5
Um Modelo Abstrato

Neste capítulo formalizamos a semântica de coleta de lixo, finalizadores e referências fracas usando uma semântica operacional, mais especificamente, SOS (structural operational semantics) [53]. Através deste modelo exploramos com maior precisão diversas questões abordadas no último capítulo. Parte dos resultados apresentados aqui foi publicado anteriormente em [44].

Poucos trabalhos descrevem formalmente o processo de coleta de lixo. Hudak [38], por exemplo, utiliza uma semântica denotacional para formalizar o conceito de contagem de referências em uma linguagem abstrata, e desenvolve um algoritmo que permite calcular estaticamente o valor aproximado da contagem de referências associada a cada objeto. A motivação principal desse trabalho está relacionada à questão de como evitar a cópia em memória de estruturas de dados complexas, e os autores não fazem qualquer menção ao processo de coleta de lixo.

Chirimar, Cunter e Riecke [23] também descrevem formalmente o conceito de contagem de referências em uma linguagem abstrata baseada em um sistema de tipos linear (lógica linear). A semântica desta linguagem é representada através de uma abordagem operacional com um nível de abstração relativamente baixo. Isso permite aos autores provar uma série de invariantes associadas ao gerenciamento de memória.

Demers e outros [26] formalizam a noção de ponteiros, e conseguem com isso representar diferentes tipo de coletores baseados em rastreamento. Não obstante, este modelo é totalmente dissociado da semântica de qualquer linguagem de programação, o que impede a especificação de outros mecanismos de programação que dependam diretamente da dinâmica da memória.

Dentre os poucos modelos abstratos que consideram explicitamente o processo de coleta de lixo, o trabalho de Morriset, Felleisen e Harper [50] é provavelmente o mais conhecido. Este modelo emprega uma representação sintática da memória heap e especifica eventos associadas a memória através de uma série de regras de transição baseadas em uma semântica contextual de passos curtos (small-step con-

PUC-Rio - Certificação Digital Nº 0310845/CA

textual evaluation semantics). Contudo, referências ordinárias não são definidas, e o conceito de conectividade é formalizado exclusivamente através de variáveis livres. Ainda assim os autores conseguem descrever, de forma bastante elegante e sucinta, diferentes algoritmos de coleta de lixo baseados em rastreamento, como mark-and-copy e coletores generacionais. A partir desse modelo Hunter [39] descreve o processo de coleta de lixo em uma linguagem simples orientada a objetos, que não inclui finalizadores.

Outros trabalhos, como por exemplo [13, 51, 68, 28], descrevem formalmente métodos alternativos para provar a conectividade de objetos, usando sistemas de tipos e técnicas de inferência sofisticadas.

Dentro deste universo de publicações não temos conhecimento de nenhum trabalho que considere formalmente as semânticas de finalizadores e referências fracas. Além disso, acreditamos que os modelos mencionados não são suficientemente expressivos ou não podem ser facilmente estendidos para descrever tais semânticas. Por conseguinte, optamos por desenvolver um modelo próprio, com níveis de abstração e expressividade mais adequados aos nossos objetivos.

Como as questões semânticas mais importantes relacionadas a finalizadores e a referências fracas decorrem da natureza assíncrona de coletores que utilizam rastreamento, além obviamente deste tipo de coletor ser mais comum na prática, baseamos a nossa formalização nos conceitos de conectividade e rastreamento, ignorando por completo técnicas de contagem de referências. O modelo que iremos apresentar possui alguma semelhança com o modelo de Morriset, Felleisen e Harper, mas inclui uma representação explícita de referências e da respectiva semântica, o que permite uma elaboração um pouco maior do conceito de conectividade. Adicionalmente, definimos um conjunto de regras de transição que descrevem a semântica da linguagem base em um nível um pouco menos abstrato. Isto torna possível especificar aspectos da coleta de lixo que não foram considerados previamente por outros autores. Em particular, conseguimos demonstrar como finalizadores e referências fracas afetam a dinâmica do processo de coleta de lixo, e a semântica da linguagem base.

5.1 A Linguagem de Programação λ_{ref}

Para descrever a semântica de coleta de lixo, finalizadores e referências fracas, desenvolvemos inicialmente uma pequena linguagem não-tipada que chamamos de λ_{ref}. A sintaxe de λ_{ref} é similar a uma linguagem de ordem superior baseada

PUC-Rio - Certificação Digital Nº 0310845/CA

no λ-cálculo, mas estendida com referências simples e expressões condicionais. Referências são representadas por *locais* ($l_i \in \mathbf{Loc}$), que podem ser vistos como endereços para células de memória pré-alocadas.

Valores ($v \in \mathbf{V}$) são representados por abstrações (funções), locais e o átomo nil.

$$v ::= \quad \lambda x_i.e \mid l_i \mid nil$$

Expressões ($e \in \mathbf{Exp}$), conforme indicado abaixo, são representadas por valores, variáveis ($x_i \in \mathbf{Id}$), aplicações, expressões condicionais, comparações, e as operações associadas a referências: uma *alocação*, indicada pelo operador **new**, aloca uma célula de memória e cria uma referência para a mesma; um *dereferenciamento*, indicado pelo operador !, recupera o valor armazenado em uma célula de memória; e uma *atribuição*, indicada pelo operador :=, armazena um valor na célula de memória associada a referência.

$$e ::= \quad v \mid x_i \mid e_1 e_2 \mid e_1 ?\ e_2 : e_3 \mid e_1 = e_2 \mid \mathbf{new} \mid !e \mid e_1 := e_2$$

Valores são associados a referências através de *ambientes* (H), que consistem em mapeamentos finitos entre locais e valores. $H_\emptyset$ representa um ambiente vazio, e portanto, $H_\emptyset(l_i)$ é indefinido para todo l_i (computações indefinidas são representadas pelo símbolo $\perp$). Para representar mudanças em um ambiente H usamos a seguinte notação:

$$H[l_i \mapsto v](l_j) \quad = \quad \begin{cases} v & \text{se } l_j = l_i \\ H(l_j) & \text{caso contrário} \end{cases}$$

e denotamos por $H[l_i \mapsto \perp]$ um novo ambiente derivado de H, mas que não contém l_i em seu domínio.

A semântica básica de λ_{ref} é definida pelo conjunto de regras descrito nas figuras 5.1 e 5.2 (este conjunto será denotado por R_{ref}). Transições são indicadas pelo símbolo $\rightarrow$ (que pode ser lido como *é reduzido em um passo para*), e ocorrem entre diferentes estados denominados *programas*. Um programa ($\mathcal{P}$) é definido como uma expressão e o contexto associado. Por ora o contexto é representado apenas pelo ambiente, ou seja, $\mathcal{P} = \langle e, H \rangle$.

Cabem aqui alguns comentários adicionais:

- Alocações, descritas pela regra **alloc**, estendem H com locais recém-criados,

PUC-Rio - Certificação Digital Nº 0310845/CA

$$\langle \mathbf{new}, H\rangle \rightarrow \langle l_i, H[l_i \mapsto nil]\rangle \quad \text{onde } l_i \notin dom(H) \qquad (\mathbf{alloc})$$

$$\langle !l_i, H\rangle \rightarrow \langle H(l_i), H\rangle \qquad (\mathbf{deref})$$

$$\langle l_i := v, H\rangle \rightarrow \langle v, H[l_i \mapsto v]\rangle \quad \text{se } l_i \in dom(H) \qquad (\mathbf{assign})$$

$$\langle (\lambda x_i.e)v, H\rangle \rightarrow \langle \{x_i/v\}e, H\rangle \qquad (\mathbf{applic})$$

$$\langle v \ ?\ e_1 : e_2, H\rangle \rightarrow \langle e_1, H\rangle \quad \text{se } v \neq nil \qquad (\mathbf{cond1})$$

$$\langle nil \ ?\ e_1 : e_2, H\rangle \rightarrow \langle e_2, H\rangle \qquad (\mathbf{cond2})$$

$$\langle v = v, H\rangle \rightarrow \langle v, H\rangle \qquad (\mathbf{comp1})$$

$$\langle v_1 = v_2, H\rangle \rightarrow \langle nil, H\rangle \quad \text{onde } v_1 \neq v_2 \qquad (\mathbf{comp2})$$

Figura 5.1: Regras de transição de λ_{ref}.

associados a células de memória vazias (contendo o valor *nil*). Esta é a única forma de introduzir novas referências no domínio de um ambiente.

- De acordo com a regra **deref**, o dereferenciamento de uma referência resulta no valor correspondente mapeado por H.
- Atribuições (**assign**) podem ser feitas apenas a referências que pertencem ao domínio do ambiente. Uma atribuição resulta no valor do lado direito do operador de atribuição.
- Aplicações, descritas pela regra **applic**, são equivalentes a β-redução do λ-cálculo: todas as variáveis ligadas na expressão que compõe o corpo da função são substituídas pelo argumento. Esta substituição, também conhecida como *substituição de contexto*, é representada por $\{x_i/v\}$ (neste caso

PUC-Rio - Certificação Digital Nº 0310845/CA

$$\frac{\langle e_1, H_1\rangle \rightarrow \langle e_2, H_2\rangle}{\langle !e_1, H_1\rangle \rightarrow \langle !e_2, H_2\rangle} \quad (\textbf{cont1})$$

$$\frac{\langle e_1, H_1\rangle \rightarrow \langle e_2, H_2\rangle}{\langle e_1 := e_3, H_1\rangle \rightarrow \langle e_2 := e_3, H_2\rangle} \quad (\textbf{cont2})$$

$$\frac{\langle e_1, H_1\rangle \rightarrow \langle e_2, H_2\rangle}{\langle v := e_1, H_1\rangle \rightarrow \langle v := e_2, H_2\rangle} \quad (\textbf{cont3})$$

$$\frac{\langle e_1, H_1\rangle \rightarrow \langle e_2, H_2\rangle}{\langle e_1 e_3, H_1\rangle \rightarrow \langle e_2 e_3, H_2\rangle} \quad (\textbf{cont4})$$

$$\frac{\langle e_1, H_1\rangle \rightarrow \langle e_2, H_2\rangle}{\langle v e_1, H_1\rangle \rightarrow \langle v e_2, H_2\rangle} \quad (\textbf{cont5})$$

$$\frac{\langle e_1, H_1\rangle \rightarrow \langle e_2, H_2\rangle}{\langle e_1 \ ?\ e_3 : e_4, H_1\rangle \rightarrow \langle e_2 \ ?\ e_3 : e_4, H_2\rangle} \quad (\textbf{cont6})$$

$$\frac{\langle e_1, H_1\rangle \rightarrow \langle e_2, H_2\rangle}{\langle e_1 = e_3, H_1\rangle \rightarrow \langle e_2 = e_3, H_2\rangle} \quad (\textbf{cont7})$$

$$\frac{\langle e_1, H_1\rangle \rightarrow \langle e_2, H_2\rangle}{\langle v = e_1, H_1\rangle \rightarrow \langle v = e_2, H_2\rangle} \quad (\textbf{cont8})$$

Figura 5.2: Regras de transição de λ_{ref} (cont).

PUC-Rio - Certificação Digital Nº 0310845/CA

x_i está sendo substituído por v).

- As regras **cond1** e **cond2** definem expressões condicionais. Condições com valores diferentes de *nil* fazem com que a expressão resulte no valor correspondente ao seu segundo argumento. Se a condição for igual a *nil*, a expressão resulta no valor correspondente ao seu terceiro argumento.

- As regras **comp1** e **comp2** definem uma operação de comparação entre valores. A comparação entre valores idênticos tem como resultado o próprio valor. A comparação entre valores diferentes resulta em *nil*.

- As regras de **cont1** até **cont8** representam regras de contexto, e definem

uma ordem de aplicação da esquerda para a direita.

Para tornar mais clara a representação de expressões, eventualmente empregamos parênteses para indicar como estão agrupados os termos de subexpressões. Também com o objetivo de melhorar a legibilidade do código de programas, estendemos a sintaxe de λ_{ref} com duas formas derivadas: *sequenciamento* e *expressões let*. A notação de sequenciamento, representada por

$$e_1;\ e_2$$

é um açucar sintático para $(\lambda x_i.e_2)e_1$, onde x_i não aparece em e_2. Esta expressão tem como simples efeito avaliar e_1, descartando o resultado correspondente, e posteriormente avaliar e_2[1]. Expressões let, representadas por

$$\textbf{let}\ x_i = e_1\ \textbf{in}\ e_2$$

são um açucar sintático para $(\lambda x_i.e_2)e_1$, onde x_i geralmente aparece em e_2. Tem como efeito avaliar e_1, substituir todas as ocorrências de x_i em e_2 pelo resultado obtido, e posteriormente avaliar e_2.

Antes de estender a semântica de λ_{ref} com coleta de lixo, precisamos introduzir mais algumas definições e alguns resultados básicos. A notação

$$\mathcal{P}_1 \xrightarrow{R} \mathcal{P}_2$$

indica uma transição qualquer em um conjunto de regras R. Para indicar que um programa $\mathcal{P}_1$ reduz para $\mathcal{P}_2$ seguindo um número finito de transições de um conjunto de regras R, usamos a notação

$$\mathcal{P}_1 \overset{R}{\Longrightarrow} \mathcal{P}_2$$

Para indicar que $\mathcal{P}_2$ é irredutível em relação a R, e $\mathcal{P}_1 \overset{R}{\Longrightarrow} \mathcal{P}_2$, usamos a notação

$$\mathcal{P}_1 \Downarrow_R \mathcal{P}_2$$

Se $\mathcal{P} \Downarrow_R \langle v, H\rangle$ para algum $v \in \mathbf{V}$, dizemos que v é o resultado de $\mathcal{P}$, denotado por $\mathcal{P} \overset{R}{=} v$. Um programa $\mathcal{P}$ é *decidível* (decidable) se existe algum v tal que $\mathcal{P} \overset{R}{=} v$. Caso contrário $\mathcal{P}$ é *não-decidível*.

[1]Obviamente este mecanismo só é útil em linguagens com efeitos colaterais, como é o caso aqui.

PUC-Rio - Certificação Digital Nº 0310845/CA

$LO(e)$ representa o conjunto contendo todos os locais que ocorrem literalmente na expressão e. $LO(H)$, o conjunto de locais que ocorrem em uma ambiente H, é definido como:

$$LO(H) = \bigcup_{e_i \in range(H)} LO(e_i)$$

onde $range(H)$ denota o contra-domínio de H. O conjunto de locais que ocorrem em uma programa $\langle e, H \rangle$ é definido simplesmente como $LO(e) \cup LO(H)$.

Um programa é classificado como *fechado* se todos os locais que ocorrem no mesmo pertencem ao domínio do seu ambiente, ou seja, o programa $\langle e, H \rangle$ é fechado se $LO(e) \cup LO(H) \subseteq dom(H)$. Caso contrário o programa é *aberto*.

Informalmente podemos pensar em um programa fechado como um programa que não contém ponteiros quebrados. Uma propriedade importante desta classe de programas é descrita pelo seguinte lema.

Lemma 5.1. *Para quaisquer programas $\mathcal{P}_1$ e $\mathcal{P}_2$ tal que $\mathcal{P}_1 \xrightarrow{R_{ref}} \mathcal{P}_2$, se $\mathcal{P}_1$ for fechado então $\mathcal{P}_2$ também é fechado.*

Demonstração. Por casos em R_{ref}.

Considere $\mathcal{P}_1 = \langle e_1, H_1 \rangle$ e $\mathcal{P}_2 = \langle e_2, H_2 \rangle$. Locais que não ocorrem em e_1 podem aparecer em e_2 apenas através de transições **alloc** e **deref**. No primeiro caso, o local recém-criado é imediatamente incluído no domínio de H_2. No segundo caso, se $\mathcal{P}_1$ é fechado então qualquer local referenciado por um local que ocorre em e_1 pertence ao domínio de H_1. Como para qualquer transição em R_{ref} temos que $dom(H_1) \subseteq dom(H_2)$, o local referenciado também pertence ao domínio de H_2.

Um local que não ocorre em H_1 pode aparecer em H_2 apenas através de transições **alloc** e **assign**. No segundo caso a referência ao qual está sendo feita a atribuição tem que ocorrer em e_1. Como $\mathcal{P}_1$ é fechado, o local pertence ao domínio de H_1, e por extensão, também ao domínio de H_2. □

Uma *substituição de locais* de l_i por l_j em uma expressão e, ou simplesmente *l-substituição*, é definida como uma substituição literal de todas as ocorrências de l_i por l_j em e. Para evitar colisões de nomes, esta operação é indefinida se

PUC-Rio - Certificação Digital Nº 0310845/CA

$l_j \in LO(e)$. A l-substituição para ambientes é definida de forma semelhante[2]:

$$\{l_i/l_j\}H = \begin{cases} H_\emptyset[l_k \mapsto \{l_i/l_j\}H(l_k)]^{l_k \in dom(H)} & \text{se } l_i \notin dom(H) \\ H_\emptyset[l_j \mapsto \{l_i/l_j\}H(l_i)][l_k \mapsto \{l_i/l_j\}H(l_k)]^{l_k \in dom(H) \setminus \{l_i\}} & \text{caso contrário} \end{cases}$$

onde o símbolo $\setminus$ representa a diferença entre conjuntos. Se $l_j \in LO(H) \cup dom(H)$, $\{l_i/l_j\}H$ é indefinido. Por fim, a l-substituição para programas é definida simplesmente como:

$$\{l_i/l_j\}\,\langle e, H\rangle = \langle \{l_i/l_j\}e, \{l_i/l_j\}H\rangle$$

Para que possamos definir uma noção de equivalência entre programas que não dependa de detalhes de implementação do sistema de memória, introduzimos o conceito de *congruência estrutural*. Dois programas $\mathcal{P}_1$ e $\mathcal{P}_2$ são estruturalmente congruentes, ou simplesmente congruentes, se existir uma sequência finita de α-substituições e de l-substituições que transformem $\mathcal{P}_1$ em $\mathcal{P}_2$. Para indicar que $\mathcal{P}_1$ e $\mathcal{P}_2$ são congruentes usamos a notação $\mathcal{P}_1 \equiv \mathcal{P}_2$.

Congruência estrutural entre expressões é definida de forma semelhante. Duas expressões são congruentes se uma puder ser transformada na outra através de uma sequência finita de α-substituições e de l-substituições.

Uma regra r é *determinística* se para quaisquer transições que seguem r, $\langle e_1, H_1\rangle \rightarrow \langle e_2, H_2\rangle$ e $\langle e_1, H_1\rangle \rightarrow \langle e_3, H_3\rangle$, é sempre verdade que $e_2 \equiv e_3$. Caso contrário r é *não-determinística*. Um conjunto de regras R é determinístico se para qualquer programa $\mathcal{P}$ tal que $\mathcal{P} \Downarrow_R \langle v, H_1\rangle$ e $\mathcal{P} \Downarrow_R \langle e, H_2\rangle$, é sempre verdade que $e \equiv v$. Caso contrário R é não determinístico.

Um conjunto de regras pode ser não-determinístico mesmo se todas as suas regras forem determinísticas. Por outro lado, uma única regra não-determinística é suficiente para fazer com que um conjunto de regras torne-se não-determinístico.

Dois conjuntos de regras R_1 e R_2 são equivalentes, denotado por $R_1 \simeq R_2$, se para qualquer resultado de qualquer programa decidível seguindo um conjunto de regras, existe um resultado congruente para o mesmo programa seguindo o outro conjunto de regras, e vice-versa.

Para provar que programas congruentes são semanticamente equivalentes, basta demonstrar que a congruência estrutural é preservada por R_{ref}. Para isso iremos precisar dos lemas 5.2 e 5.3.

[2]A notação $f(x_i)^{x_i \in \mathbf{X}}$ representa a expressão $f(x_1)f(x_2)...f(x_n)$, para todo $x_i \in \mathbf{X}$. Os termos correspondentes a cada x_i podem aparecer nesta expressão em qualquer ordem.

PUC-Rio - Certificação Digital Nº 0310845/CA

Lemma 5.2. *Para quaisquer programas $\mathcal{P}_1$ e $\mathcal{P}_2$ tal que $\mathcal{P}_1 \xrightarrow{R_{ref}} \mathcal{P}_2$, e para qualquer l-substituição $\{l_i/l_j\}$ definida em $\mathcal{P}_1$, temos que $\{l_i/l_j\}\mathcal{P}_1 \xrightarrow{R_{ref}} \mathcal{P}_3$, onde $\mathcal{P}_3 \equiv \mathcal{P}_2$.*

Demonstração. Por casos em R_{ref}.

É trivial mostrar que $\mathcal{P}_3 = \{l_i/l_j\}\mathcal{P}_2$ para qualquer transição em R_{ref}, exceto **applic** e **alloc**. Estas transições introduzem, respectivamente, uma aleatoriedade no nome de variáveis e de locais.

Transições **applic** são da forma $\langle(\lambda x_i.e_1)v, H\rangle \rightarrow \langle\{x_i/v\}e_2, H\rangle$, onde $e_1 = e_2$, ou e_1 pode ser transformado em e_2 através de uma sequência de α-substituições. Aplicando a substituição $\{l_i/l_j\}$ ao programa $\langle(\lambda x_i.e_1)v, H\rangle$ obtemos:

$\{l_i/l_j\}\,\langle(\lambda x_i.e_1)v, H\rangle = \langle(\{l_i/l_j\}\lambda x_i.e_1)\{l_i/l_j\}v, \{l_i/l_j\}H\rangle \rightarrow$
$\langle\{x_i/(\{l_i/l_j\}v)\}(\{l_i/l_j\}e_2), \{l_i/l_j\}H\rangle = \{l_i/l_j\}\,\langle\{x_i/v\}e_2, H\rangle$

Transições **alloc** são da forma $\langle\mathbf{new}, H\rangle \rightarrow \langle l_k, H[l_k \mapsto nil]\rangle$. Aplicando a substituição $\{l_i/l_j\}$ ao programa $\langle\mathbf{new}, H\rangle$ obtemos:

$\{l_i/l_j\}\,\langle\mathbf{new}, H\rangle = \langle\mathbf{new}, \{l_i/l_j\}H\rangle \rightarrow \langle l_l, \{l_i/l_j\}H[l_l \mapsto nil]\rangle$

Se $l_l \neq l_k$ então

$\langle l_l, \{l_i/l_j\}H[l_l \mapsto nil]\rangle = \{l_i/l_j\}(\{l_k/l_l\}\,\langle l_k, H[l_k \mapsto nil]\rangle)$

caso contrário, se $l_l = l_k$ então

$\langle l_l, \{l_i/l_j\}H[l_l \mapsto nil]\rangle = \{l_i/l_j\}\,\langle l_k, H[l_k \mapsto nil]\rangle)$

□

Lemma 5.3. *Para quaisquer programas $\mathcal{P}_1$ e $\mathcal{P}_2$ tal que $\mathcal{P}_1 \equiv \mathcal{P}_2$, se $\mathcal{P}_1 \xrightarrow{R_{ref}} \mathcal{P}_3$, então $\mathcal{P}_2 \xrightarrow{R_{ref}} \mathcal{P}_4$, onde $\mathcal{P}_4 \equiv \mathcal{P}_3$.*

Demonstração. Primeiramente consideramos o caso especial $\mathcal{P}_1 = \mathcal{P}_2 = \mathcal{P}$. Por casos em R_{ref} não é difícil perceber que para qualquer $\mathcal{P}$ existe no máximo uma regra que define uma possível redução. Como qualquer transição em R_{ref} é determinística, temos que $\mathcal{P}_3 \equiv \mathcal{P}_4$.

A prova do caso geral é obtida a partir do caso especial usando o Lemma 5.2, e considerando-se que abstrações são equivalentes módulo α-substituição. □

Lemma 5.4. *Para quaisquer programas $\mathcal{P}_1$ e $\mathcal{P}_2$ tal que $\mathcal{P}_1 \equiv \mathcal{P}_2$, se $\mathcal{P}_1 \Downarrow_{R_{ref}} \mathcal{P}_3$, então $\mathcal{P}_2 \Downarrow_{R_{ref}} \mathcal{P}_4$, onde $\mathcal{P}_4 \equiv \mathcal{P}_3$.*

PUC-Rio - Certificação Digital Nº 0310845/CA

Demonstração. Por indução. Basta aplicar o Lemma 5.3 em cada passo da derivação, e notar que para qualquer programa em λ_{ref} existe no máximo uma única regra que define a próxima transição. □

Corolário 5.5. *R_{ref} é determinístico.*

5.2 Coleta de Lixo

De maneira geral uma coleta de lixo consiste na remoção de ligações que não afetam o resultado de um programa. Uma definição formal e bem rigorosa deste conceito pode ser feita através da seguinte regra:

$$\begin{aligned} \langle e, H\rangle \rightarrow \langle e, H[l_i \mapsto \bot]\rangle & \\ \text{onde } l_i \in dom(H),\ \langle e, H\rangle \stackrel{R}{=} v_1, & \qquad (\mathbf{gc}) \\ \langle e, H[l_i \mapsto \bot]\rangle \stackrel{R}{=} v_2,\ v_2 \equiv v_1 & \end{aligned}$$

Se uma ligação não afeta o resultado final de um programa, ela pode ser removida (coletada).

Certamente essa regra é um tanto abstrata, e não indica um método prático para descobrir quais ligações são efetivamente coletáveis. Conforme vimos no Capítulo 2, uma alternativa comum e conservadora para determinar ligações coletáveis envolve a construção do grafo de conectividade a partir do conjunto-raiz do programa. Uma ligação que não pertença a este grafo, não pode mais ser recuperada, logo não tem como afetar o resultado do programa.

Iremos nos referir à linguagem λ_{ref} estendida com coleta de lixo como λ_{gc}, e ao respectivo conjunto de regras como R_{gc}. Em λ_{gc} coletamos referências (locais), ou mais precisamente, as ligações associadas às referências. Uma referência é acessível se ela pertence ao conjunto-raiz, que neste caso corresponde ao conjunto de locais que ocorrem na expressão do programa, ou se ela ocorre em qualquer expressão ligada a uma referência que pode ser alcançada a partir do conjunto-raiz.

Considere como exemplo o grafo de conectividade da Figura 5.3, onde o conjunto-raiz contém apenas o local l_i. O local l_k pode ser alcançado através do caminho indicado. O local l_m, por outro lado, não é acessível, sendo assim considerado lixo.

Para determinar a conectividade de uma referência utilizamos a função *reachable*, que indica se um local particular é acessível. Ela é definida como

PUC-Rio - Certificação Digital Nº 0310845/CA

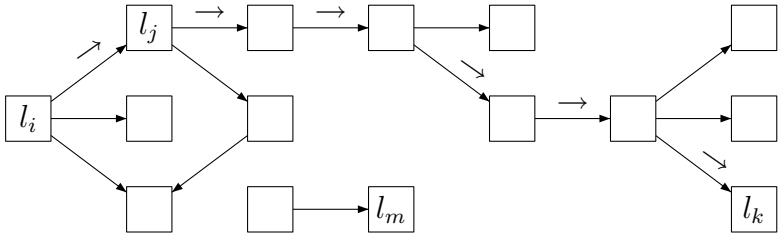

Figura 5.3: Um grafo de conectividade em λ_{ref}.

o menor ponto fixo que satisfaz a expressão

$$reachable(l_i, e, H) = (l_i \in LO(e)) \vee (\exists l_j \in LO(e) \mid reachable(l_i, H(l_j), H))$$

Apesar desta definição ainda ser um pouco abstrata, ela representa claramente o processo de percorrer um grafo de referências registrando quais referências foram encontradas. Uma implementação real de *reachable* pode seguir qualquer algoritmo tradicional de rastreamento, e portanto, é decidível.

Para determinar se um local é coletável (desconexo), definimos a função *dead*. Para ser coletável, além de ser desconexo, o local precisa pertencer ao domínio de H (só faz sentido desalocar dinamicamente aquilo que foi previamente alocado).

$$dead(l_i, e, H) = (l_i \in dom(H)) \wedge \neg reachable(l_i, e, H)$$

A *morte*, tal como definida por *dead*, é invariante em λ_{ref}: ligações desconexas jamais tornam-se acessíveis novamente. Os lemas seguintes descrevem esta propriedade de maneira formal.

Lemma 5.6. *Para qualquer expressão e_1, ambiente H_1, e local l_i, se $dead(l_i, e_1, H_1)$ e $\langle e_1, H_1\rangle \overset{R_{ref}}{\rightarrow} \langle e_2, H_2\rangle$, então $dead(l_i, e_2, H_2)$.*

Demonstração. Por casos em R_{ref} e usando a definição de *dead*.

Suponha a existência de um l_i tal que $dead(l_i, e_1, H_1)$ e $\neg dead(l_i, e_2, H_2)$. Como não existem transições que removem locais de um ambiente, $\neg dead(l_i, e_2, H_2)$ implica em $reachable(l_i, e_2, H_2)$.

O local l_i pode se tornar acessível em $\langle e_2, H_2\rangle$ somente se uma das seguintes condições for satisfeita:

(i) l_i aparece em e_2.

PUC-Rio - Certificação Digital Nº 0310845/CA

(ii) Existe um l_j que ocorre em e_2, e l_i é acessível a partir de $H_2(l_j)$.

A primeira condição pode ser satisfeita apenas se l_i ocorrer em e_1, ou se e_1 conter um termo com a forma $!l_k$ onde l_i ocorre em $H(l_k)$. Em ambos os casos teríamos que $\neg dead(l_i, e_1, H_1)$, o que contradiz a hipótese inicial. Observe que não precisamos considerar **alloc** já que supomos que $l_i \in dom(H_1)$.

Se a segunda condição é satisfeita, l_i não pode ser acessível em $H_1(l_j)$, ou l_j deve estar morto em $\langle e_1, H_1\rangle$. O local l_i pode se tornar acessível em $H_2(l_j)$ somente através de transições **assign**. Neste caso l_i, ou um l_k tal que $reachable(l_i, H_1(l_k), H_1)$, precisaria ocorrer em e_1. Logo l_i não poderia estar morto em $\langle e_1, H_1\rangle$. Finalmente, como provamos acima, se l_j está morto em H_1, ele não pode aparecer em H_2. □

Lemma 5.7. *Para qualquer expressão e_1, ambiente H_1, e local l_i, se $dead(l_i, e_1, H_1)$ e $\langle e_1, H_1\rangle \stackrel{R_{ref}}{\Longrightarrow} \langle e_2, H_2\rangle$, então $dead(l_i, e_2, H_2)$.*

Demonstração. Por indução. Basta aplicar o Lemma 5.6 nos passos da derivação. □

Usando o conceito de rastreamento de referências podemos redefinir coleta de lixo como uma transição que remove do ambiente as ligações que não são acessíveis a partir do conjunto-raiz. A coleta de *uma* ligação (um local) pode ser representada por

$$\langle e, H\rangle \rightarrow \langle e, H[l_i \mapsto \perp]\rangle \quad \text{se } dead(l_i, e, H) \qquad (\mathbf{gc1})$$

Como um local coletado pode ser reintroduzido (reusado) por uma alocação, a introdução da regra **gc1** torna o Lema 5.7 inválido. Não obstante, as múltiplas versões de um local reusado são semanticamente distintas, e o processo de reuso de locais não tem qualquer significado prático, e nem gera nenhum problema na linguagem definida até aqui. Um resultado similar ao Lema 5.4 pode ser provado para R_{gc}. Para efetuar esta demonstração empregamos um resultado análogo ao Lema de Postergação (Postponement Lemma) definido em [50].

Lemma 5.8. *Seja $\stackrel{gc}{\longrightarrow}$ uma transição **gc1** qualquer. Se existem programas $\mathcal{P}_1$, $\mathcal{P}_2$ e $\mathcal{P}_3$ tal que $\mathcal{P}_1 \stackrel{gc}{\longrightarrow} \mathcal{P}_2 \stackrel{R_{ref}}{\longrightarrow} \mathcal{P}_3$, então existe um programa $\mathcal{P}_4$ tal que $\mathcal{P}_1 \stackrel{R_{ref}}{\longrightarrow} \mathcal{P}_4 \stackrel{gc}{\longrightarrow} \mathcal{P}_5$, onde $P_5 \equiv P_3$.*

PUC-Rio - Certificação Digital Nº 0310845/CA

Demonstração. Por casos em R_{gc} e usando o Lema 5.6.

Exceto por **assign**, a aplicabilidade de qualquer regra em R_{ref} a um dado programa depende apenas da forma sintática da expressão que compõe o programa. Como transições **gc1** nunca mudam a forma de uma expressão, elas não afetam a aplicabilidade das regras de R_{ref}.

A aplicabilidade de **assign** também não é afetada por transições **gc1**, já que estas não adicionam novos locais e nem removem locais acessíveis do ambiente do programa.

Ainda que a aplicabilidade de **alloc** não seja afetada por transições **gc1**, o seu resultado pode ser: o nome de locais coletados pode mudar após uma coleta de lixo. Apesar disso, todos os programas resultantes são congruentes.

A prova é facilmente completada usando o Lema 5.6. □

Lemma 5.9. *Para qualquer programa $\mathcal{P}$, se existe um valor v_1 tal que $\mathcal{P} \overset{R_{gc}}{=} v_1$, então $\mathcal{P} \overset{R_{ref}}{=} v_2$, onde $v_2 \equiv v_1$.*

Demonstração. Considere a sequência finita de reduções:

$$\mathcal{P} \xrightarrow{R_{gc}} \mathcal{P}_1 \xrightarrow{R_{gc}} ... \xrightarrow{R_{gc}} \mathcal{P}_{n-1} \xrightarrow{R_{gc}} \mathcal{P}_n$$

e suponha que $\mathcal{P}_n = \langle v_1, H_n \rangle$.

Usando o Lema 5.8 e indução podemos escrever uma sequência alternativa de reduções na qual todas as transições de coleta de lixo são realizadas na parte final da sequência:

$$\mathcal{P} \xrightarrow{R_{ref}} \mathcal{P}'_1 \xrightarrow{R_{ref}} ... \xrightarrow{R_{ref}} \mathcal{P}'_i \xrightarrow{gc} ... \xrightarrow{gc} \mathcal{P}'_{n-1} \xrightarrow{gc} \mathcal{P}'_n$$

onde $\mathcal{P}'_n = \langle v'_1, H'_n \rangle$ e $v'_1 \equiv v_1$.

Considerando que transições **gc1** nunca mudam a expressão que compõe o programa temos que $\mathcal{P}'_i = \langle v'_1, H'_i \rangle$. Por casos nos elementos de R_{ref}, e usando o Lema 5.4, é fácil verificar que $\mathcal{P} \overset{R_{ref}}{=} v_2$, onde $v_2 \equiv v_1$. □

Usando este lema podemos provar que R_{ref} e R_{gc} são equivalentes, ou em outras palavras, que estender R_{ref} com coleta de lixo não tem nenhum efeito semântico visível.

Lemma 5.10. $R_{ref} \simeq R_{gc}$.

Demonstração. Considere qualquer programa decidível $\mathcal{P}$ tal que $\mathcal{P} \Downarrow_{R_{ref}} \langle v, H \rangle$. Como $R_{gc} \supset R_{ref}$, seguindo apenas regras em R_{ref} é sempre possível replicar a mesma sequência de transições tal que $\mathcal{P} \Downarrow_{R_{gc}} \langle v, H \rangle$.

A prova é completada usando o Lema 5.9. □

PUC-Rio - Certificação Digital Nº 0310845/CA

Corolário 5.11. *R_{gc} é determinístico.*

Uma transição **gc1** representa a coleta de uma única referência. Um *ciclo de coleta de lixo* (representado por $\stackrel{gc}{\Longrightarrow}$) em um programa $\mathcal{P}$ é definido como uma sequência ininterrupta de transições **gc1** tal que se $\mathcal{P} \stackrel{gc}{\Longrightarrow} \langle e, H\rangle$, então todos os locais no domínio de H são acessíveis.

Lemma 5.12. *Ciclos de coleta de lixo sempre terminam.*

Demonstração. Qualquer programa tem um número finito de locais desconexos, que diminui em um para cada transição **gc1**. □

Uma alternativa simples para fazer com que a coleta de lixo sempre aconteça em ciclos é substituir a regra **gc1** pela regra

$$\begin{aligned} &\langle e, H_1\rangle \rightarrow \langle e, H_2\rangle \\ &\qquad\qquad \text{se } \exists l_i \mid dead(l_i, e, H_1) \qquad\qquad (\mathbf{gc2}) \\ &\qquad\qquad \text{onde } H_2 = H_\emptyset[l_j \mapsto H_1(l_j)]^{l_j \in \{l_k \mid reachable(l_k, e, H_1)\}} \end{aligned}$$

Na transição acima o ambiente original é substituído por uma cópia modificada na qual apenas os locais acessíveis são definidos, descartando-se assim, de maneira implícita, todas as ligações coletáveis[3]. Cabe observar ainda que apesar da condição

$$\exists l_i \mid dead(l_i, e, H_1)$$

ser efetivamente desnecessária para disparar um ciclo de coleta, ela impede que **gc2** seja aplicado indefinidamente.

Se a coleta de lixo acontecesse sempre em ciclos, seguindo uma regra como **gc2**, seria possível garantir que este processo nunca gera ponteiros quebrados. Em λ_{gc} no entanto a coleta de lixo é efetuada através de **gc1**, e programas fechados podem dar origem a programas abertos.

Considere por exemplo um programa com dois objetos que contêm referências mútuas. Se apenas um dos objetos for coletado, o objeto remanescente irá apontar para um objeto coletado, ou seja, esta coleta resulta em um ponteiro quebrado. Note que isso acontece sempre que um objeto referenciado por outro é coletado primeiro. Não obstante, como o ponteiro quebrado é inacessível, esse tipo de situação não representa um problema efetivo.

[3]Este padrão é semelhante à dinâmica de coletores stop-and-copy.

PUC-Rio - Certificação Digital Nº 0310845/CA

Desse modo, para que possamos continuar considerando programas abertos como incorretos, devemos modificar a nossa definição de programas fechados. Um programa $\langle e, H\rangle$ é fechado se e somente se a seguinte condição for satisfeita

$$\{l_i \mid reachable(l_i, e, H)\} \subseteq dom(H)$$

A partir desta definição podemos provar um resultado semelhante ao Lema 5.1 para λ_{gc}.

Lemma 5.13. *Para quaisquer programas $\mathcal{P}_1$ e $\mathcal{P}_2$ tal que $\mathcal{P}_1 \xrightarrow{R_{gc}} \mathcal{P}_2$, se $\mathcal{P}_1$ for fechado então $\mathcal{P}_2$ também é fechado.*

Demonstração. Por casos em R_{gc}.

Considere $\mathcal{P}_1 = \langle e_1, H_1\rangle$ e $\mathcal{P}_2 = \langle e_2, H_2\rangle$. Locais que não ocorrem em e_1 podem aparecer em e_2 apenas através de transições **alloc** e **deref**. No primeiro caso, o local recém-criado é imediatamente incluído no domínio de H_2. No segundo caso, se $\mathcal{P}_1$ é fechado então qualquer local referenciado por um local que ocorre em e_1 pertence ao domínio de H_1. Como para esta transição temos que $dom(H_1) = dom(H_2)$, o local referenciado também pertence ao domínio de H_2.

Um local que não ocorre em H_1 pode aparecer em H_2 apenas através de transições **alloc** e **assign**. No segundo caso a referência ao qual está sendo feita a atribuição tem que ocorrer em e_1. Como $\mathcal{P}_1$ é fechado, o local pertence ao domínio de H_1, e por extensão, também ao domínio de H_2 (para esta transição também temos que $dom(H_1) = dom(H_2)$).

A prova é completada considerando-se que uma coleta de lixo (**gc1**) nunca remove locais que são acessíveis a partir da expressão do programa. □

5.3 Finalizadores

Iremos representar finalizadores como rotinas que são automaticamente executadas após um local tornar-se desconexo. Para associar um finalizador a um local estendemos λ_{gc} com o operador **finalize**: a expressão **finalize** l_i $\lambda x_i.e$ registra a função $\lambda x_i.e$ como um finalizador para o local l_i.

$$\langle \textbf{finalize}\ l_i\ v, H, F\rangle \rightarrow \langle nil, H, F[l_i \mapsto v]\rangle \quad \text{onde } l_i \in dom(H) \qquad (\textbf{fin-reg})$$

PUC-Rio - Certificação Digital Nº 0310845/CA

Para armazenar os finalizadores registrados pelo programa cliente adicionamos um novo ambiente (F) ao contexto de programas[4]. A linguagem λ_{gc} estendida com finalizadores será denominada de λ_{fin}, e o conjunto de regras correspondente será indicado por R_{fin}.

A execução de finalizadores deve acontecer de forma concorrente ou intercalada com o processamento das expressões de um programa. Um alternativa simples para modelar esta dinâmica é usar a notação de sequenciamento, conforme descrito na regra abaixo (**fin-exec1**). Um finalizador associado a um objeto morto pode ser invocado a qualquer momento durante a execução de um programa, recebendo como único parâmetro a referência a ser finalizada.

$$\langle e, H, F\rangle \rightarrow \langle F(l_i)l_i;\ e, H, F[l_i \mapsto \bot]\rangle \quad \text{se } (l_i \in dom(F)) \wedge dead(l_i, e, H) \qquad (\textbf{fin-exec1})$$

Como consequência da forma com que finalizadores são invocados, a qual em um certo sentido é análoga a dinâmica de escalonamento de aplicações multithread, linguagens de programação que suportam finalizadores normalmente não são determinísticas. Este certamente é o caso aqui.

Lemma 5.14. *R_{fin} é não-determinístico.*

Demonstração. Por contra-exemplo.

Considere o programa

let $x_i =$ **new in**
 let $x_j =$ **new in** (
  finalize x_j $(\lambda x_k.(x_i := nil))$;
  $x_i := \lambda x_k.x_k$;
  $!x_i$
)

que pode ser reduzido para $\langle (!l_i, H_\emptyset[l_i \mapsto \lambda x_k.x_k], F[l_j \mapsto \lambda x_k.(l_i := nil)]\rangle$. Duas sequências distintas de transições em R_{fin} que levam a dois resultados não equivalentes são:

(i) $\langle (!l_i, H_\emptyset[l_i \mapsto \lambda x_k.x_k], F[l_j \mapsto \lambda x_k.(l_i := nil)]\rangle$
$\overset{\textbf{deref}}{\rightarrow} \langle (\lambda x_k.x_k, H_\emptyset[l_i \mapsto \lambda x_k.x_k], F[l_j \mapsto \lambda x_k.(l_i := nil)]\rangle$

[4]Apesar da nossa omissão, todas as demais regras definidas até aqui devem ser modificadas para incluir F. A partir de agora iremos omitir também novas regras de contexto.

PUC-Rio - Certificação Digital Nº 0310845/CA

$$\stackrel{R_{fin}}{\Longrightarrow} \langle(\lambda x_k.x_k, H_\emptyset[l_i \mapsto nil], F[l_j \mapsto \bot]\rangle$$

(ii) $\langle(!l_i, H_\emptyset[l_i \mapsto \lambda x_k.x_k], F[l_j \mapsto \lambda x_k.(l_i := nil)]\rangle$
$\stackrel{\textbf{fin-exec1}}{\longrightarrow} \langle(\lambda x_k.(l_i := nil))l_j; !l_i, H_\emptyset[l_i \mapsto \lambda x_k.x_k], F[l_j \mapsto \bot]\rangle$
$\stackrel{R_{fin}}{\Longrightarrow} \langle nil, H_\emptyset[l_i \mapsto nil], F[l_j \mapsto \bot]\rangle$

Obviamente, $\lambda x_k.x_k \not\equiv nil$. □

Corolário 5.15. $R_{fin} \not\simeq R_{gc}$

Ao ser invocado através de uma transição **fin-exec1**, um finalizador sempre recebe como parâmetro um local desconexo. Além disso, como em λ_{fin} um finalizador na verdade representa um fechamento, ele pode conter referências para outras referências, algumas das quais também encontram-se inacessíveis. Isto introduz a possibilidade de ressurreição de ligações: se durante a finalização uma referência desconexa for atribuída a uma referência acessível, a referência atribuída pode permanecer acessível após o fim da execução do finalizador.

Para evitar problemas relacionados a ponteiros quebrados é necessário postergar a coleta de todas as referências que podem ser acessadas a partir de um finalizador. Além disso, como a referência que está sendo finalizada é passada como um parâmetro para o finalizador, este deve ser executado antes que a respectiva referência seja efetivamente coletada[5]. Para modelar esta semântica, redefinimos uma coleta de lixo como:

$$\begin{aligned} \langle e, H, F\rangle \rightarrow \langle e, H[l_i \mapsto \bot], F\rangle & \\ \text{se } dead(l_i, e, H) \wedge (l_i \notin dom(F)) & \qquad (\textbf{gc3}) \\ \wedge(\nexists l_j \in dom(F) \mid reachable(l_i, F(l_j)l_j, H)) & \end{aligned}$$

Com a inclusão desta nova regra, sempre que um finalizador é associado a uma ligação, a coleta desta ligação é postergada. Conforme já mencionamos, esse atraso é bastante comum em linguagens orientadas a objetos, e pode ter um impacto relevante no desempenho de aplicações. Mas ele é necessário pois impede que rotinas de finalização ressuscitem objetos coletados, garantindo uma semântica correta.

[5]Como uma alternativa a essa semântica poderíamos redefinir o conceito de conectividade para que finalizadores registrados fossem incluídos no conjunto-raiz, como acontece por exemplo em Ruby. Porém isso faria com que objetos finalizáveis que formassem ciclos de referências nunca fossem coletados. Conforme iremos discutir mais adiante, a nossa opção permite evitar esse problema.

PUC-Rio - Certificação Digital Nº 0310845/CA

Lemma 5.16. *Para quaisquer programas $\mathcal{P}_1$ e $\mathcal{P}_2$ tal que $\mathcal{P}_1 \xrightarrow{R_{fin}} \mathcal{P}_2$, se $\mathcal{P}_1$ for fechado então $\mathcal{P}_2$ também é fechado.*

Demonstração. Por casos em R_{fin}.

A prova é idêntica a prova do Lema 5.13, exceto pela regra de coleta de lixo e pela possibilidade de ressurreição. É fácil perceber que **gc3** impede a coleta de objetos que podem ser ressuscitados, garantindo que a execução de um finalizador não torna acessíveis locais que já foram coletados. Para concluir a prova basta considerar também que uma coleta de lixo nunca remove locais acessíveis. □

Como vimos em capítulos anteriores, em algumas aplicações a ordem de invocação de finalizadores é importante. Coletores de lixo podem ordenar a invocação de finalizadores pelo menos de duas formas distintas: usando informações topológicas ou cronológicas (ordem de elaboração). A ordem topológica normalmente é mais significativa do ponto de vista da semântica da aplicação, sendo capaz de garantir um grau maior de correção. Um objeto deve ser mantido em um estado válido enquanto existirem outros objetos que podem acessá-lo (mesmo que apenas através dos respectivos finalizadores). Assim um finalizador só deve ser executado se todos os finalizadores que se referem ao objeto já tiverem sido executados. Esta semântica pode ser representada substituindo-se **fin-exec1** pela seguinte transição

$$\begin{aligned}\langle e, H, F\rangle \rightarrow \langle F(l_i)l_i;\ e, H, F[l_i \mapsto \bot]\rangle & \\ \text{se } (l_i \in dom(F)) \wedge dead(l_i, e, H) & \qquad (\textbf{fin-exec2}) \\ \wedge(\nexists l_j \in dom(F) \mid reachable(l_i, F(l_j)l_j, H)) &\end{aligned}$$

Existe porém um problema grave com a regra acima: ela não permite que finalizadores que referenciam-se formando ciclos sejam invocados, impedindo a coleta dos objetos finalizáveis. Como nesta situação não existe uma solução ideal para determinar qual a melhor ordem de invocação, alguns autores optam por deixar esse gerenciamento de ciclos sob a responsabilidade exclusiva do programador, mesmo que isso leve a vazamentos de memória. Em λ_{fin} iremos adotar uma linha diferente, optando por eliminar completamente essa potencial fonte de erros. Sempre que existir um ciclo entre finalizadores, iremos quebrar o ciclo através da invocação arbitrária de um destes finalizadores. Para descrever esta semântica

PUC-Rio - Certificação Digital Nº 0310845/CA

substituímos **fin-exec2** por

$$\begin{aligned}\langle e, H, F\rangle \rightarrow \langle F(l_i)l_i;\ e, H, F[l_i \mapsto \bot]\rangle \\ \text{se } (l_i \in dom(F)) \wedge dead(l_i, e, H) \\ \wedge(\nexists l_j \in dom(F) \mid reachable(l_i, F(l_j)l_j, H) \\ \wedge \neg reachable(l_j, F(l_i)l_i, H))\end{aligned} \qquad (\textbf{fin-exec3})$$

Por fim, vale ressaltar que na semântica desenvolvida nesta seção, mesmo que uma referência ressuscite, o finalizador correspondente só é executado uma única vez. Nada impede porém que o finalizador seja reabilitado para uma referência ressuscitada. Em Java, aparentemente por razões arbitrárias, finalizadores não podem ser reativados após terem sido executados.

5.4 Referências Fracas

Para não sobrecarregar a notação usada aqui, dificultando desnecessariamente a compreensão do leitor, iremos desenvolver a semântica de referências fracas a partir de λ_{gc} ao invés de λ_{fin}. Não existe porém qualquer incompatibilidade entre as semânticas de finalizadores e referências fracas, e a interação entre estas, conforme já discutimos, é mínima.

Podem ser consideradas ao menos três alternativas distintas para adicionar referências fracas a λ_{gc}:

(i) Introduzir um novo conjunto de valores (um conjunto de locais especiais) para representar referências fracas. Fraqueza seria assim uma propriedade estática de objetos[6].

(ii) Inserir uma marcação sintática (tag) antes dos locais que devem ser tratados como referências fracas. Observe que neste caso os valores associados a uma referência só seriam efetivamente ignorados pelo coletor de lixo se todas as ocorrências desta referência fossem marcadas. Esta solução é flexível, mas um tanto complexa.

(iii) Estender o contexto de programas com um conjunto que registra quais locais representam referências fracas.

A última alternativa é tanto simples quanto flexível, sendo adotada aqui. Para implementá-la estendemos o contexto de programas com um conjunto auxiliar W,

[6]Essa opção é similar à definição de um tipo específico para representar referências fracas.

PUC-Rio - Certificação Digital Nº 0310845/CA

que registra quais locais estão marcados como fracos (representam referências fracas). Para criar uma referência fraca introduzimos o operador de atribuição fraca ($\overset{w}{:=}$), que efetua uma atribuição ordinária, e marca como fraco o local do lado esquerdo do operador de atribuição.

Devemos ressaltar que existe uma pequena mas importante diferença entre este modelo e a semântica de referências fracas adotada por Java e por C#. Nessas linguagens referências fracas são representadas por instâncias de classes específicas, e podem referenciar no máximo um único objeto. Em λ_{gc} uma referência fraca é criada marcando-se uma referência ordinária como fraca, de forma semelhante ao que acontece em Perl. Além disso, uma referência fraca é ligada a uma expressão sintática, que pode conter um ou mais locais, ou até mesmo nenhum.

Ainda assim não é difícil representar uma semântica como a de Java usando o nosso modelo. Para isso basta apenas definir uma operação atômica que cria um local, imediatamente o marca como fraco, e o associa a um valor. Esta operação pode ser definida como

$$new_{weak} = \lambda x_i.(\textbf{let } x_j = \textbf{new in } (x_j \overset{w}{:=} x_i;\ x_j))$$

A expressão $new_{weak}\ v$ cria uma referência fraca para o valor v, de forma semelhante a chamada `new WeakReference(obj)` em Java[7].

Para representar como referências fracas são criadas, introduzimos as seguintes regras, as quais substituem **assign**.

$$\langle l_i := v, H, W\rangle \rightarrow \langle v, H[l_i \mapsto v], W \setminus \{l_i\}\rangle \quad \text{se } l_i \in dom(H) \qquad (\textbf{s-assign})$$

$$\left\langle l_i \overset{w}{:=} v, H, W \right\rangle \rightarrow \langle v, H[l_i \mapsto v], W \cup \{l_i\}\rangle \quad \text{se } l_i \in dom(H) \qquad (\textbf{w-assign})$$

onde, conforme já mencionamos, programas são estendidos com um ambiente adicional (um conjunto W) para rastrear locais marcados como referências fracas[8].

Por definição, referências fracas devem ser ignoradas pelo coletor de lixo durante o processo de rastreamento. A forma mais natural de incorporar esta regra

[7]Para fazer com que uma referência fraca aponte para um único objeto, poderíamos adotar a solução (i) descrita acima, definindo a semântica correspondente de forma apropriada. Entretanto não iremos considerar esta opção, que é menos expressiva do que a semântica que vamos descrever.

[8]Naturalmente as demais regras definidas até aqui devem ser modificadas para incluir W.

PUC-Rio - Certificação Digital Nº 0310845/CA

em nossa semântica é modificando a definição de dead:

$$dead_w(l_i, e, H, W) = (l_i \in dom(H)) \wedge \neg stReachable(l_i, e, H, W)$$

onde *stReachable* (strongly reachable) indica se um local é acessível através de pelo menos um caminho que não inclui referências fracas[9]. Esta função é definida como o menor ponto fixo que satisfaz a relação

$$\begin{aligned} stReachable(l_i, e, H, W) = (l_i \in LO(e)) \\ \vee (\exists l_j \in LO(e) \mid (l_j \notin W) \\ \wedge stReachable(l_i, H(l_j), H, W)) \end{aligned}$$

Durante a avaliação de um programa, qualquer local acessível através de uma referência fraca pode tornar-se fracamente conexo, e eventualmente ser coletado. Mas se uma referência fraca é dereferenciada, referências mortas, e mesmo referências já coletadas, podem reaparecer na expressão do programa. Quando uma referência morta reaparece, outras referências mortas acessíveis a partir da mesma também ressuscitam. Considere por exemplo o programa inicial

$$\begin{array}{l} \textbf{let}\ \ x_i = \textbf{new}\ \ \textbf{in} \\ \ \textbf{let}\ \ x_j = \textbf{new}\ \ \textbf{in}\ (\\ \quad x_i := \lambda x_k.x_k; \\ \quad x_j \overset{w}{:=} \lambda x_l.!x_i; \\ \quad (\textbf{let}\ \ x_m :=!x_j\ \ \textbf{in} \\ \quad\ x_m?\ x_m : \lambda x_n.nil)nil \\ \ ) \end{array}$$

que é redutível para

$$\langle (\textbf{let}\ x_m =!l_j\ \textbf{in}\ (x_m?\ x_m : \lambda x_n.nil))nil, H_\emptyset[l_i \mapsto \lambda x_k.x_k][l_j \mapsto \lambda x_l.!l_i], \{l_j\}\rangle$$

Nesse programa o local l_i está inacessível (no sentido forte). Porém, se aplicarmos a regra **deref** como a próxima transição, ele irá ressuscitar.

De forma semelhante ao que acontece com finalizadores, uma ressurreição é segura (o programa cliente só pode ter acesso a células de memória previamente

[9]Para diferenciar essa definição daquela apresentada na Seção 5.2 adicionamos o subscrito w (weak). Podemos considerar esta definição de morte mais fraca, pois objetos mortos podem ser eventualmente acessados (e ressuscitados) pelo programa cliente através de referências fracas.

PUC-Rio - Certificação Digital Nº 0310845/CA

alocadas) desde que o local ressuscitado não tenha sido coletado. Dois problemas distintos podem ocorrer quando uma referência fraca, associada a um local coletado, é dereferenciada: ponteiros quebrados e conflito de locais.

Qualquer referência, acessível exclusivamente através de caminhos que passam por referências fracas, pode eventualmente ser coletada, gerando assim um problema de ponteiros quebrados. Considere novamente o grafo de conectividade apresentado na Figura 5.3 (página 85), e suponha que $l_i \in W$. Se l_j ou mesmo l_k for coletado, o grafo cuja raiz é l_i pode ser considerado inseguro. Para evitar problemas desta natureza a semântica da linguagem deve incluir um mecanismo que bloqueia o acesso do programa cliente a referências que possam levar a ponteiros quebrados.

O conflito de locais é um problema específico do nosso modelo, e ocorre quando um local acessível através de uma referência fraca é coletado, e posteriormente é reintroduzido através de uma nova alocação (**alloc**). O mesmo local passa então a representar duas ligações distintas: uma já coletada e outra recém-alocada. Estas ligações obviamente não tem qualquer relação semântica, e o uso de uma referência comum a ambas pode levar o programa a se comportar de forma indesejada e imprevisível.

Existem duas opções para evitar os problemas que acabamos de descrever:

- Limpar todas as referências fracas que dão acesso a locais fracamente conexos.
- Limpar todas as referências fracas que dão acesso a locais que foram coletados.

A segunda opção aparentemente é mais flexível, já que possibilita o acesso ao objeto até o último instante antes dele ser coletado. Na prática porém o rastreamento de objetos coletáveis e a desalocação dos mesmos são operações casadas. Portanto, as duas alternativas acima acabam sendo equivalentes.

Para representar como uma referência fraca é limpa iremos coletar inicialmente apenas locais que são realmente inacessíveis, ou seja, que não podem ser acessados nem mesmo através de referências fracas. Para isso podemos usar uma regra semelhante a **gc1**, mas onde o local coletado é removido de W. Caso essa remoção não fosse efetuada, novas alocações poderiam resultar em referências fracas ao invés de referências ordinárias. Como indicado abaixo, uma coleta de lixo é disparada somente após um objeto tornar-se inacessível no sentido mais rigoroso deste

PUC-Rio - Certificação Digital Nº 0310845/CA

conceito.

$$\langle e, H, W\rangle \rightarrow \langle e, H[l_i \mapsto \perp], W \setminus \{l_i\}\rangle \quad \text{se } dead(l_i, e, H) \qquad \textbf{(gc4)}$$

Para que objetos acessíveis exclusivamente através de referências fracas possam eventualmente ser coletados é necessário limpar as respectivas referências. Uma referência fraca deve ser limpa quando ela levar a um local que não é fortemente acessível a partir do conjunto-raiz. Isto pode ser representado por

$$\langle e, H, W\rangle \rightarrow \langle e, H[l_i \mapsto nil], W \setminus \{l_i\}\rangle \quad \text{se } (l_i \in W) \wedge (\exists l_j \in LO(H(l_i)) \mid dead_w(l_j, e, H, W)) \qquad \textbf{(w-clear)}$$

Observe que se um local fortemente acessível a partir de uma referência fraca l_i estiver morto, ele estará necessariamente conectado a um local que ocorre em $H(l_i)$. Consequentemente, para determinar se uma referência fraca deve ser limpa, basta verificar os locais que ocorrem em sua ligação.

Como mencionamos, uma outra alternativa para representar essa semântica é limpar referências fracas que levam a objetos coletados. Uma possível solução é coletar objetos inacessíveis e simultaneamente limpar as referências fracas associadas a eles. Para descrever esta dinâmica podemos usar a regra abaixo, que define a coleta de lixo de um único objeto, bem como a limpeza de todas as referências fracas que permitem acessá-lo.

$$\langle e, H, W\rangle \rightarrow \left\langle e, H[l_i \mapsto \perp][l_j \mapsto nil]^{l_j \in S}, W \setminus (\{l_i\} \cup S)\right\rangle \quad \text{se } dead_w(l_i, e, H, W) \quad \text{onde } S = \{l_j \mid (l_j \in W) \wedge stReachable(l_i, H(l_j), H, W)\} \qquad \textbf{(gc5)}$$

O uso da função *stReachable* como pré-condição desta transição evita a limpeza desnecessária de referências fracas que referem-se a locais acessíveis diretamente através de outras referências fracas. Considere novamente a Figura 5.3, e suponha que l_i e l_j são fracos. Se l_k for coletado, esta pré-condição garante que somente l_j será eventualmente limpo.

Na semântica descrita por **gc4** uma referência fraca evita a coleta de objetos referenciados até a sua limpeza. Na prática porém, como a limpeza e a descoberta de objetos coletáveis acontece de forma atômica, referências fracas a princípio não

PUC-Rio - Certificação Digital Nº 0310845/CA

interferem com o ciclo de vida de objetos (diferentemente do que acontece com finalizadores).

Se nos referirmos a λ_{gc} estendida com referências fracas como λ_{weak}, e o respectivo conjunto de regras como R_w, podemos provar que a semântica de λ_{weak} é correta através dos seguintes lemas.

Lemma 5.17. *Para quaisquer programas $\mathcal{P}_1$ e $\mathcal{P}_2$ tal que $\mathcal{P}_1 \xrightarrow{R_w} \mathcal{P}_2$, se $\mathcal{P}_1$ for fechado então $\mathcal{P}_2$ também é fechado.*

Demonstração. Por casos em R_w.

A prova é idêntica a prova do Lema 5.13, exceto pela regra de coleta de lixo. Para completar esta prova é necessário considerar apenas que uma coleta de lixo através de **gc4** nunca remove locais que são acessíveis a partir da expressão do programa. Note que esta regra impede, em um sentido rigoroso (usando *dead* ao invés de $dead_w$), a ressurreição de objetos através de referências fracas, e evita ainda o reuso de nomes. □

Lemma 5.18. *Em λ_{weak}, qualquer local fracamente morto em um programa fechado pode eventualmente ser coletado.*

Demonstração. Considere um programa fechado $\langle e, H, W\rangle$ e um objeto l_i tal que $dead_w(l_i, e, H, W)$.

Se l_i não é acessível, então ele pode ser imediatamente coletado através de uma transição **gc4**.

Se l_i é fracamente acessível, então existe um conjunto não nulo de locais S tal que para qualquer local $l_j \in S$ temos que $reachable(l_j, e, H)$, $l_j \in W$ e $stReachable(l_i, H(l_j), H, W)$. Pela definição de *stReachable*, para cada $l_j \in S$ temos que $l_i \in LO(H(l_j))$, ou existe um l_k tal que $l_k \in LO(H(l_j))$, $stReachable(l_i, H(l_k), H, W)$ e $dead_w(l_k, e, H, W)$. Isso implica que todos os locais em S estão sujeitos a uma transição **w-clear**. Se todos os locais em S forem limpos, l_i não será mais acessível, podendo ser coletado através de uma transição **gc4**. □

Como locais fracamente acessíveis podem ser ressuscitados, o resultado de um programa pode depender de quando (ou mesmo se) uma ou mais coletas de lixo serão efetuadas. Diferentes sequências de reduções eventualmente irão levar a diferentes resultados. Formalizamos esta conclusão através do seguinte lema.

Lemma 5.19. *R_{weak} é não-determinístico.*

PUC-Rio - Certificação Digital Nº 0310845/CA

Demonstração. Por contra-exemplo. Considere mais uma vez o programa

$$\langle (\textbf{let } x_i = !l_j \textbf{ in } (x_i?\ x_i : \lambda x_j.nil))nil, H_\emptyset[l_i \mapsto \lambda x_l.x_l][l_j \mapsto \lambda x_k.!l_i], \{l_j\}\rangle$$

Duas sequências de transições em R_{weak} que levam a resultados distintos são:

(i) $\langle (\textbf{let } x_i = !l_j \textbf{ in } (x_i?\ x_i : \lambda x_j.nil))nil, H_\emptyset[l_i \mapsto \lambda x_l.x_l][l_j \mapsto \lambda x_k.!l_i], \{l_j\}\rangle$
$\overset{\textbf{w-clear}}{\rightarrow} \langle (\textbf{let } x_i = !l_j \textbf{ in } (x_i?\ x_i : \lambda x_j.nil))nil, H_\emptyset[l_i \mapsto \lambda x_l.x_l][l_j \mapsto nil], \emptyset\rangle$
$\overset{\textbf{gc4}}{\rightarrow} \langle (\textbf{let } x_i = !l_j \textbf{ in } (x_i?\ x_i : \lambda x_j.nil))nil, H_\emptyset[l_i \mapsto \bot][l_j \mapsto nil], \emptyset\rangle$
$\overset{\textbf{deref}}{\rightarrow} \langle (\textbf{let } x_i = nil \textbf{ in } (x_i?\ x_i : \lambda x_j.nil))nil, H_\emptyset[l_i \mapsto \bot][l_j \mapsto nil], \emptyset\rangle$
$\overset{R_{weak}}{\Longrightarrow} \langle nil, H_\emptyset, \emptyset\rangle$

(ii) $\langle (\textbf{let } x_i = !l_j \textbf{ in } (x_i?\ x_i : \lambda x_j.nil))nil, H_\emptyset[l_i \mapsto \lambda x_l.x_l][l_j \mapsto \lambda x_k.!l_i], \{l_j\}\rangle$
$\overset{\textbf{deref}}{\rightarrow} \langle (\lambda x_k.!l_i?\ \lambda x_k.!l_i : \lambda x_j.nil)nil, H_\emptyset[l_i \mapsto \lambda x_l.x_l][l_j \mapsto \lambda x_k.!l_i], \{l_j\}\rangle$
$\overset{R_{weak}}{\Longrightarrow} \langle \lambda x_l.x_l, H_\emptyset, \emptyset\rangle$

Obviamente $\lambda x_l.x_l \not\equiv nil$. □

Corolário 5.20. $R_{gc} \not\subset R_w$.

O modelo desenvolvido nesta seção permite-nos expressar de forma simples a semântica de funções que modificam a força de uma referência, como por exemplo, as funções `weaken` e `makeWeak` encontradas respectivamente em Perl e em GNU Smalltalk. Esta semântica é equivalente a atribuir o valor ligado a uma referência a própria referência, empregando um operador de atribuição fraco ou normal, conforme o resultado desejado. Para transformar um referência normal em fraca basta usar a expressão

$$l_i :\overset{w}{=} !l_i$$

O efeito oposto pode ser obtido com a expressão

$$l_i := !l_i$$

Uma referência suave, cujo valor só é coletado após a utilização da memória ultrapassar um limite pré-estabelecido, pode ser modelada empregando uma função auxiliar que informa a memória disponível para a aplicação. A semântica correspondente seria descrita simplesmente acrescentando um teste de disponibilidade de memória à regra **w-clear**. A referência suave é limpa, e os objetos referenciados tornam-se coletáveis somente se existir pouca memória livre.

PUC-Rio - Certificação Digital Nº 0310845/CA

5.4.1 Mecanismos de Notificação

Considerando-se que filas de notificação podem ser facilmente implementadas através do uso de callbacks (cada callback insere a referência fraca correspondente na fila), iremos abordar aqui apenas a semântica de notificação através de callbacks.

Para representar a invocação de callbacks, que devem ser executados de forma concorrente com o processamento da expressão do programa, usamos uma semântica semelhante àquela definida para finalizadores. Um callback é executado como uma nova aplicação que é inserida na expressão do programa (através da notação de sequenciamento) após a referência fraca correspondente ser limpa pelo coletor de lixo.

Para armazenar os callbacks associados a referências fracas estendemos o contexto de um programa com um conjunto C[10]. Para registrar um callback introduzimos o operador **notify**, usado de forma semelhante ao operador **finalize**. A expressão **notify** $l_i\ \lambda x_i.e$ registra a função $\lambda x_i.e$ como o callback que deve ser invocado quando a referência fraca l_i for limpa.

$$\langle \textbf{notify}\ l_i\ v, H, W, C\rangle \rightarrow \langle nil, H, W, C[l_i \mapsto v]\rangle \quad \text{onde } l_i \in dom(H),\ l_i \in W \qquad (\textbf{cb-reg})$$

A limpeza de referências e a invocação de callbacks podem ser representadas pelas regras abaixo, que substituem **w-clear**.

$$\langle e, H, W, C\rangle \rightarrow \langle e, H[l_i \mapsto nil], W \setminus \{l_i\}, C\rangle \quad \text{se } (l_i \in W) \wedge (l_i \notin dom(C)) \wedge (\exists l_j \in LO(H(l_i)) \mid dead_w(l_j, e, H, W)) \qquad (\textbf{w-clear1})$$

$$\langle e, H, W, C\rangle \rightarrow \langle C(l_i)l_i; e, H[l_i \mapsto nil], W \setminus \{l_i\}, C[l_i \mapsto \bot]\rangle \quad \text{se } (l_i \in W) \wedge (l_i \in dom(C)) \wedge (\exists l_j \in LO(H(l_i)) \mid dead_w(l_j, e, H, W)) \qquad (\textbf{w-clear2})$$

O callback recebe como parâmetro a referência fraca à qual foi associado. Como no momento da invocação do callback o local que efetivamente disparou

[10] As demais regras apresentadas até aqui devem ser modificadas para incluir este novo contexto.

PUC-Rio - Certificação Digital Nº 0310845/CA

a limpeza da referência fraca ainda não foi coletado, poderíamos ter optado por passá-lo como parâmetro. Isso entretanto iria atrasar a sua coleta, tal como acontece com objetos finalizáveis.

Por fim, precisamos definir como tratar a coleta de referências fracas com callbacks ativos. Se considerarmos a notificação por callback como um mecanismo de finalização, é interessante que o callback não desapareça se a referência fraca tornar-se inacessível. Optamos assim por impedir a coleta de referências fracas com callbacks ativos. Para isso redefinimos a coleta de lixo como

$$\langle e, H, W, C\rangle \rightarrow \langle e, H[l_i \mapsto \bot], W \setminus \{l_i\}, C\rangle$$
$$\text{se } dead(l_i, e, H) \wedge (l_i \notin dom(C))$$
$$\wedge(\nexists l_j \in dom(C) \mid stReachable(l_i, C(l_j)l_j, H, W)) \quad (\mathbf{gc6})$$

Note que da mesma forma que fizemos com finalizadores, foi necessário adicionar uma condição específica para impedir a coleta de objetos acessíveis através de callbacks registrados, evitando o surgimento de ponteiros quebrados.

PUC-Rio - Certificação Digital Nº 0310845/CA

5.4.2 Tabelas Fracas

Antes de considerar a semântica de tabelas fracas precisamos determinar uma forma de descrever a semântica de arrays associativos (tabelas). Gostaríamos de implementar tabelas usando apenas os mecanismos básicos do modelo desenvolvido até aqui, sem precisar estender λ_{weak} com fizemos na última seção. Uma possível alternativa consiste na representação de tabelas ($t_i \in \mathbf{V}$) usando abstrações e expressões condicionais, conforme definido pelas operações:

$$\begin{aligned} table &= \lambda x_i.nil \\ get\ t_i\ k_i &= t_i k_i \\ set\ t_i\ k_i\ v &= \lambda x_i.((x_i = k_i)?\ v : t_i x_i) \end{aligned}$$

onde k_i é um valor qualquer que representa uma chave de busca ($k_i \in \mathbf{V}$); *table* é um construtor que cria uma tabela vazia; *get* recupera o valor associado a uma chave específica; e *set* cria uma nova tabela estendendo a tabela original com um novo par chave/valor.

Infelizmente essa implementação apresenta um problema sério: a abstração usada para representar tabelas preserva todos os valores que são associados a

uma chave durante a execução de um programa. Ainda que a semântica da tabela esteja correta, ela faz com que os locais que ocorrem em valores sobrescritos continuem sendo considerados acessíveis apesar de efetivamente não serem, impedindo portanto a sua coleta.

Uma solução um pouco mais rebuscada, e que evita esse problema, representa tabelas como listas de pares chave/valor. Um par é representado por uma abstração contendo apenas uma expressão condicional, onde cada elemento do par ocupa uma cláusula diferente. Uma lista é representada por um par cujo primeiro elemento contém o primeiro elemento da lista. O segundo elemento do par representa o resto da lista (o tail).

A partir dessa representação podemos definir as operações *get* e *set*. A operação *get* checa se a tabela não é vazia, e se não for, compara a chave do seu primeiro elemento com a chave de busca. Se as chaves forem iguais, o valor associado ao primeiro elemento é retornado; caso contrário *get* é invocado recursivamente recebendo como argumentos a chave de busca, e a tabela original sem o primeiro elemento (o resto da lista). Se a tabela passada como argumento for vazia (*nil*), *get* retorna *nil*. No Apêndice A apresentamos a definição completa desta e de todas as demais operações discutidas nesta seção.

A definição de *set* é um pouco mais complicada, já que envolve a construção de uma nova tabela. Assim como *get*, *set* percorre a lista que representa a tabela (usando recursão), procurando por uma chave igual a chave de busca. Se a chave não for encontrada, a lista que representa a tabela original é retornada, mas com o par recebido como argumento concatenado no final da lista. Se a chave for encontrada, o valor associado é substituído pelo valor recebido como argumento, e a tabela resultante é retornada.

Para implementar tabelas fracas com chaves fracas e valores fortes basta encapsular as chaves em referências fracas antes de compor o par chave/valor. Adicionalmente, para encontrar um elemento a partir de uma chave de busca é necessário dereferenciar o primeiro elemento de cada par antes de efetuar a comparação. Podemos limpar as tabelas implicitamente, como acontece por exemplo em Java, modificando a operação de inserção para checar e descartar os elementos com chaves iguais a *nil* antes de efetuar qualquer inserção.

PUC-Rio - Certificação Digital Nº 0310845/CA

5.4.3 Ephemerons

A implementação de tabelas fracas que descrevemos na seção anterior apresenta o mesmo problema de circularidade de referências discutido na Seção 4.3.5. Se um elemento de uma tabela fraca usa como chave um local referenciado pelo valor de um outro elemento, e vice-versa, estes elementos não serão limpos, e os objetos referenciados não serão coletados ao menos enquanto a tabela não for coletada. Para contornar esta restrição podemos implementar tabelas fracas usando ephemerons.

Um ephemeron pode ser definido como um par chave/valor no qual a conectividade do valor depende da conectividade da chave: se a chave é acessível, então o valor também é. Repare que essa relação entre a chave e o valor é similar àquela existente entre uma referência e o objeto referenciado, mas com uma importante distinção. Um local pode ser usado como chave em inúmeros pares, o que permite que ele esteja associado simultaneamente a múltiplos valores.

Como não é possível replicar a relação de conectividade inerente a referências usando apenas a semântica básica de λ_{weak}, precisamos estender esta linguagem para conseguir descrever ephemerons corretamente. Iremos representar um ephemeron ($p_i \in \mathbf{V}$) como uma par chave/valor no qual chaves não nulas são representadas por locais ($k_i \in Loc \cup \{nil\}$), e o valor é representado por um valor qualquer ($v \in \mathbf{V}$).

$$p_i = (k_i, v)$$

Para criar um ephemeron introduzimos o construtor ϕ, que recebe como parâmetros uma chave e um valor. A chave deve ser diferente de *nil* (ephemerons podem ter chaves nulas, mas apenas como consequência de uma limpeza). Esta semântica é descrita pela regra

$$\langle \phi\ l_i\ v, H, W \rangle \rightarrow \langle (l_i, v), H, W \rangle \qquad (\mathbf{ephem})$$

Chaves e valores são acessados através de operações de projeção, indicadas pelo operador π_i.

$$\langle \pi_1(k_i, v), H, W \rangle \rightarrow \langle k_i, H, W \rangle \qquad (\mathbf{proj1})$$

PUC-Rio - Certificação Digital Nº 0310845/CA

$$\langle \pi_2(k_i, v), H, W\rangle \rightarrow \langle v, H, W\rangle \qquad (\textbf{proj2})$$

Conforme discutimos na Seção 4.3.5, o valor armazenado por um ephemeron só precisa ser preservado enquanto a chave for acessível a partir do conjunto-raiz. Podemos limpar um ephemeron assim que a respectiva chave tornar-se fracamente acessível[11].

$$\langle e, H[l_i \mapsto (l_j, v)], W\rangle \rightarrow \langle e, H[l_i \mapsto (nil, nil)], W\rangle \quad \text{se } dead_w(l_j, e, H) \qquad (\textbf{w-clear3})$$

De forma análoga ao que acontece com referências fracas ordinárias (ver **w-clear**), os objetos referenciados por um ephemeron só serão coletados após a limpeza do mesmo. Consequentemente, contanto que ephemerons sejam rastreados de maneira correta, não é preciso modificar a regra que descreve a coleta de lixo (**gc4**).

Para evitar que a chave e o valor sejam coletados antes do respectivo ephemeron ser limpo, o conjunto de locais que ocorrem em um ephemeron é definido simplesmente como o conjunto de locais que ocorrem no valor, mais o local usado como chave, se esta for não nula.

$$LO((k_i, e)) = LO(k_i) \cup LO(e)$$

Para rastrear ephemerons iremos redefinir $dead_w$ como

$$dead_w(l_i, e, H, W) = (l_i \in dom(H)) \wedge \neg stReachable_\phi(l_i, e, H, W)$$

onde a função $stReachable_\phi$ indica se um local é acessível considerando a semântica de ephemerons. Para definir esta função, uma tarefa que não é das mais simples, precisamos utilizar mais algumas construções auxiliares.

$LO^-(e)$ irá denotar o conjunto de locais que ocorrem literalmente na expressão e, mas ignorando as ocorrências de locais em ephemerons. Usando esta abstração iremos redefinir *stReachable* como uma função que efetua o rastreamento de ob-

[11]Por simplicidade, optamos por não limpar ephemerons que ocorrem na expressão do programa.

PUC-Rio - Certificação Digital Nº 0310845/CA

jetos fortemente acessíveis desconsiderando ephemerons.

$$\begin{aligned} stReachable(l_i, e, H, W) = (l_i \in LO^-(e)) \\ \vee (\exists l_j \in LO^-(e) \mid (l_j \notin W) \\ \wedge stReachable(l_i, H(l_j), H, W)) \end{aligned}$$

Precisamos encontrar também uma forma de indicar os ephemerons que aparecem em uma expressão. Iremos representar por $EO(e)$ o conjunto de todos os ephemerons que aparecem literalmente na expressão e. Se um ephemeron que aparece na expressão e é composto por um outro ephemeron qualquer, ambos fazem parte de $EO(e)$. O conjunto de ephemerons que ocorre em um programa $\langle e, H, W\rangle$ é definido simplesmente como:

$$EO(\langle e, H, W\rangle) = EO(e) \cup \bigcup_{e_i \in range(H)} EO(e_i)$$

A partir destas construções podemos definir $stReachable_\phi$ como o menor ponto fixo que satisfaz a seguinte relação:

$$\begin{aligned} stReachable_\phi(l_i, e_1, H, W) = \\ stReachable(l_i, e_1, H, W) \\ \vee(\exists (l_j, e_2) \in EO(\langle e_1, H, W\rangle) \mid stReachable_\phi(l_j, e_1, H, W) \\ \wedge stReachable(l_i, e_2, H, W)) \end{aligned}$$

Um local é fortemente conexo se ele pode ser acessado diretamente através de referências ordinárias, ou se ele pode ser acessado diretamente através de algum ephemeron cuja chave é fortemente conexa.

Note que essa definição não faz nenhuma referência a conectividade dos ephemerons, e em um primeiro momento a conectividade do valor depende apenas da conectividade da chave (como acontece em Glasgow Haskell). Porém, como ephemerons inacessíveis são eventualmente coletados, locais acessíveis apenas através de ephemerons inacessíveis também tornam-se coletáveis, com uma única exceção. Considere os locais l_i e l_j tal que $H(l_i) = (l_j, l_i)$ e l_j é acessível. Neste caso l_i não vai ser coletado enquanto l_j não tornar-se inacessível.

Essa restrição, diferentemente daquela associada a ciclos entre elementos de uma tabela, não constitui um problema sério na medida em que não representa um vazamento de memória. Mesmo assim podemos evitá-la tornando a conectividade do valor dependente da conectividade do ephemeron, como acontece na

PUC-Rio - Certificação Digital Nº 0310845/CA

definição original de Hayes [35]. Para isso é necessário empregar uma expressão mais complexa para definir $stReachable_\phi$. Ao invés de considerar diretamente a conectividade de ephemerons que estão na memória, consideramos a conectividade de referências que apontam para ephemerons. Além disso efetuamos também o rastreamento dos ephemerons que ocorrem na expressão do programa.

$stReachable_\phi$ é redefinido então como o menor ponto fixo que satisfaz a expressão:

$$\begin{aligned} stReachable_\phi(l_i, e_1, H, W) = {} & stReachable(l_i, e_1, H, W) \\ & \vee (\exists (l_j, e_2), l_k \mid (l_j, e_2) \in EO(H(l_k)) \\ & \quad \wedge stReachable_\phi(l_j, e_1, H, W) \\ & \quad \wedge stReachable_\phi(l_k, e_1, H, W) \\ & \quad \wedge stReachable(l_i, e_2, H, W)) \\ & \vee (\exists (l_j, e_2) \in EO(e_1) \mid stReachable_\phi(l_j, e_1, H, W) \\ & \quad \wedge stReachable(l_i, e_2, H, W)) \end{aligned}$$

Através de alguns exemplos podemos verificar que $stReachable_\phi$ consegue efetuar o rastreamento de valores referenciados por ephemerons conforme o esperado. Considere o grafo de conectividade apresentado na figura abaixo, onde ephemerons são representados por dois retângulos adjacentes contendo respectivamente a chave e o valor.

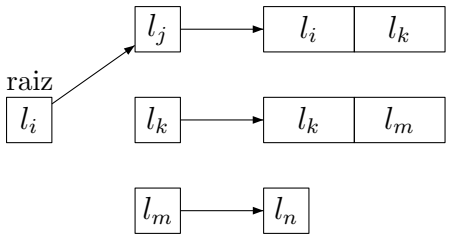

Figura 5.4: Rastreando ephemerons: exemplo 1.

Como l_j é diretamente acessível a partir da raiz, o ephemeron ao qual ele esta ligado ((l_i, l_k)) é rastreado. A chave desse ephemeron contém um local acessível, e portanto o valor correspondente, l_k, é considerado acessível. O ephemeron ao qual l_k esta ligado ((l_k, l_m)) é então rastreado, fazendo com que l_m, e o valor para o qual este aponta, l_n, também sejam considerados acessíveis.

Em um programa no qual os locais l_j e l_k ocorrem exclusivamente nas ligações $H(l_j) = (l_k, l_j)$ e $H(l_k) = (l_j, l_k)$, os ephemerons nunca são rastreados, e l_j e l_k são

PUC-Rio - Certificação Digital Nº 0310845/CA

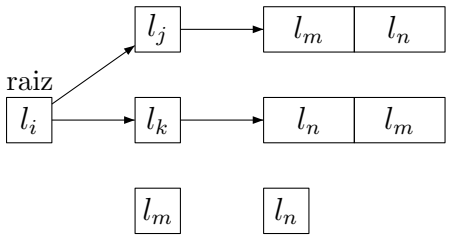

Figura 5.5: Rastreando ephemerons: exemplo 2.

coletados normalmente.

Em uma configuração como a apresentada na Figura 5.5, apesar da existência de um ciclo interno entre os ephemerons referenciados por l_j e l_k, as chaves destes ephemerons não são consideradas acessíveis. Os ephemerons portanto seriam eventualmente limpos, e l_m e l_n tornariam-se então coletáveis.

Assim como a definição original de $reachable$, a definição de $stReachable_\phi$ é um pouco abstrata, e não indica um método prático para efetuar o rastreamento de objetos em uma linguagem que oferece suporte a referências fracas e ephemerons. No próximo capítulo porém apresentamos um algoritmo que permite efetuar este rastreamento.

No Apêndice A descrevemos detalhadamente uma implementação de tabelas fracas usando ephemerons. Além de evitar vazamentos de memória, essa implementação é mais simples do que a versão que emprega apenas referências fracas ordinárias.

5.5 Referências Fracas e Finalização

Geralmente não faz muito sentido definir uma rotina de finalização que seja totalmente desacoplada do estado do objeto a ser finalizado. Daí a razão de finalizadores tradicionais receberem como parâmetro uma referência para o objeto finalizado. No caso de callbacks associados a referências fracas, em algumas linguagens o objeto finalizado também é passado como parâmetro para a rotina de finalização (por exemplo, Modula-3). No entanto, conforme já discutimos, essa semântica faz com que o processo de coleta de lixo seja postergado, eventualmente prejudicando o desempenho da aplicação.

É possível contudo manter o desacoplamento entre callbacks e objetos finalizados, e ainda assim garantir acesso a informações de estado durante a finalização. Para isso o coletor deve passar para o callback apenas a referência fraca, já limpa. O programa cliente armazena em uma tabela global (a *tabela de finalização*) as

PUC-Rio - Certificação Digital Nº 0310845/CA

informações necessárias a finalização de cada objeto, empregando como chave de busca as respectivas referências fracas. Ao ser invocado, o callback acessa a tabela usando o parâmetro recebido (a referência fraca), e recupera as informações desejadas.

Como as informações usadas na finalização de um objeto, dependendo da aplicação, podem mudar a qualquer tempo, não é suficiente guardar na tabela de finalização apenas cópias de valores. O objeto a ser finalizado deve encapsular as informações necessárias a sua finalização em um segundo objeto, mantendo como atributo uma referência para o mesmo. Uma segunda referência é armazenada então na tabela de finalização.

Apenas como exemplo, iremos implementar essa estratégia de finalização em λ_{weak}. Um objeto pode ser definido como um conjunto de campos e métodos, implementado através de uma estrutura de dados similar a uma `struct` de C ou um `record` de Pascal. Em λ_{weak} podemos implementar objetos, nesse sentido mais específico, através de listas. Por simplicidade iremos considerar apenas objetos compostos por dois campos, que serão representados por pares.

Suponha que f_i é uma função a ser usada para finalizar o objeto referenciado por l_i, e que ela espera como parâmetro um valor v armazenado no primeiro campo deste objeto. Se a tabela de finalização for armazenada pela referência l_t, podemos definir a seguinte operação para registrar um finalizador:

$$
\begin{aligned}
finalize_{obj}\ l_i\ f_i \quad = \quad & \textbf{let}\ x_i = \textbf{new}\ \textbf{in}\ (\\
& \quad x_i \overset{w}{:=} l_i; \\
& \quad l_t := set\ !l_t\ x_i\ (\pi_1 !l_i); \\
& \quad \textbf{notify}\ x_i\ (\lambda x_j.(f_i(get\ !l_t\ x_j))) \\
&)
\end{aligned}
$$

onde *set* e *get* são as operações para inserir e recuperar dados de uma tabela associativa (ver Seção 5.4.2), π_1 recupera o primeiro valor de um par, e a expressão **notify** $l_i\ v$ registra o callback v para a referência fraca l_i (ver Seção 5.4.1). $finalize_{obj}$ executa os seguintes passos:

(i) Atribui a referência a ser finalizada para uma referência fraca recém-criada.

(ii) Armazena o valor contido no primeiro campo do objeto na tabela de finalização. Como chave de busca é usada a referência fraca recém-criada (que será passada pelo coletor como parâmetro na invocação do respectivo

PUC-Rio - Certificação Digital Nº 0310845/CA

callback).

(iii) A rotina de finalização é encapsulada em uma nova função. O callback recupera o valor armazenado na tabela de finalização, e invoca a rotina de finalização original (f_i), passando o valor recuperado como parâmetro. Esta nova função é registrada como o callback da referência fraca recém-criada.

Após l_i tornar-se inacessível, a função f_i é automaticamente executada, recebendo como parâmetro apenas os valores armazenados por l_i realmente necessários para a sua finalização. Note que a execução efetiva do finalizador pode acontecer depois de l_i ter sido coletado.

Callbacks que recebem como argumento o objeto referenciado, ao invés da referência fraca, são equivalentes a finalizadores tradicionais. Para explorar formalmente essa relação vamos considerar novamente a semântica básica de finalizadores definida na Seção 5.3 através das seguintes regras:

$$\langle e, H, F\rangle \rightarrow \langle F(l_i)l_i;\ e, H, F[l_i \mapsto \bot]\rangle \quad \text{se } (l_i \in dom(F)) \wedge dead(l_i, e, H) \qquad (\textbf{fin-exec1})$$

$$\langle e, H, F\rangle \rightarrow \langle e, H[l_i \mapsto \bot], F\rangle \quad \text{se } dead(l_i, e, H) \wedge (l_i \notin dom(F)) \wedge (\nexists l_j \in dom(F) \mid reachable(l_i, F(l_j)l_j, H)) \qquad (\textbf{gc3})$$

Podemos definir um mecanismo de finalização semelhante a esse alterando a semântica de referências fracas para que callbacks recebam como parâmetro o objeto fracamente referenciado. Para isso é necessário modificar **w-clear2** da seguinte forma (iremos nos referir a esta versão modificada de λ_{weak} como λ_{weak_f}).

$$\langle e, H, W, C\rangle \rightarrow \langle C(l_i)H(l_i); e, H[l_i \mapsto nil], W \setminus \{l_i\}, C[l_i \mapsto \bot]\rangle \quad \text{se } (l_i \in W) \wedge (l_i \in dom(C)) \wedge (\exists l_j \in LO(H(l_i)) \mid dead_w(l_j, e, H, W)) \qquad (\textbf{w-clear4})$$

Nessa regra a execução do callback é representada pela expressão $C(l_i)H(l_i)$, onde $H(l_i)$ corresponde ao valor fracamente referenciado por l_i. Esse valor pode ser um local, ou um fechamento qualquer. Para evitar o surgimento de ponteiros quebrados esses locais não podem ser coletados ao menos enquanto eles forem fracamente acessíveis. Essa condição é garantida pela regra de coleta, redefinida

PUC-Rio - Certificação Digital Nº 0310845/CA

como:

$$\langle e, H, W, C\rangle \rightarrow \langle e, H[l_i \mapsto \bot], W \setminus \{l_i\}, C\rangle$$
$$\text{se } dead(l_i, e, H) \wedge (l_i \notin dom(C))$$
$$\wedge(\nexists l_j \in dom(C) \mid stReachable(l_i, C(l_j)H(l_j), H, W)$$
$$(\mathbf{gc7})$$

Para comparar a expressividade de um conjunto de regras podemos usar o conceito de simulação, semelhante ao apresentado por Milner [49]. Sejam R_1 e R_2 dois conjuntos de regras quaisquer. Dizemos que R_1 simula R_2, denotado por $R_1 \mathcal{S} R_2$, se para qualquer transição entre dois programas $\mathcal{P}_1$ e $\mathcal{P}_2$

$$\mathcal{P}_1 \xrightarrow{R_2} \mathcal{P}_2$$

existe uma ou mais transições em R_1 tal que

$$\mathcal{P}_1 \xRightarrow{R_1} \mathcal{P}_3$$

onde $\mathcal{P}_3 \equiv \mathcal{P}_2$. Se $R_1 \mathcal{S} R_2$, podemos dizer que R_1 é ao menos tão expressivo quanto R_2. Obviamente se $R_1 \mathcal{S} R_2$ e $R_2 \mathcal{S} R_1$ temos que $R_1 \simeq R_2$.

Não é possível comparar λ_{fin} e λ_{weak_f} diretamente usando simulação porque a estrutura de programas nestas duas linguagens é diferente. Para contornar essa dificuldade iremos empregar um artifício simples: definir uma transformação reversível que mapeia programas de λ_{fin} em programas de λ_{weak_f}[12]. Seja $\langle e_1, H_1, F\rangle$ um programa qualquer em λ_{fin}, $\mathbf{T}$ uma transformação entre programas $\mathbf{T} : \lambda_{fin} \rightarrow \lambda_{weak\_f}$, e $\langle e_2, H_2, W, C\rangle$ um programa em λ_{weak_f} tal que $\mathbf{T}(\langle e_1, H_1, F\rangle) = \langle e_2, H_2, W, C\rangle$. O programa $\langle e_2, H_2, W, C\rangle$ é construído da seguinte forma:

(i) A expressão do programa original é preservada ($e_2 = e_1$).

(ii) Todas as ligações existentes em H_1 são preservadas, ou seja, para qualquer l_i que pertence ao domínio de H_1, temos que $H_2(l_i) = H_1(l_i)$.

(iii) Para todo $l_i \in dom(F)$ é alocado um novo local l_j em H_2, e o contexto do programa é atualizado para refletir os efeitos da atribuição $l_j \overset{w}{:=} l_i$. Adicionalmente, a função registrada como finalizador de l_i é armazenada como o

[12]Se a transformação for reversível, podemos garantir que ela preserva a estrutura de programas.

PUC-Rio - Certificação Digital Nº 0310845/CA

callback de l_j, ou seja, $C(l_j) = F(l_i)$.

Considerando esta transformação, não é difícil perceber que para qualquer transição em λ_{fin}

$$\mathcal{P}_1 \xrightarrow{R_{fin}} \mathcal{P}_2$$

existe uma transição correspondente em λ_{weak_f}

$$\mathbf{T}(\mathcal{P}_1) \xrightarrow{R_{weak\_f}} \mathcal{P}_3$$

onde $\mathcal{P}_3 \equiv \mathbf{T}(\mathcal{P}_2)$. Os dois casos mais interessantes são:

- Transições **gc3** correspondem a transições **gc7**. Em ambos os casos locais são coletados se não forem acessíveis, não estiverem associados a um finalizador (callback), e não forem referenciados por nenhum outro finalizador.

- Transições **fin-exec1** correspondem a transições **w-clear4**. O finalizador é executado após o objeto tornar-se inacessível, no sentido forte. O objeto em questão é passado como parâmetro para o finalizador.

A partir desta análise podemos concluir que a linguagem λ_{weak_f} é ao menos tão expressiva quanto λ_{fin}, e os mecanismos de finalização considerados portanto são equivalentes.

PUC-Rio - Certificação Digital Nº 0310845/CA