Administração de Memória

- nossos interpretadores alocam memória dinâmicamente mas nunca a liberam
- em situações reais precisamos cuidar para que memória não mais utilizada seja liberada para novas alocações
- especialmente importante em linguagens com alocação implícita de memória "heap" como Lisp, Smalltalk, Prolog, Java, ML, etc.
- técnicas de alocação e liberação de memória constituem área de pesquisa chamada **administração de memória** ou **coleta de lixo**.

Alocação dinâmica de memória nos interpretadores

- 3 razões:
  - Ambientes são alocados quando avaliamos expresões lambda e quando aplicamos funções.
  - Árvores de expressões são alocadas após o *parsing*.
  - Durante a avaliação criamos *consC, lamC, etc.*
- 3 abordagens:
  - eliminar usos de ambientes através do uso de uma pilha
    - Funciona para chamada de funções, não para avaliação de lambdas.
  - eliminar a alocação dinâmica de células e ambientes criando um número fixo de registros deste tipo e tentando reutilizá-los (veremos 3 abordagens)
  - ignoraremos o problema contando com a lentidão na alocação (ou seja, esperamos que nunca atinjamos o limite de memória).

Coleta de lixo

- para podermos utilizar um número pré-fixado de registros para *células* e *ambientes* precisamos, quando todos os registros estão ocupados, procurar aqueles que são utilizados por valores que não são mais usados no programa
- processo de encontrar estas células é chamado de "coleta de lixo"
- coleta de lixo torna a implementação da linguagem mais complexa mas impede o programador de introduzir erros de "vazamento de memória".

Um detalhe importante

- Em nosso interpretador, todos os elementos utilizados são criados pelo Racket
- O próprio Racket implementa coleta de lixo, desta maneira entidades que não podem ser mais usadas eventualmente tem seu espaço reutilizado
- Porém, isso não elmina a questão de como, em outras implementações, em particular quando modelamos o *Storage*, podemos garantir o reuso do espaço de memória.
- Além disso, nas versões Storage, precisamos fazer a coleta de lixo lá.

Exemplo

- considere a sequência

->(set x ´(a b))

->(set x (cdr x))

➔ A célula guardamos o símbolo *‘a* não vai mais ser utilizada, na verdade, não há mais nenhuma referência a ela…

Utilizando uma pilha para funções

- eliminaremos todos os usos de ambientes resultado é menor uso de memória e avaliação mais eficiente
- ao avaliamos a lista de parâmetros, empilhamos os valores calculados
- *applyValueOp* desempilha seus argumentos da pilha
- *applyUserFun* vai buscar seus arumentos na pilha. Para isso *eval* recebe um novo parâmeto *AR* (de Activation Record) com o ínidice na pilha do primeiro argumento da função atual. AR é ajustado ao retornarmos de uma função.
- pilha é um vetor de *SEXP´*s, *stackTop* é a próxima posição disponível (ver código).
- "*rho:ENV*" é trocado por "*AR:integer*".

Utilizando uma pilha

- *evalList* deixa de ser função para ser procedimento e agora empilha valores
- chamada para *applyUserFun:*

```
newAR:= stacktop;
evalList(args);
eval := appplyUserFun(optr, newAR);
```

- applyValuOp passa a tirar seus argumentos da pilha:

```
s1:= topArg;
popArgos(1);
if arity(op) = 2
  then begin
            s2 := s1;
            s1 := topArg;
            popArgs(1)
       end; /*obs: não checamos mais num. args.*/
```

Utilizando uma pilha: applyUserFun

```
function applyUserFun (nm: NAME; newAR: integer): SEXP;
var f: FUNDEF;
begin
   f:= fetchFun(nm);
   if f= nil  then ...
   with f^ d begin
         applyUserFun := eval(body, newAR);
         popArgos(lenghNL(formals));
      end
   end;
```

Utilizando uma pilha: EXPREC

- Declaração de EXPREC

```
EXPREC = record
            case etype: EXPTYPE of
              ...
                VAREXP: (varble: NAME; offset: integer); (* 0
    para globais*)
              ...
            end;
```

- em *eval*

```
VAREXP:
    if offset > 0
       then eval := argStack[AR+offset-1]
       else ...
```

Manutenção da Heap

- as expressões simbólicas alocadas durante a execução de um programa podem ficar inativas.
- não se pode prever o padrão de execução de um programa para descobrir antecipadamente as SEXP´s inativas
- só se pode saber se um SEXP está ativo se ele está *acessível*:

  DEF. Uma SEXP está **acessível** se existe uma referência a ele em:
  - globalEnv
  - currentEXP (i.e. em uma VALEXP dentro de currentEXP)
  - corpo da definição de uma função em *fundefs*
  - *nilValue, trueValue*
  - *argStack*
  - qualquer outra SEXP acessível
- nem todos os acessíveis são ativos mas a recíproca é verdadeira (administração conservadora)

Manutenção da Heap

- informações não acessíveis não podem afetar a computação do programa, assim podem ser eliminadas e seus registros reciclados
- propriedades importantes
  - não reciclar registros acessíveis
  - reciclar todos os resitros inacessíveis (vazamento de memória)
- duas estratégias básicas:
  - contagem de referências
  - coleta de lixo

Contagem de referências

- cada célula (SEXPREC) carrega em sua representação um campo com o número de referências ela.
- uma célula pode ter mais de uma referência a ela:

  ->(set x ´(a b))

  ->(set y x)

  ->(set z (cdr y))
- ao se criar uma célula o contador é inicializado para zero. A cada atribuição onde uma variável ou campo de outra célula recebe a célula como valor, o contador é incrementado em 1.
- quando o contador atinge 0, a célula poder ser reciclada.

Contagem de referências: implementação

- Nova declaração de SEXPREC:

```
SEXPREC = record
        refcnt: integer;
        case sxptype: SEXPTYPE of
        ... igual...
    end;
```

- Manipulação de SEPXREC´s
  - quando SEXPREC é colocado na pilha, incrementamos o contador, quando é retirado da pilha, decrementamos o contador
  - em *assign* o valor atribuído tem o seu refcnt incrementado o valor anterior seu refcnt decrementado
  - na operação CONSOP de applyValueOp, os dois SEXPREC´s que estão sendo agrupados tem seu refcnt incrementado
  - etc.

Contagem de referências: final

- modificações são inúmeras, listamos apenas algumas principais
- alguns problemas a que devemos ficar atentos:
  - Em atribuições a célula do novo valor deve ser incrementada ANTES de decrementar a do valor antigo, senão atribuições triviais podem gerar reciclagem indevida de células:

    ->(begin (set y 20)(set y y))
  - em *applyValueOp* não podemos decrementar os contadores das células ao retirarmos da pilha pois podemos estar aplicando a operação "cons".
  - Os módulos *eval, appluValueOp, applyUserFun, applyCrtlOp* passam a ser procedimentos ao invés de funções:

    (define f (x y) y)

    (+ (f 2 (+ 3 4)) (+ 5 6))

    ==> ao calcularmos ao calcularmos (+ 3 4) criamos a célula com "7" e empilhamos como argumento. Ao retornarmos da função desempilhamos os argumentos, reciclando a célula, será utilizada para guardar o resultado de (+ 5 6). Assim, precisamos empilhar o resultado de função.

Mark-scan ou Mark-and-sweep

- a contagem de referências embora fácil de administrar não consegue eliminar células que fazem parte de um ciclo de referências (a aponta para b, b aponta para a).
- a solução é fazer periódicamente uma coleta das células inúteis, ou “coleta de lixo”
- forma mais antiga: “mark-scan” ou “mark-and-sweep”
- pontos principais:
  - células livres mantidas em lista ligada
  - quando atingimos um certo valor de ocupação iniciamos coleta
  - primeira fase: descobrimos as células ativas em algoritmo baseado na definição de células ativas, marcando cada uma delas como “viva”
  - segunda fase: varremos a memória retornando todas as células não marcadas para a lista livre.

Mark-scan: implementação

- adição do procedimento *gc*
- todas as chamadas a *mkSExp* são substituídas por chamadas a *allocSExp*, que tem como função alocar um registro da lista livre
- em *applyValueOp* toda a alocação de memória deve ocorrer antesde tirarmos os argumentos da pilha (para evitar que estes sejam coletados como lixo)
- novas declarações:

```
MEMSIZE = 500;
memory: array[1..MEMSIZE] of SEXP;
freeSEXP: SEXP;
```

- adição do procedimento *initMemory*

Mark-scan código - 1

Mark-scan código - 2

Mark-scan código - 3

Mark-scan código - 5

O método dos semi-espaços

- sacrifica uso de memória por rapidez
- usa apenas metade da memória de cada vez (semi-espaço)
- memória é alocada sequencialmente do espaço livre
- quando memória está sem espaço coleta de lixo é iniciada
- na verdade o que é coletado é o “não lixo”

Método dos semi-espaços: coleta

- repete-se algo como a fase de busca de “mark-scan”, quando se encontra um objeto atingível não visitado anteriormente ele é copiado para o outro semi-espaço
- algoritmo marca a versão antiga do objeto copiado e deixa indicado seu novo endereço no novo semi-espaço
- quando se encontra um objeto atingível já copiado, atualiza-se o endereço da referência para o endereço indicado na cópia antiga do objeto
- obs: em sistemas de memória virtual uso de espaço extra não é em geral problema

Semi-espaços: exemplo - 1

Semi-espaços: exemplo - 2

Semi-espaços: exemplo - 3

Semi-espaços: exemplo - 4

Semi-espaços: exemplo- 5

Semi-espaços: implementaçao

```
/*Declarações:*/
memory:array[1..2] of array[1..HALFMEMSIZE] of SEXP
nextloc: integer; /*entre 1 e HALFMEMSIZE + 1*/
halfinuse: 1..2;

procedure initMemory;
  var i:integer;
  begin
     for i:= 1 to HALFMEMSIZE do begin
          new(memory[1][i]);
           new(memory[2][i])
          end;
     nextloc:= 1;
     halfinuse := 1;
   end;
```

Semi-espaços: implementação - 2


```
function allocSExp (t: SEXPTYPE):SEXP;
  var s:SEXP;
  begin
      if nextloc > HALFMEMSIZE then begin
         switchMem;
         if nextloc > HALFMEMSIZE /*coleta não adiantou*/
           then begin
                 writeln("memory overflow!!!");
                  nextloc := 1;
                  goto99
                end;
         if currentExp = nil then begin
                  writeln ("called gc during parsing. Reenter input');
                  goto99;
                 end
       end;
       s := memory[halfinuse][nextloc];
       nextloc := nextloc + 1
       s^.sxptype := t;
      s^.moved := false;
      alloceSExp := s;

 end;
```

Semi-espaços: implementação - 3

```
procedure switchMem;
   var newHalf: 1..2;
        newnextloc: integer;
    begin
       write(´Switching memories...’);
        newhalf := 3 - halfinuse;
        initnewHalf;
        traverse;
        writeln(.....<dados sobre a coleta>);
        halfinuse:= newhalf;
        nextlock := newnextloc
     end;
```

Semi-espaços: implementação - 4

```
procedure initNewHalf;
  var fd: FUNDEF;
        vl: VALUELIST;
         i: integer;
   begin
       newnextloc := 1;
        nilValue := moveSExp(nilValue);
        trueValue := moveSExp(trueValue);
        vl := globalEnv^.values;
         while vl <> nil do begin
              vl^.head := moveSExp(vl^.head);
              vl := vl^.tail;
            end;
          for i:= 1 to statcktop-1 do
            argStack[i] := moveSExp(argStack[i]);
          if currentExp <> nil then moveExp(currentExp);
          fd := fundefs;
          while fd <> nil do begin
                moveExp(fd^.body);
                fd := fd^.nextfundef
              end
  end;
```

Semi-espaços: implementação - 5

```
procedure moveExp (e: EXP);
  var el: EXPLIST;
   begin
       with e^ do
            case etype of
                 VALEXP: sxp:= moveSExp(sxp);
                  VAREXP: ;
                   APEXP: begin
                                 el := args;
                                  while el <> nil do begin
                                     moveExp(el^.head);
                                     el := el^.tail
                                   end
                                end
              end; /*fim do case e do with*/
      end;
```

Semi-espaços: implementação - 6

```
procedure trasverse;
   var queueeptr: integer;
    begin
        queueptr := 1;
        while queueptr < newnextloc do begin
            with memory[newhalf][queueptr]^  do
               if sxptype = LISTSXP then begin
                  carval := moveSExp(carval);
                  cdrval := moveSExp(cdrval);
                end; /*if*/
              queueptr := queueptr + 1;
          end; /*while*/
      end;
```

Semi-espaços: implementação - 7


```
function moveSExp(s: SEXP): SEXP;
   var: target: SEXP;
    begin
      if s^.moved then moveSExp := s^.carval /*carval guarda novo end.*/
      else begin
                target := memory[newhalf][newnextloc];
                 newnextloc := newnextloc + 1;
                 target^.sxptype := s^.sxptype;
                  target^.moved := false;
                  case s^.sxptype of
                        NILSXP: ;
                        NUMSXP: taret^.intval := s^.intval;
                        SYMSXP: target^.symval := s^.symval;
                        LISTSXP:  begin
                                        target^.carval:= s^.carval;
                                        target^.cdrval:= s^.cdrval
                                    end
                   end; /*case*/
                   s^.carval := target; /*guarda novo endereço em carval*/
                   s^.moved := true;
                   moveSExp := target;
               end;
end;
```

Final

- métodos apresentados são os mais básicos
- existem variações como método das "gerações" ("generational garbage collection")
- coleta de lixo concorrente: sobrecarga distribuída ao longo do processamento
- híbridos: contagem de referência eficiente para grande parte dos casos, podemos combinar com outros métodos