valor booleano, definido

pela variável b na Listagem 11. Essa booleano é verdadeiro caso algum valor de alguma tabela de ephemerons tenha sido marcado e falso caso contrário.

A função `convergeephemerons` chama a função `traverseephemerons`. Caso essa última retorne verdadeiro, ou seja, caso algum valor tenha sido marcado, a função `convergeephemerons` chama uma função da implementação original do coletor, `propagateall`. A função `propagateall` não foi modificada. Sua responsabilidade é rastrear o grafo de referências do programa e expandir a barreira de objetos cinza, de acordo com o algoritmo tricolor marking visto na Seção 2.1.1. Consequentemente, objetos referenciados direta ou indiretamente por valores em tabelas de ephemerons que foram marcados durante a última execução de `traverseephemerons` serão marcados. Como esses objetos podem ser chaves pertencentes a tabelas de ephemerons, a função `convergeephemerons` chama novamente `traverseephemerons`. Esse comportamento se repete até que nenhum valor em nenhuma tabela de ephemerons tenha sido marcado, o que fará a função `traversephemerons` retornar falso. A função `convergeephemerons` é uma adaptação para o coletor de Lua da segunda fase do mecanismo de original de ephemerons visto na Seção 2.3.1.

Listagem 11 Pseudo-código das funções `convergeephemerons` e `traversephemerons`

```
function convergeephemerons(...)
 1: while traverseephemerons(...) do
 2:    propagateall(...)
 3: end while

function traverseephemerons(...)
 1: for all t | t ∈ weak do
 2:    b ⇐ ⊥
 3:    if type(t) ≡ ephemeron then
 4:       for all e | e ∈ hash(t) do
 5:          if key(e) está marcada then
 6:             marca o valor
 7:             b ⇐ ⊤
 8:          end if
 9:       end for
10:    end if
11: end for
12: return b
```

Após a execução de `convergeephemerons` e ainda na fase atômica, o coletor chama a função `cleartable`, que assim como `propagateall` faz parte da implementação original do coletor. A função `cleartable` não só limpa as entradas das tabelas fracas como também as entradas das tabelas

de ephemerons que não foram marcadas. Por fim, o coletor prossegue para as fases de desalocação de memória e execução dos finalizadores, que não foram modificadas. As implementações das funções `convergeephemerons` e `traverseephemerons`, assim como as modificações feitas em funções originais do coletor de lixo de Lua, podem ser encontradas no Apêndice A.

5.2 Análise de Eficiência

Nesta seção, fazemos uma análise do comportamento do coletor de lixo de Lua na coleta de tabelas de ephemerons e tabelas fracas. Considere um programa A que cria um número K_e de tabelas de ephemerons e um programa B que cria um número K_f de tabelas fracas, apenas com chaves fracas. Suponha que cada tabela de ephemerons possui e_h entradas na parte hash e e_a entradas na parte array e cada tabela fraca possui f_h entradas na parte hash e f_a entradas na parte array.

Lua possui um coletor de lixo incremental, onde as operações de coleta são realizadas a curtos passos, intercaladas com a execução da aplicação. Vimos que o coletor de lua possui quatro fases, porém, para simplificar o entendimento de seu comportamento na coleta de tabelas de ephemerons e tabelas fracas, iremos dividir a coleta de lixo em duas fases: uma fase atômica, onde um conjunto de operações de coleta deve ser executado em um único passo, e uma fase não atômica, onde a coleta e a execução da aplicação são intercaladas (essa fase engloba as fases de rastreamento, desalocação de memória e execução dos finalizadores vistas anteriormente). Vamos analisar cada uma dessas duas fases em separado.

Quando tratamos com tabelas de ephemerons e tabelas fracas, algumas funções da coleta são cruciais para o desempenho. No caso essas funções são:

- `traversetable`: percorre uma tabela marcando suas chaves e valores se for o caso.
- `traverseephemerons`: itera sobre a lista de tabelas fracas e tabelas de ephemerons, percorrendo as tabelas de ephemerons encontradas e marcando os valores cujas chaves foram alcançadas.
- `cleartable`: limpa as entradas das tabelas de ephemerons e das tabelas fracas que não foram alcançadas na fase de rastreamento.

Dentre as três funções apresentadas, apenas a função `traversetable` é executada na fase não atômica. Para cada tabela, seja ela comum, fraca ou uma tabela de ephemerons, essa função é executada uma vez. A função `traversetable` possui dois loops, um para percorrer a parte array da tabela e

outro para percorrer a parte hash. No entanto, quando a tabela em questão é uma tabela de ephemerons, o loop que percorre a parte hash não é executado, pois como as chaves podem ser coletáveis, os valores não devem ser marcados. Sendo assim, o custo de `traversetable` para percorrer uma tabela fraca é $\mathcal{O}(f_h + f_a)$ e o custo para percorrer uma tabela é ephemerons é $\mathcal{O}(e_a)$. Sempre que `traversetable` percorre uma tabela fraca ou uma tabela de ephemerons, ela insere a tabela na lista `weak`. Essa lista será utilizada nas funções `cleartable` e `traverseephemerons`.

Na fase atômica, `traversetable` é executada uma vez para cada tabela fraca e para cada tabela de ephemerons. Isso se deve ao fato de que, para evitar complexidade na barreira de escrita do algoritmo tricolor marking, essas tabelas são mantidas cinza, podendo ser percorridas novamente. Sendo assim, após rastrear as tabelas fracas e as tabelas de ephemerons, a fase atômica executa a função `convergeephemerons`. Como vimos na seção anterior, essa função irá chamar `traverseephemerons` repetidas vezes até que esta não marque nenhuma valor. Para o programa B, onde apenas tabelas fracas foram criadas, `traverseephemerons` é chamada apenas uma vez, e esta percorre a lista de tabelas fracas também uma única vez. Isso ocorre porque como não existem tabelas de ephemerons, nenhum valor é marcado. O custo das chamadas à função `traverseephemerons` para o programa B é $\mathcal{O}(K_f)$.

Quando a lista `weak` contém tabelas de ephemerons, como no programa A, temos que considerar o melhor e o pior caso de `convergeephemerons`. No melhor caso, não existem valores que apontem direta ou indiretamente para uma chave em outra tabela de ephemerons. Assim, a lista será percorrida no máximo duas vezes: uma para marcar os valores das chaves que foram alcançadas e outra caso algum valor tenha sido marcado na vez anterior. Dessa forma, no melhor caso, o custo das chamadas à função `traverseephemerons` para o programa A é $\mathcal{O}(K_e \times e_h)$

No pior caso, existe um encadeamento de chaves e valores. O primeiro exemplo desse tipo de encadeamento está na Figura 5.1. O ponteiro inicial é forte e vem de alguma parte do programa, mas não de uma tabela de ephemerons. Nesse exemplo, temos que o valor de cada tabela de ephemerons aponta para a chave da próxima de modo que a função `traverseephemerons` será executada $K_e + 1$ vezes, uma vez para marcar cada valor e uma última vez que não modifica nada. Para esse pior caso temos que o custo das chamadas à função `traversephemerons` para o programa A é $\mathcal{O}(K_e^2 \times e_h)$.

Outro exemplo de pior caso é quando as chaves e os valores de uma mesma tabelas estão encadeados, como mostra a Figura 5.2. Nesse caso a função `traverseephemerons` será executada $(2 \times e_h) + 1$ vezes, uma vez para

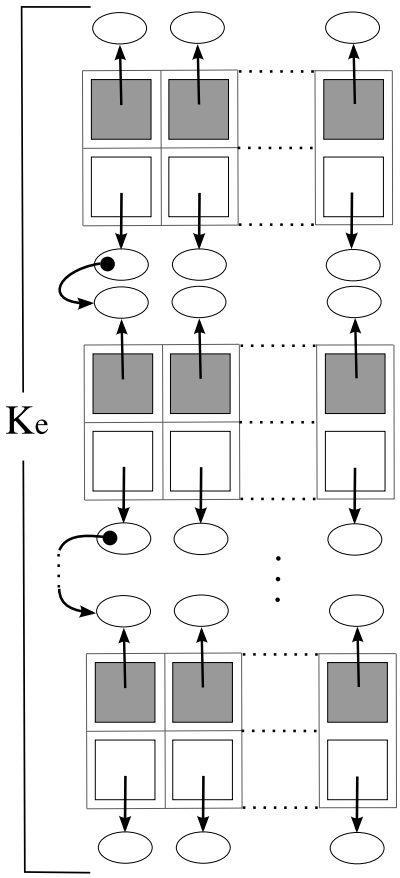

Figura 5.1: Encadeamento de tabelas

marcar cada valor e uma última vez que não modifica nada. Sendo assim, o custo das chamadas à função `traversephemerons` para o programa A é $\mathcal{O}(e_h^2 \times K_e)$.

A função `cleartable` se comporta da mesma maneira tanto para tabelas de ephemerons quanto para tabelas fracas. O custo dessa função é $\mathcal{O}(K_e \times (e_h + e_a))$ para o programa A e $\mathcal{O}(K_f \times (f_h + f_a))$ para o programa B. Dessa forma, podemos concluir que o custo da coleta de tabelas de ephemerons para o programa A é $\mathcal{O}(K_e \times (e_h + e_a))$ no melhor caso e para o programa B é $\mathcal{O}(K_f \times (f_h + f_a))$. Se considerarmos os programas A e B idênticos, com exceção de que A usa tabelas de ephemerons e B tabelas fracas, e tanto as tabelas de ephemerons quanto as tabelas fracas não possuem ciclos, temos que o custo de cada programa é praticamente o mesmo. Isso ocorre, pois a função `traverseephemerons` percorre a parte hash da tabela de ephemerons, compensando a função `traversetable` que só percorre a parte array.

Contudo, nos dois exemplos de pior caso mostrados, o custo da coleta para o programa A torna-se quadrático [1]. Mais especificamente, para o caso

[1] O custo da coleta é quadrático no tamanho do encadeamento, independete de melhor ou pior caso. Como no melhor caso não existe encadeamento entre tabelas, o custo é linear.

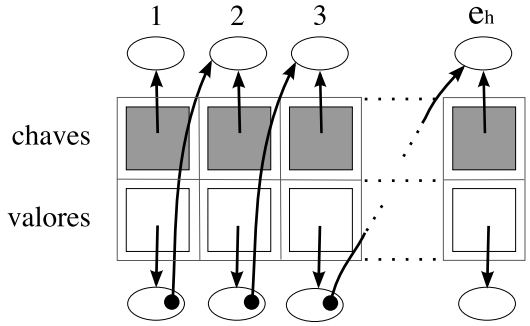

Figura 5.2: Encadeamento de chaves e valores

mostrado na Figura 5.1, o custo da coleta é $\mathcal{O}(K_e^2 \times e_h)$ e para o caso mostrado na Figura 5.2 o custo da coleta é $\mathcal{O}(e_h^2 \times K_e)$. Os encadeamentos entre tabelas ou entre chaves e valores que mostramos são raros de ocorrer, porém, quando ocorrem afetam de forma considerável a eficiência. O custo da coleta do programa B por sua vez continua linear, mas devemos lembrar que como em B são utilizadas tabelas fracas ao invés de tabelas de ephemerons, os ciclos não são coletados, acarretando um desperdício de memória (que potencialmente pode ser bem pior que uma queda na eficiência).

5.3 Medidas de Eficiência

Com o intuito de medir a eficiência do coletor de lixo ao tratar tabelas de ephemerons, realizamos dois testes em nossa implementação. No primeiro teste comparamos a eficiência do coletor ao tratar tabelas de ephemerons sem ciclos e tabelas de ephemerons encadeadas, como mostrado na Figura 5.1. O teste foi executado primeiramente para diferentes quantidades de tabelas de ephemerons sem ciclos, variando de 100 a 1000 tabelas com um espaçamento de 100 (cada tabela continha 500 entradadas). Em seguida, o teste foi executado novamente para essas mesmas quantidades de tabelas, porém, com todas as tabelas encadeadas. Dessa forma, podemos comparar o tempo de execução do melhor caso e o tempo de execução do pior caso. O resultado desse teste é mostrado na Figura 5.3. A curva referente as tabelas de ephemerons encadeadas se assemelha à curva de uma função quadrática, como esperado. Podemos observar que a eficiência do coletor de lixo é bastante afetada ao tratar do pior caso. No entanto, ao usar tabelas de ephemerons todos os ciclos são coletados, enquanto que via tabelas fracas eles permanecem na memória.

O segundo teste foi realizado a fim de comparar a eficiência do coletor no tratamento de tabelas fracas e tabelas de ephemerons. Dessa vez, nem as tabelas fracas nem as tabelas de ephemerons continham ciclos, já que eles não

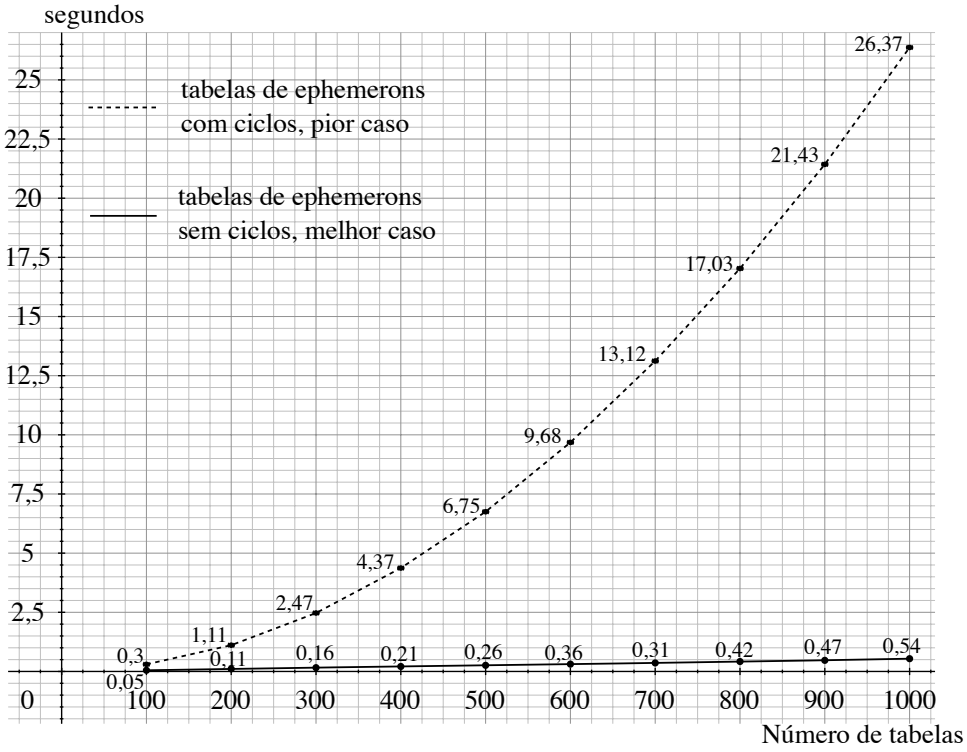

Figura 5.3: Coleta de tabelas de ephemerons: pior caso x melhor caso

podem ser coletados quando usamos tabelas fracas. Executamos esse segundo teste para mesmas quantidades de tabelas e entradas usadas no teste anterior, primeiro com tabelas fracas e em seguida com tabelas de ephemerons. O resultado pode ser visto na Figura 5.4. Note que quase não existe diferença entre o tempo de coleta. Acreditamos que o melhor resultado para a coleta de tabelas de ephemerons se deve a algum ruído na execução dos testes, e não a uma eficiência maior na coleta dessas tabelas. Sendo assim, na ausência de ciclos, nossa implementação do mecanismo de ephemerons é tão eficiente quanto a implementação de tabelas fracas.

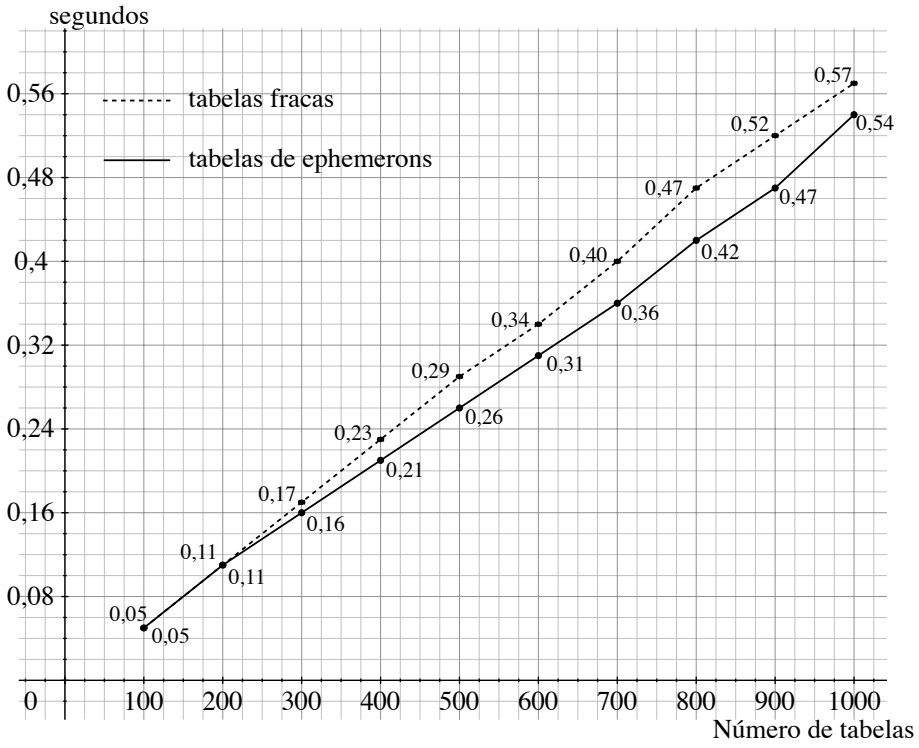

Figura 5.4: Coleta de tabelas fracas x coleta de tabelas de ephemerons

6
Conclusão

Finalizadores e referências fracas recebem suporte de praticamente todas as linguagens de programação que oferecem coleta de lixo automática. Para uma série recorrente de problemas, esses dispositivos podem ser usados em soluções elegantes, eficazes e por vezes únicas. Porém, ao contrário de finalizadores, referências fracas ainda são pouco conhecidas e pouco utilizadas, tanto na comunidade acadêmica quanto na indústria.

Referências fracas são um mecanismo mais simples e mais expressivo que finalizadores. Vimos que, quando referências fracas são associadas a um mecanismo de notificação, elas podem ser empregadas inclusive como um mecanismo alternativo de finalização. A finalização através de callbacks, via notificação ativa, possui algumas vantagens em relação aos finalizadores tradicionais:

- Quando o objeto finalizável é desacoplado da rotina de finalização, os atrasos na reciclagem de memória são evitados.
- Quando o objeto finalizável é passado como parâmetro do callback, ele continua acessível antes de ser efetivamente removido, fornecendo à aplicação um maior controle sobre a coleta.
- Callbacks podem ser associados a qualquer tipo da linguagem.
- Vários callbacks pode ser associados a um mesmo objeto, assim como um mesmo callback pode ser associado a vários objetos.

A notificação passiva fornece esses mesmo benefícios, porém, apresenta algumas vantagens em relação à callbacks e finalizadores. Conforme discutimos, em sistemas que empregam coletores de lixo baseados em rastreamento a utilização de callbacks ou finalizadores pode introduzir linhas de execução concorrentes na aplicação. Além disso, o indeterminismo desses coletores pode afetar negativamente o desempenho de algumas aplicações. Na notificação passiva, o problema de concorrência e sincronização é eliminado. Quanto à questão do indeterminismo, o coletor ainda é responsável por preencher a fila de notificações. O programa pode esperar por condições específicas para executar as ações associadas à finalização de um objeto, porém, isso irá depender do

coletor já ter inserido o objeto na fila. Por isso dizemos que a notificação passiva atenua o problema do indeterminismo, mas não o elimina completamente. Apesar de perder um pouco de automação, pois o programa deve invocar as rotinas de finalização explicitamente, acreditamos que as vantagens do mecanismo de notificação passiva o tornam uma opção mais adequada para a implementação de finalizadores na maioria dos casos.

Para todos os usos encontrados de finalizadores, pudemos elaborar uma solução através de referências fracas, mais especificamente, via um mecanismo de notificação passiva. Em alguns casos, esse mecanismo constituiu-se em uma solução mais simples e intuitiva. Baseado na discussão sobre finalizadores tradicionais, callbacks e filas de notificações apresentada no Capítulo 3, decidimos implementar um mecanismo de notificação passiva para Lua. Na implementação atual da linguagem, finalizadores podem ser usados apenas com um tipo específico, `userdata`. Com nosso mecanismo de notificação passiva, finalizadores podem ser usados com qualquer tipo da linguagem Lua. Além disso, eliminamos o problema de objetos ressucitáveis, pois o objeto permanece acessível até ser finalizado. No entanto, vimos que nossa implementação, como se encontra atualmente, não pode ser utilizada numa versão futura da linguagem Lua, pois a construção da fila de notificação não é devidamente controlada pelo coletor, o que pode ocasionar erros de memória.

Além da implementação de finalizadores via referências fracas, tratamos outro problema que constitui uma das contribuições mais importantes deste trabalho: a modificação do coletor de lixo da linguagem Lua para que este oferecesse suporte ao mecanismo de ephemerons. Agora, o programador pode optar por usar uma tabela de ephemerons ao invés de uma tabela fraca. Dessa forma, os ciclos existentes entre chaves e valores de uma tabela de ephemerons serão coletados. Isso torna essas tabelas uma opção ainda mais adequada que tabelas fracas para implementação de tabelas de propriedades. Tudo o que pode ser feito via referência fracas pode ser feito também via as tabelas de ephemerons. A fim de analisar a implementação do mecanismo de ephemerons, executamos testes de eficiência e realizamos uma análise informal do custo da coleta para tabelas e ephemerons e tabelas fracas. Como principal resultado, vimos que, na ausência de ciclos, tanto a coleta de tabelas fracas quanto a coleta de tabelas de ephemerons possuem o mesmo custo e o mesmo nível de eficiência. Contudo, quando existem ciclos, vimos que a eficiência da coleta de tabelas de ephemerons é bastante afetada. De fato, o custo passa de linear, no melhor caso (sem ciclos), para um custo exponencial, no pior caso. No entanto, devemos levar em consideração que a ocorrência do pior caso é rara. Sendo assim, devido a esses resultados, nossa implementação do mecanismo de

ephemerons para Lua pode ser incluída numa versão futura da linguagem.

6.1 Contribuições

Neste trabalho, tentamos mostrar como é possível implementar finalizadores via referências fracas. Além disso, resolvemos o problema de ciclos em tabelas fracas presente na linguagem Lua através do mecanismo de ephemerons. Resumidamente, as principais contribuições deste trabalho são:

- Efetuamos uma pesquisa informal sobre os usos de referências fracas e finalizadores na comunidade acadêmica e na indústria.
- Através do resultado da pesquisa e de uma pesquisa bibliográfica, identificamos e descrevemos os principais usos desses mecanismos.
- Mostramos, para cada uso encontrado de finalizadores, uma implementação através de referências fracas.
- Discutimos como o mecanismo de notificação passiva pode ser vantajoso em relação a callbacks e finalizadores tradicionais.
- Implementamos um mecanismo de notificação passiva para a linguagem Lua acoplado às tabelas fracas.
- Estudamos em detalhes o coletor de lixo de Lua e o mecanismo de ephemerons e estabelecemos a melhor adaptação desse último ao primeiro.
- Implementamos um suporte a ephemerons para a linguagem Lua, resolvendo o problema de ciclos em tabelas fracas. O programador pode agora utilizar uma tabela de ephemerons ao invés de uma tabela fraca tradicional.

Um ponto importante que não abordamos neste trabalho refere-se a como implementar um mecanismo de notificação passiva para Lua de forma mais adequada. Como foi discutido, o mecanismo que implementamos pode causar erros por falta de memória, já que na fase atômica da coleta de lixo, o próprio coletor precisa alocar memória para construir a fila de notificações e essa construção não é controlada adequadamente. Assim, como uma linha futura do trabalho, seria interessante estudar uma melhor implementação do mecanismo de notificação passiva.

Referências Bibliográficas

[Atkins88] ATKINS, M. C.; NACKMAN, L. R.. **The active deallocation of objects in object-oriented systems**. Software: Practice and Experience, 18(11):1073–1089, 1988. 1

[Bloch01] BLOCH, J.. **Effective Java Programming Language Guide**. Prentice Hall, first edition, June 2001. 1

[Boehm91] BOEHM, H.-J.; DEMERS, A. J. ; SHENKER, S.. **Mostly parallel garbage collection**. In: PROCEEDINGS OF THE ACM SIGPLAN 1991 CONFERENCE ON PROGRAMMING LANGUAGE DESIGN AND IMPLEMENTATION, p. 157–164. ACM Press, June 1991. 2.1.1

[Boehm03] BOEHM, H.-J.. **Destructors, finalizers, and synchronization**. In: PROCEEDINGS OF THE ACM SYMPOSIUM ON PRINCIPLES OF PROGRAMMING LANGUAGES, p. 262–272. ACM Press, 2003. 1, 3.2, 3.2.2, 3.2.2

[Brownbridge85] BROWNBRIDGE, D. R.. **Cyclic reference counting for combinator machines**. In: PROCEEDINGS OF THE ACM CONFERENCE ON FUNCTIONAL PROGRAMMING LANGUAGES AND COMPUTER ARCHITECTURE, p. 273–288, New York, NY, USA, 1985. Springer Verlag. 2.1.3, 3.1, 3.1.1, 3.2.3

[CLS07] **C# language specification**. http://msdn.microsoft.com/library/default.asp?url=/library/en-us/csspec/html/CSharpSpecStart.asp, último acesso em 2 de abril de 2007. 3

[Cheney70] CHENEY, C. J.. **A nonrecursive list compacting algorithm**. Communications of the ACM, 13(11):677–678, November 1970. 2.1.1

[Christiansen03] CHRISTIANSEN, T.; TORKINGTON, N.. **Perl Cookbook**. O'Reilly, second edition, August 2003. 3.2, 3.2.3

[Cohen83] COHEN, J.; NICOLAU, A.. **Comparison of compacting algorithms for garbage collection**. ACM Transactions on Programming Languages and Systems, 5(4):532–553, October 1983. 2.1.1

[Collins60] COLLINS, G. E.. **A method for overlapping and erasure of lists**. Communications of the ACM, 2(12):655–657, December 1960. 2.1.1

[Deutsch76] DEUTSCH, L. P.; BOBROW, D. G.. **An efficient, incremental, automatic garbage collector**. Communications of the ACM, 19(9):522–526, September 1976. 2.1.1

[Dijkstra78] DIJKSTRA, E. W.; LAMPORT, L.; MARTIN, A. J.; SCHOLTEN, C. S. ; STEFFENS, E. F. M.. **On-the-fly garbage collection: An exercise in cooperation**. Communications of the ACM, 21(11):966–975, November 1978. 2.1.1, 2.1.1

[Dybvig93] DYBVIG, R. K.; BRUGGEMAN, C. ; EBY, D.. **Guardians in a generation-based garbage collector**. In: SIGPLAN CONFERENCE ON PROGRAMMING LANGUAGE DESIGN AND IMPLEMENTATION, p. 207–216, 1993. 1

[Fenichel69] FENICHEL, R. R.; YOCHELSON, J. C.. **A LISP garbage collector for virtual memory computer systems**. Communications of the ACM, 12(11):611–612, November 1969. 2.1.1

[GCS07] **Google code search**. http://www.google.com/codesearch, último acesso em 2 de abril de 2007. 3

[GHC07] TEAM, T. H.. **Hugs/ghc extension libraries: Weak**. http://www.dcs.gla.ac.uk/fp/software/ghc/lib/hg-libs-15.html, último acesso em 2 de abril de 2007. 4.3

[Gamma95] GAMMA, E.; HELM, R.; JOHNSON, R. ; VLISSIDES, J.. **Design Patterns: Elements of Reusable Objetct-Oriented Software**. Addison Wesley, 1995. 3.1.4

[Haskell07] **The Glasgow Haskell Compiler user's guide, version 6.2**. http://www.haskell.org/ghc, último acesso em 2 de abril de 2007. 1

[Hayes92] HAYES, B.. **Finalization in the collector interface**. In: IWMM '92: PROCEEDINGS OF THE INTERNATIONAL WORKSHOP ON MEMORY MANAGEMENT, p. 277–298, London, UK, 1992. Springer Verlag. 1, 2.1.2, 3.2.4

[Hayes97] HAYES, B.. **Ephemerons: a new finalization mechanism**. In: OOPSLA '97: PROCEEDINGS OF THE 12TH ACM SIGPLAN CONFERENCE ON OBJECT-ORIENTED PROGRAMMING, SYSTEMS, LANGUAGES, AND APPLICATIONS, p. 176–183, New York, NY, USA, 1997. ACM Press. 1, 2.1.2, 2.1.3, 2.3.1, 3.1, 5, 5.1

[Homing93] HOMING, J.; KALSOW, B.; MCJONES, P. ; NELSON, G.. **Some useful modula-3 interfaces**. Technical Report 113, Digital Equipment Corporation, Systems Research Center, December 1993. 2.1.3

[Ierusalimschy06] IERUSALIMSCHY, R.. **Programming in Lua**. Lua.org, second edition, 2006. 1, 1.1, 3.1.2

[Jensen91] JENSEN, K.; WIRTH, N.; MICKEL, A. B. ; MINER, J. F.. **Pascal User Manual and Report: ISO Pascal Standard**. Springer Verlag, fourth edition, September 1991. 2.1

[Jones96] JONES, R.; LINS, R.. **Garbage Collection: Algorithms for Automatic Dynamic Memory Management**. Wiley, first edition, 1996. 1, 2, 2.1, 2.1.1

[Jones00] JONES, S. P.; MARLOW, S. ; ELLIOTT, C.. **Stretching the storage manager: Weak pointers and stable names in Haskell.** In: IMPLEMENTATION OF FUNCTIONAL LANGUAGES, 11TH INTERNATIONAL WORKSHOP, volumen 1868 de **Lecture Notes in Computer Science**, p. 37–58. Springer Verlag, 2000. 3.1.2, 4.3

[Kernighan88] KERNIGHAN, B. W.; RITCHIE, D. M.. **The C Programming Language**. Prentice Hall, second edition, 1988. 2.1

[Krugle07] **Krugle - code search for developers**. http://www.krugle.com, último acesso em 2 de abril de 2007. 3

[LML07] **Lua mailing list**. http://www.lua.org/lua-l.html, último acesso em 2 de abril de 2007. 3

[Leal05] LEAL, M. A.. **Finalizadores e Referências Fracas: Interagindo com o Coletor de Lixo**. PhD thesis, Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro, 2005. 1, 2.2, 3, 3.1, 3.2, 4.1

[Liu00] LIU, J. W. S.. **Real-Time Systems**, chapter 2, p. 26–33. Prentice Hall, first edition, April 2000. 2.1.1

[McCarthy60] MCCARTHY, J.. **Recursive functions of symbolic expression and their computation by machine, part I**. Communications of the ACM, 3(4):184–195, April 1960. 2.1.1

[Modula07] **Critical Mass Modula-3 5.1 documentation**. http://www.elegosoft.com/cm3/doc, último acesso em 2 de abril de 2007. 2.1.3, 4.1

[Muhammad06] MUHAMMAD, H. H.. **Estudo sobre APIs de linguagens de script**. Master's thesis, Pontifícia Universidade Católica do Rio de Janeiro, August 2006. 3.2.1

[Rees84] REES, J. A.; ADAMS, N. I. ; MEEHAN, J. R.. **The T Manual**. Yale University Computer Science Department, fourth edition, January 1984. 1

[Richter02] RICHTER, J.. **Applied Microsoft .NET Framework Programming**. Microsoft Press, first edition, January 2002. 3

[Rossum06] VAN ROSSUM, G.. **Python library reference**, March 2006. Release 2.4.3. 2.1.3

[Rovner85] ROVNER, P.. **On adding garbage collection and runtime types to a strongly-typed, statically-checked, concurrent language**. Technical report CSL-84-7, Xerox PARC, Palo Alto, CA, 1985. 1

[SUN04] MICROSYSTEMS, S.. **Java 2 plataform standard edition 5.0: API specification**, 2004. 1, 3.2.1

[Schwartz81] SCHWARTZ, R.; MELLIAR-SMITH, P. M.. **The finalization operation for abstract types**. In: ICSE'81: 5TH INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING, p. 273–282. IEEE Press, 1981. 3.2.4

[Schwartz05] SCHWARTZ, R.; PHOENIX, T. ; D'FOY, B.. **Learning Perl**. O'Reilly, fourth edition, July 2005. 3

[Sebesta02] SEBESTA, R. W.. **Concepts of Programming Languages**. Addison Wesley, fifth edition, 2002. 3.2.4

[Stroustrup97] STROUSTRUP, B.. **The C++ Programing Language**. Addison Wesley, third edition, June 1997. 2.1.2

[TCG07] **Tecgraf, computer graphics technology**. http://www.tecgraf.puc-rio.br/, último acesso em 2 de abril de 2007. 3

[Venners98] VENNERS, B.. **Object finalization and cleanup**. JavaWorld, June 1998. 3.2.4

[Wandler90] WANDLER, P.. **Comprehending monads**. In: 1990 ACM CONFERENCE ON LISP AND FUNCTIONAL PROGRAMMING, p. 61–78. ACM Press, 1990. 1

[Wandler95] WANDLER, P.. **Monads for functional programming**. Lecture Notes in Computer Science, 925:24–52, 1995. 1

[Wilson92] WILSON, P. R.. **Uniprocessor garbage collection techniques**. In: PROCEEDINGS OF THE 1992 INTERNATIONAL WORKSHOP ON MEMORY MANAGEMENT, volumen 637, p. 1–42, Saint-Malo (France), 1992. Springer Verlag. 2, 2.1, 2.1.1

[Wise77] WISE, D. S.; FRIEDMAN, D. P.. **The one-bit reference count**. BIT Numerical Mathematics, 17(3):351–359, September 1977. 2.1.1

[Xerox85] Xerox Palo Alto Research Center (PARC), Palo Alto, CA. **InterLISP Reference Manual**, October 1985. 1

A
Modificações Realizadas no Coletor de Lixo da Linguagem Lua

Neste apêndice, apresentamos as modificações feitas nas funções originais do coletor de lixo da linguagem Lua assim como as novas funções implementadas a fim de fornecer suporte ao mecanismo de ephemerons e à tabela de notificações. A seguir, listamos o código referente à implementação das mais relevantes novas funções para o mecanismo de ephemerons, `convergeephemerons` e `traverseephemerons`.

```
static void convergeephemerons(global_State *g, GCObject *l){
  while(traverseephemerons(g, l)) propagateall(g);
}

static int traverseephemerons(global_State *g, GCObject *l){
  int marked = 0;
  while(l){
    Table *h = gco2h(l);
    if(testbit(h->marked, EPHEMERONBIT)){
      int i = sizenode(h);
      while (i--) {
        Node *n = gnode(h, i);
        if (!ttisnil(gval(n)) && /* non-empty entry? */
            !iscleared(key2tval(n), 1) && iscleared(gval(n), 0)) {
          markvalue(g, gval(n));
          marked = 1;
        }
      }
    }
    l = h->gclist;
  }
  return marked;
}
```

A próxima função, `traversetable`, foi modificada a partir de sua implementação original. As linhas 11 a 19 testam se a string contida no campo `__mode` da metatabela contém o caractere "e", o que classifica a tabela como uma tabela de ephemerons. Em seguida, esse pedaço de código marca a tabela como sendo uma tabela de ephemerons e a insere na lista `weak`. O próximo pedaço de código referente à implementação de ephemerons está na linha 39. Essa linha testa se a tabela não é uma tabela de ephemerons, pois quando o é a parte hash não deve ser percorrida.

```
1:   static int traversetable (global_State *g, Table *h) {
2:     int i;
3:     int weakkey = 0;
4:     int weakvalue = 0;
5:     int isephemeron = 0;
6:     const TValue *mode;
7:     if (h->metatable)
8:       markobject(g, h->metatable);
9:     mode = gfasttm(g, h->metatable, TM_MODE);
10:    if (mode && ttisstring(mode)) {  /* is there a weak or ephemeron mode? */
11:      isephemeron = (strchr(svalue(mode), 'e') != NULL);
12:      weakkey = !isephemeron && (strchr(svalue(mode), 'k') != NULL);
13:      weakvalue = !isephemeron && (strchr(svalue(mode), 'v') != NULL);
14:      if (isephemeron) {
15:        h->marked &= ~EPHEMERON;
16:        h->marked |= cast_byte(isephemeron << EPHEMERONBIT);
17:        h->gclist = g->weak;
18:        g->weak = obj2gco(h);
19:      }
20:      else if (weakkey || weakvalue) {  /* is really weak? */
21:        h->marked &= ~(KEYWEAK | VALUEWEAK);  /* clear bits */
22:        h->marked |= cast_byte((weakkey << KEYWEAKBIT) |
23:                               (weakvalue << VALUEWEAKBIT));
24:        h->gclist = g->weak;  /* must be cleared after GC, ... */
25:        g->weak = obj2gco(h);  /* ... so put in the appropriate list */
26:      }
27:    }
28:    if (weakkey && weakvalue) return 1;
29:
30:    /* mark the array part, even if it's an ephemeron */
31:    if (!weakvalue) {
```

```
32:       i = h->sizearray;
33:       while (i--){
34:         markvalue(g, &h->array[i]);
35:       }
36:     }
37:
38:     /* only mark the hash part if it's not an ephemeron */
39:     if( !isephemeron) {
40:       i = sizenode(h);
41:       while (i--) {
42:         Node *n = gnode(h, i);
44:         lua_assert(ttype(gkey(n)) != LUA_TDEADKEY || ttisnil(gval(n)));
45:         if (ttisnil(gval(n)))
46:           removeentry(n);  /* remove empty entries */
47:         else {
48:           lua_assert(!ttisnil(gkey(n)));
49:           if (!weakkey) markvalue(g, gkey(n));
52:           if (!weakvalue) markvalue(g, gval(n));
55:         }
56:       }
57:     }
58:     return (weakkey || weakvalue) || isephemeron;
59:   }
```

A função `cleartable` mostrada a seguir foi modificada a fim de oferecer suporte à tabela de notificações. Como função auxiliar de `cleartable` implementamos a função `marknotify` cujo código é mostrado após `cleartable`. As linhas 6 a 8, testam se o campo `__notify` foi definido e atribuem a variável local a tabela de notificações. A seguir, nas linhas 19 a 24, antes de remover o valor da parte array da tabela, ele é copiado para a tabela de notificações. E por último, nas linhas 36 a 41, antes de remover o valor da parte hash da tabela, ele é copiado também para a tabela de notificações.

```
1:   static void cleartable (lua_State *L, GCObject *l) {
2:     while (l) {
3:       Table *h = gco2h(l);
4:
5:       global_State *g = G(L);
6:       const TValue *notify = gfasttm(g, h->metatable, TM_NOTIFY);
```

```
7:      Table *notifications = NULL;
8:      if(notify && ttistable(notify)) notifications = hvalue(notify);
9:
10:     lua_assert(testbit(h->marked, VALUEWEAKBIT) ||
11:                testbit(h->marked, KEYWEAKBIT) ||
12:                testbit(h->marked, EPHEMERONBIT));
13:
14:     int i = h->sizearray;
15:     if (testbit(h->marked, VALUEWEAKBIT)) {
16:       while (i--) {
17:         TValue *o = &h->array[i];
18:         if (iscleared(o, 0)){  /* value was collected? */
19:           if(notifications != NULL){
20:             /* mark notify if it's not already marked and insert the object
21:             in the array part */
22:             marknotify(g, notify);
23:             setobj2t(L, luaH_setnum(L, notifications, i+1), o);
24:           }
25:           setnilvalue(o);  /* remove value */
26:         }
27:       }
28:     }
29:     i = sizenode(h);
30:     while (i--) {
31:       Node *n = gnode(h, i);
33:       if (!ttisnil(gval(n)) &&  /* non-empty entry? */
34:           (iscleared(key2tval(n), 1) || iscleared(gval(n), 0))) {
35:
36:         if(notifications != NULL){
37:           /* mark notify if it's not already marked and insert the object
38:            in the hash part */
39:           marknotify(g, notify);
40:       setobj2t(L, luaH_set(L, notifications, key2tval(n)), gval(n));
41:     }
42:         setnilvalue(gval(n));  /* remove value ... */
43:         removeentry(n);  /* remove entry from table */
44:       }
45:     }
46:     l = h->gclist;
```

```
47:    }
48:  }

static void marknotify(global_State *g, const TValue *notify){
  GCObject *gcnotify = gcvalue(notify);
  if(!isgray(gcnotify)){
    makewhite(g, gcnotify);
    markvalue(g, notify);
  }
}
```