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

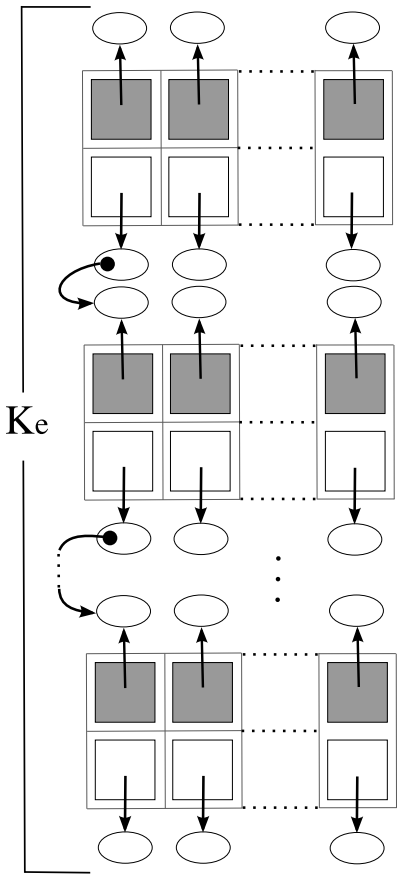

Figura 5.1: Encadeamento de tabelas

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.

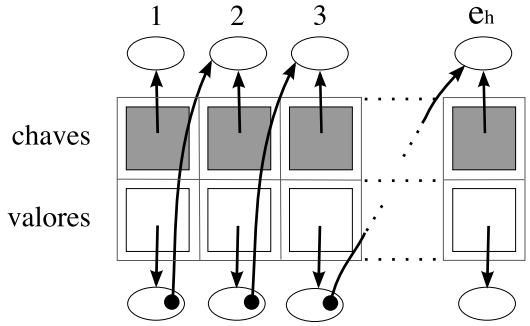

Figura 5.2: Encadeamento de chaves e valores

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

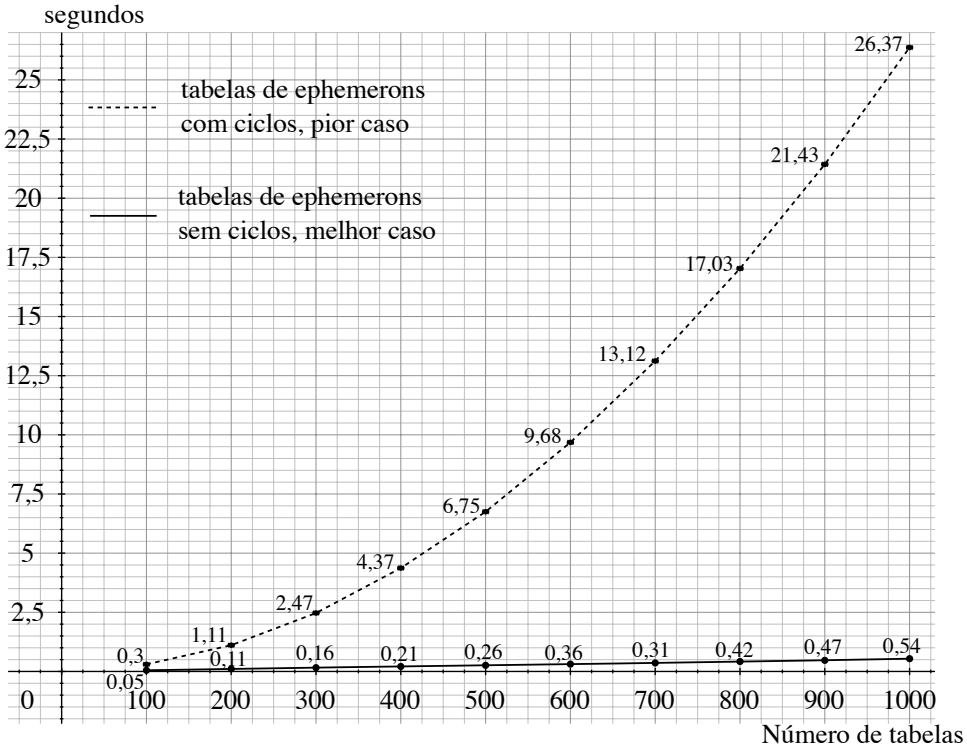

Figura 5.3: Coleta de tabelas de ephemerons: pior caso x melhor caso

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.

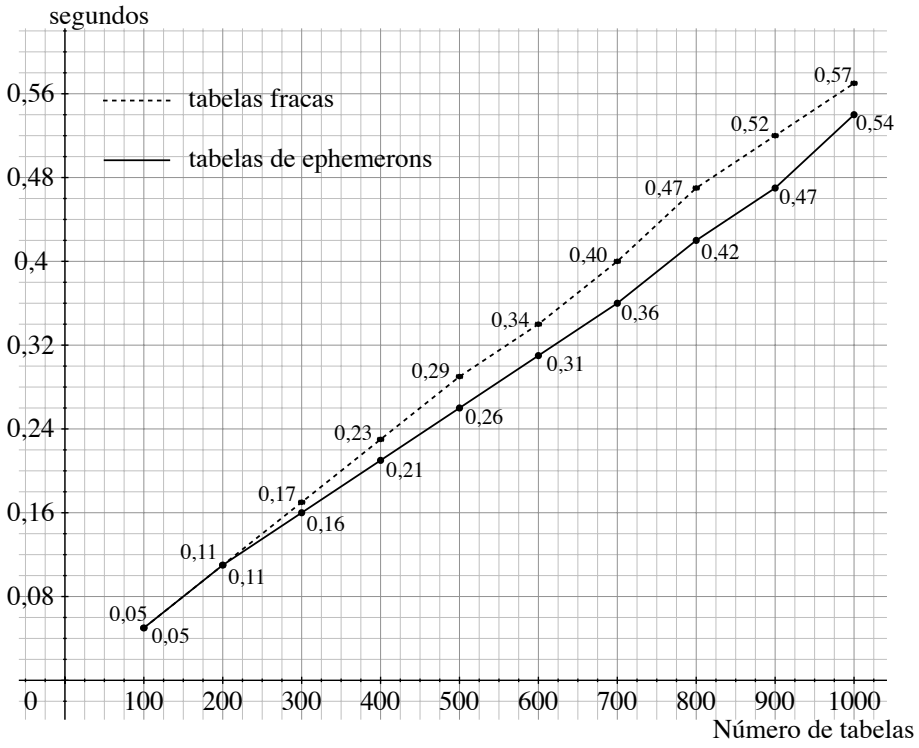

Figura 5.4: Coleta de tabelas fracas x coleta de tabelas de ephemerons