Linguagens de Programação

Modelos de Custo

Andrei Rimsa Álvares
andrei@cefetmg.br

Sumário

- Introdução
- Modelo de custo para listas
- Modelo de custo para chamadas de funções
- Modelo de custo para pesquisas em Prolog
- Modelo de custo para arranjos
- Modelos de custo espúrios

INTRODUÇÃO

Linguagens de Programação

Introdução

- Considere uma lista `L` em Prolog com milhões de elementos; qual das seguintes abordagens você acha que é mais rápida:
  - Adicionar um novo elemento a cabeça de uma lista?

```
?- length(L, 5000000), time(LL = [a|L]).
% 0 inferences, 0.000 CPU in 0.000 seconds (61% CPU, 0 Lips)
L = [_A, _B, _C, _D, _E, _F, _G, _H, _I|...],
LL = [a, _A, _B, _C, _D, _E, _F, _G, _H|...].
```

Esta primeira abordagem é mais rápida

  - Ou adicionar um novo elemento ao final de uma lista?

```
?- length(L, 5000000), time(append(L, [a], LL)).
% 5,000,000 inferences, 0.346 CPU in 0.369 seconds (94% CPU, 14443478 Lips)
L = [_A, _B, _C, _D, _E, _F, _G, _H, _I|...],
LL = [_A, _B, _C, _D, _E, _F, _G, _H, _I|...].
```

Introdução

- Este tipo de informação é raramente explícito na especificação da linguagem, mas é uma parte importante da expertise de um programador competente

- Todo programador experiente possui um modelo de custo de uma linguagem; um modelo mental dos custos relativos de várias operações

Cuidado: modelar precisamente custos absolutos envolveria conhecer o comportamento da linguagem, do sistema operacional, do processador, da memória...

Aqui serão considerados apenas custos relativos

MODELO DE CUSTO PARA LISTAS

Linguagens de Programação

Modelo de Custo para Listas

- Prolog possui duas formas de especificar listas
  - Uma forma mais conveniente usando colchetes

```
?- L = [1,2,3,4].
L = [1, 2, 3, 4].
```

  - E outra mais detalhada usando o predicado ponto

```
?- L = .(1,.(2,.(3,.(4,[])))).
L = [1, 2, 3, 4].
```

Cuidado: para que o predicado ponto funcione no interpretador swipl, inicialize-o em modo tradicional:

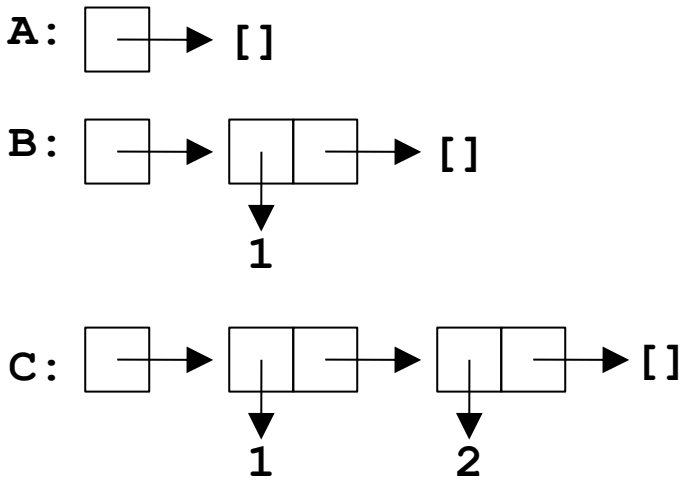
```
$ swipl --traditional
```

Como listas são implementadas internamente em Prolog?

A lista *Cons-Cell*

- Uma lista é implementada como uma cadeia de ponteiros
  - Este tipo de lista é utilizado em várias linguagens, como ML, Prolog, LISP, entre outras

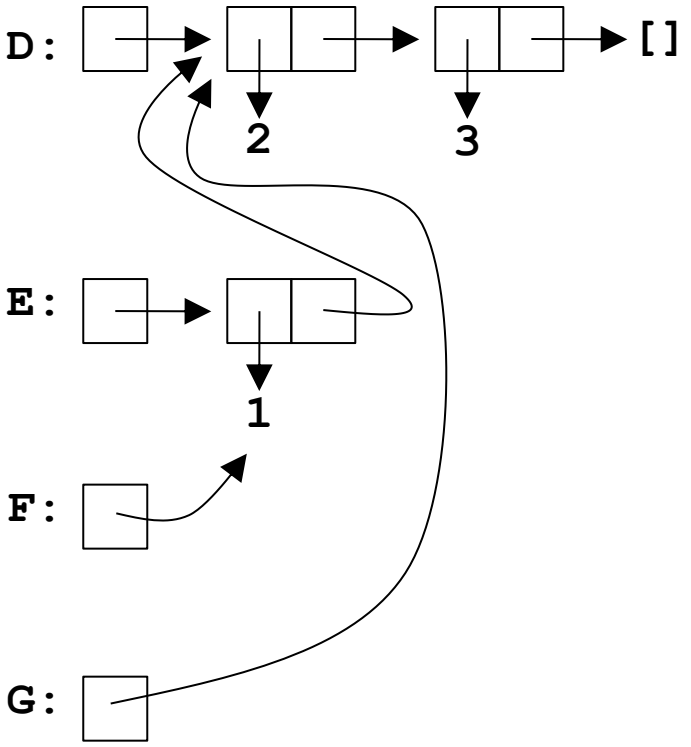
```
?- A = [],
|    B = .(1,[]),
|    C = .(1,.(2,[])).
A = [],
B = [1],
C = [1, 2].
```



A lista *Cons-Cell*

- Uma importante consequência desta representação é que listas podem compartilhar estrutura umas com as outras

```
?- D = [2,3],
|    E = [1|D],
|    E = [F|G].
D = [2, 3],
E = [1, 2, 3],
F = 1,
G = [2, 3].
```



Como saber que Prolog compartilha estrutura em listas? Ou seja, que `E = [1|D]` não gera uma cópia de `D`?

Length

- Tanto o predicado `length` de Prolog quanto a função `length` de ML devem percorrer a lista para contar a quantidade de elementos
  - Não tem atalho neste caso, o tempo gasto é proporcional ao tamanho da lista

```
length([], 0).
length([_|L], Y) :- length(L, Y1), Y is Y1 + 1.
```

Em Prolog

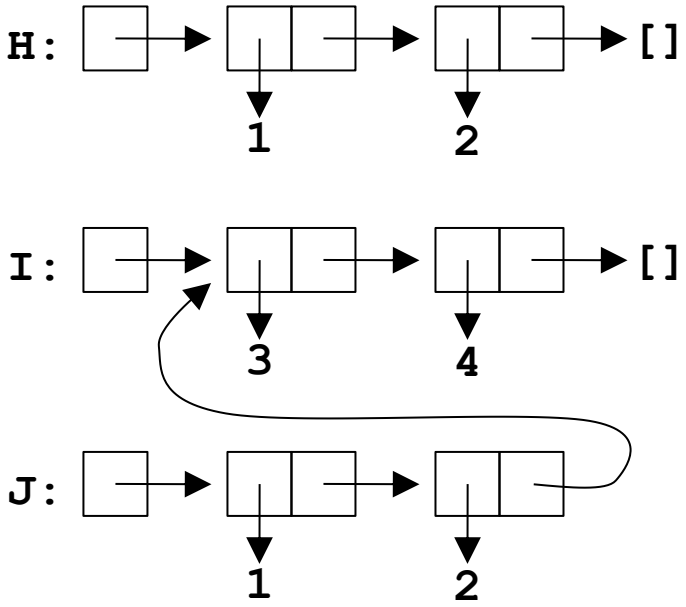
```
fun length nil = 0
|   length (_::tail) = 1 + length tail;
```

Em ML

Append

- O predicado `append` de Prolog também pode ser custoso, já que é preciso fazer uma cópia da primeira lista

```
?- H = [1,2],
|    I = [3,4],
|    append(H, I, J).
H = [1, 2],
I = [3, 4],
J = [1, 2, 3, 4].
```



Append

- O predicado `append` precisa copiar o prefixo
  - O tempo gasto é proporcional ao tamanho da primeira lista

```
append([], L, L).
append([X|L1], L2, [X|L3]) :- append(L1, L2, L3).
```

Em Prolog

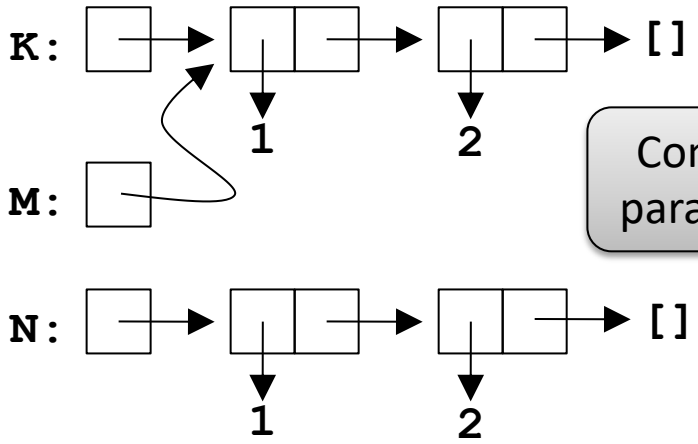
```
fun append nil l = l
|   append (head::tail) l = head::append tail l;
```

Em ML

Unificação de Listas em Prolog

- Considere a declaração de três listas iguais `K`, `M`, `N`

```
?- K = [1,2],
|    M = K,
|    N = [1,2].
K = [1, 2],
M = [1, 2],
N = [1, 2].
```



Como funciona a unificação para `K = M`? E para `K = N`?

- Para verificar se as listas unificam é preciso comparar elemento por elemento

```
&=([], []).
&=([H|L1], [H|L2]) :- &=(L1, L2).
```

Prolog pode usar atalhos se encontrar uma estrutura compartilhada, mas no pior caso é preciso comparar a estrutura completa de ambas as listas

Cuidado: a unificação é ainda menos eficiente que `length` ou `append`, já que também precisa unificar todos os seus elementos

Modelo de Custo de Listas

- Colocar um elemento no começo de uma lista (*consing*) gasta tempo constante
- Extrair a cabeça ou a cauda de uma lista gasta tempo constante
- Computar o resultado do tamanho de uma lista (como uma função) gasta tempo proporcional ao tamanho da lista
- Computar o resultado da concatenação de duas listas (como uma função) gasta tempo proporcional ao tamanho da primeira lista
- Unificar duas listas gasta, no pior caso, tempo proporcional ao seu tamanho

Aplicação

- Considere a função `reverse` para reverter os elementos de uma lista

```
reverse([], []).
reverse([Head|Tail], Rev) :-
  reverse(Tail, TailRev),
  append(TailRev, [Head], Rev).
```

Qual o problema desta implementação? É possível criar uma variação mais eficiente?

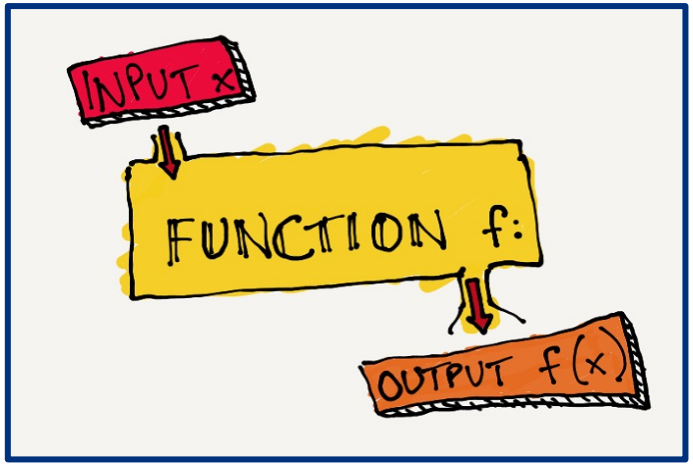
- Concatenar ao final da lista é altamente ineficiente; o modelo de custos guia programadores a adicionar elementos à cabeça da lista

```
reverse(X, Y) :- rev(X, [], Y).

rev([], Sofar, Sofar).
rev([Head|Tail], Sofar, Rev) :-
  rev(Tail, [Head|Sofar], Rev).
```

Esta implementação é muito mais eficiente: gasta tempo linear ao invés de quadrático

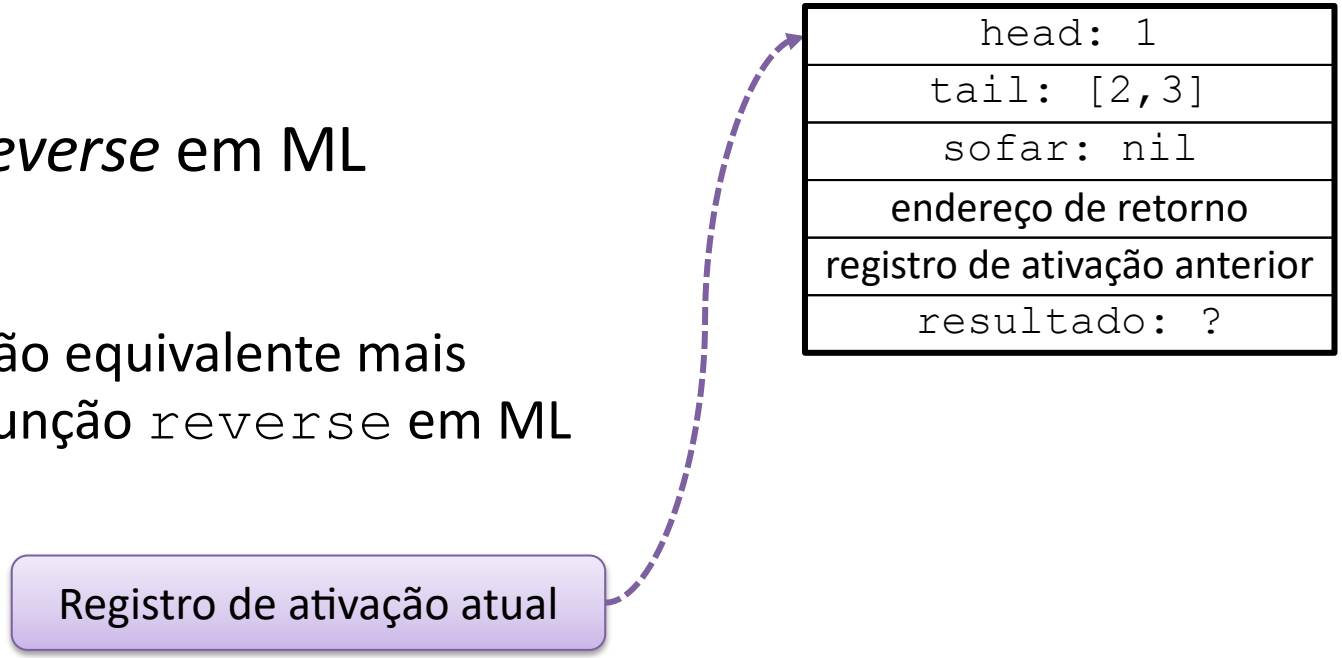
MODELO DE CUSTO PARA CHAMADAS DE FUNÇÕES



Linguagens de Programação

Reverse em ML

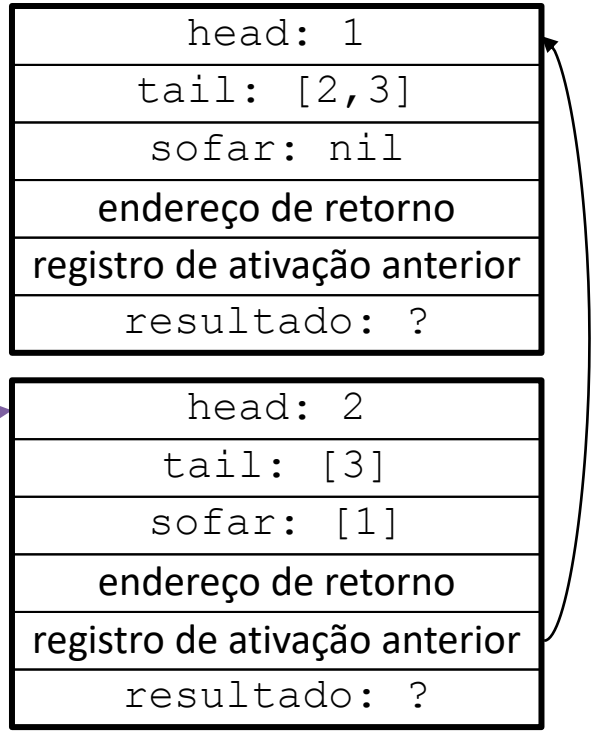
- Implementação equivalente mais eficiente da função `reverse` em ML




```
fun reverse x =
  let
    fun rev (nil, sofar) = sofar
    |   rev (head::tail, sofar) =
      rev (tail, head::sofar);
  in
    rev(x, nil)
  end;
```

Reverse em ML

- Implementação equivalente mais eficiente da função `reverse` em ML



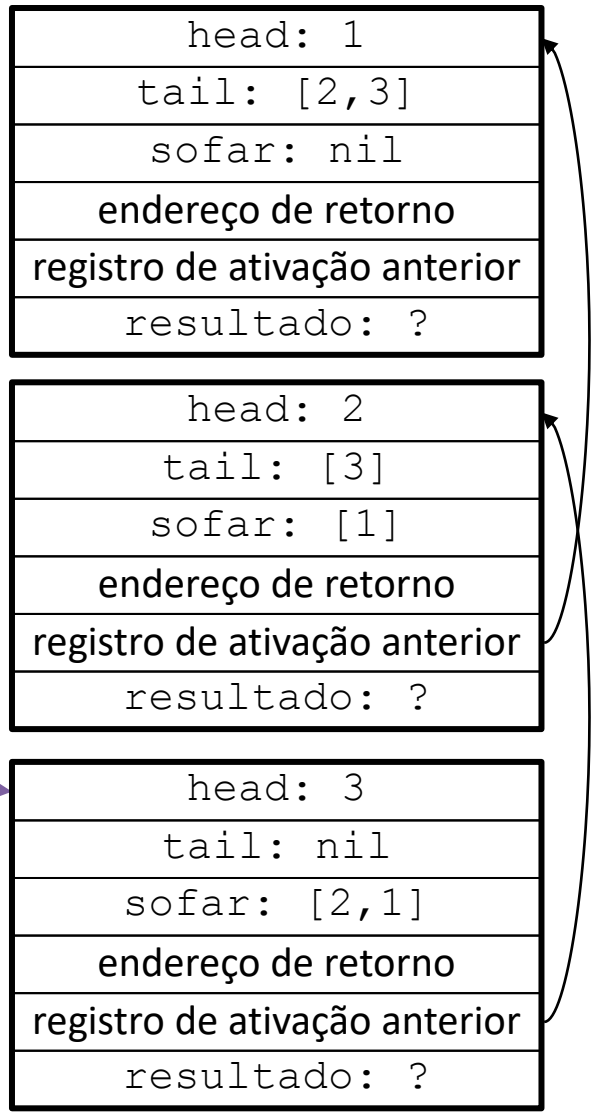

```
fun reverse x =
  let
    fun rev (nil, sofar) = sofar
    |   rev (head::tail, sofar) =
      rev (tail, head::sofar);
  in
    rev(x, nil)
  end;
```

Reverse em ML

- Implementação equivalente mais eficiente da função `reverse` em ML

Registro de ativação atual

```
fun reverse x =
  let
    fun rev (nil, sofar) = sofar
    |   rev (head::tail, sofar) =
      rev (tail, head::sofar);
  in
    rev(x, nil)
  end;
```

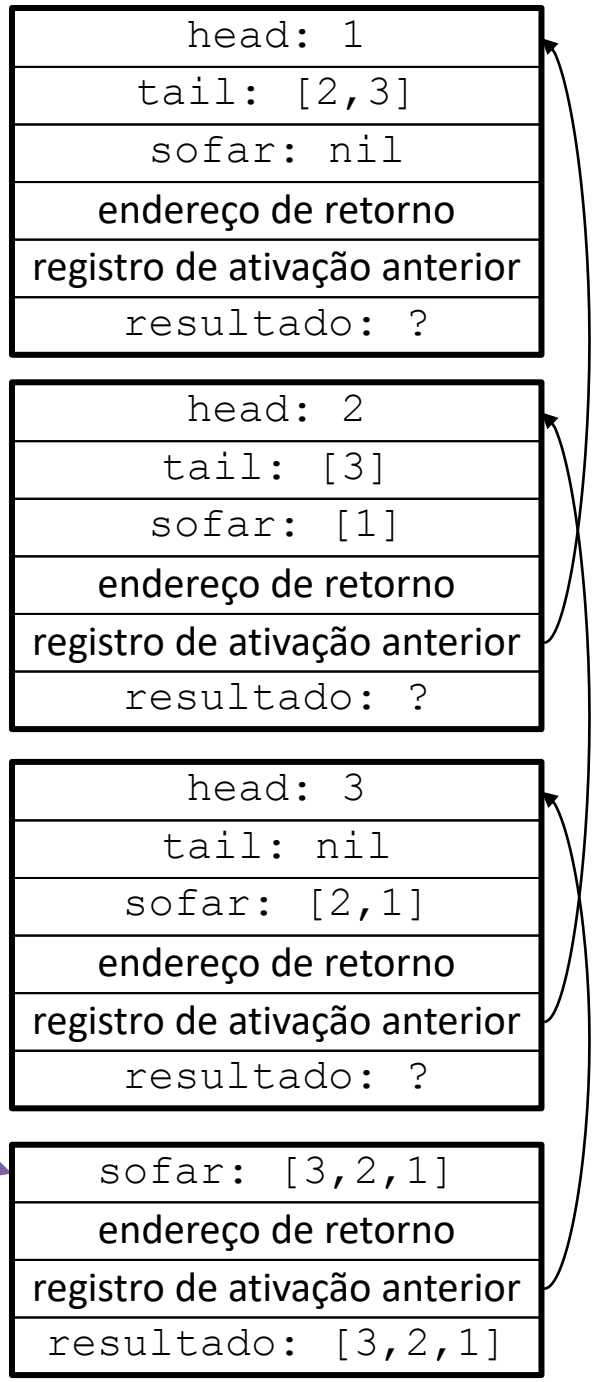


Reverse em ML

- Implementação equivalente mais eficiente da função `reverse` em ML

Registro de ativação atual

```
fun reverse x =
  let
    fun rev (nil, sofar) = sofar
    |   rev (head::tail, sofar) =
      rev (tail, head::sofar);
  in
    rev(x, nil)
  end;
```

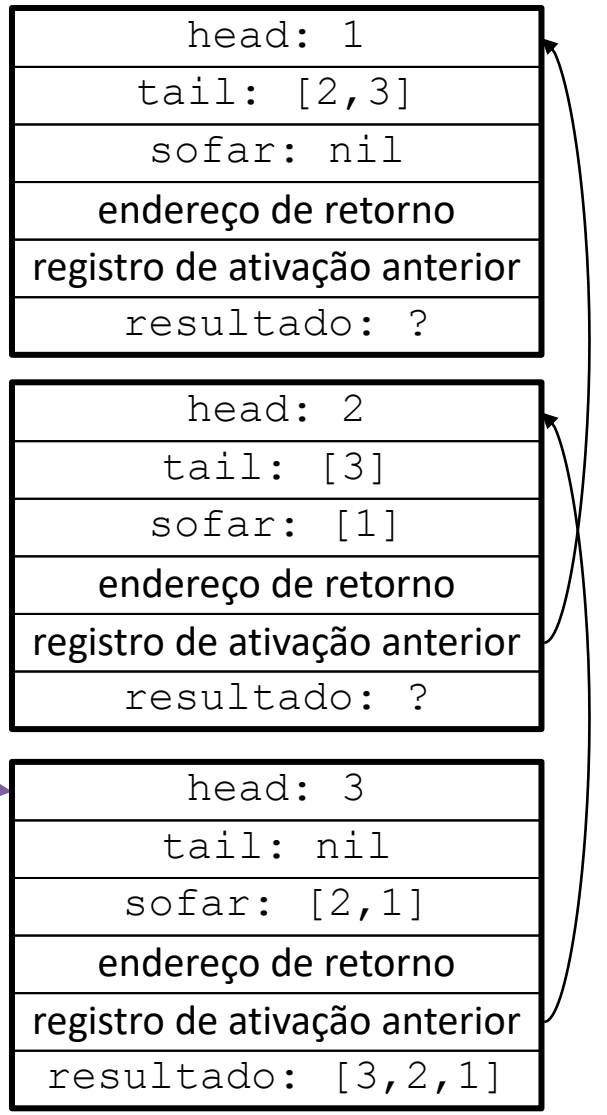


Reverse em ML

- Implementação equivalente mais eficiente da função `reverse` em ML

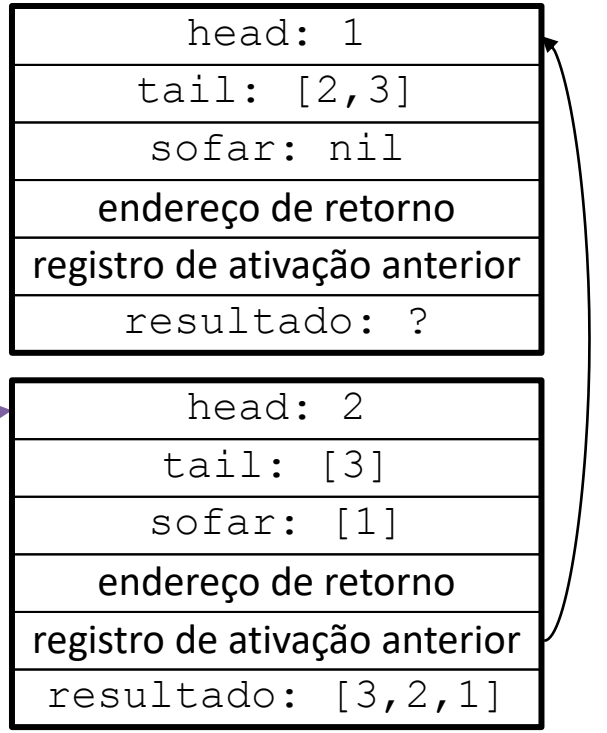
Registro de ativação atual

```
fun reverse x =
  let
    fun rev (nil, sofar) = sofar
    |   rev (head::tail, sofar) =
      rev (tail, head::sofar);
  in
    rev(x, nil)
  end;
```



Reverse em ML

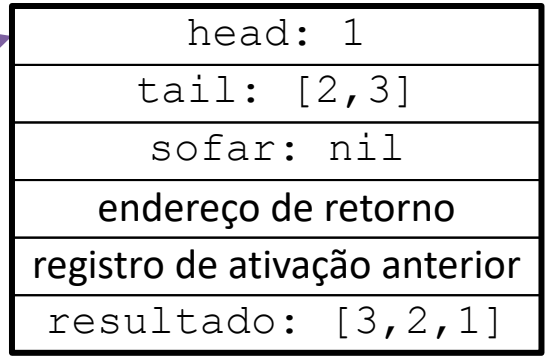
- Implementação equivalente mais eficiente da função `reverse` em ML




```
fun reverse x =
  let
    fun rev (nil, sofar) = sofar
    |   rev (head::tail, sofar) =
      rev (tail, head::sofar);
  in
    rev(x, nil)
  end;
```

Reverse em ML

- Implementação equivalente mais eficiente da função `reverse` em ML



Registro de ativação atual

```
fun reverse x =
  let
    fun rev (nil, sofar) = sofar
    |   rev (head::tail, sofar) =
      rev (tail, head::sofar);
  in
    rev(x, nil)
  end;
```

Chamada em Cauda (*Tail Call*)

- No exemplo anterior, após a ativação (recursiva) de cada chamada, nenhuma outra computação foi feita
  - A função apenas retornou o valor para seu chamador

- Este tipo de chamada de função tem um nome especial: **chamada em cauda** (*tail call*)

Todas as chamadas do exemplo anterior foram chamadas em cauda

Recursão em Cauda (*Tail Recursive*)

- Uma função recursiva tem recursão de cauda se todas as suas chamadas recursivas forem chamadas de cauda
  - A função `rev` é recursiva de cauda, como pode ser visto pela sua segunda alternativa

```
fun reverse x =
  let
    fun rev (nil, sofar) = sofar
    |   rev (head::tail, sofar) =
      rev (tail, head::sofar);
  in
    rev(x, nil)
  end;
```

Qual a vantagem de se ter recursão de cauda?

Otimização de Chamada em Cauda (*Tail-Call Optimization*)

- Quando uma função faz uma chamada em cauda, ela não precisa mais de seu registro de ativação

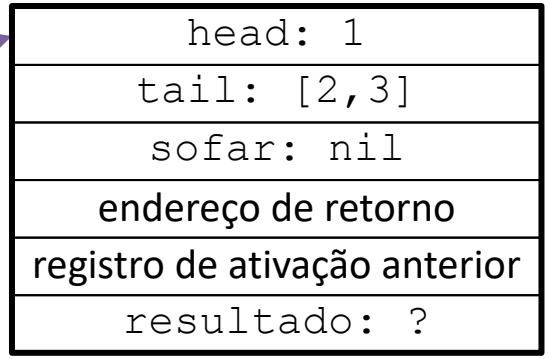
- A maioria dos sistemas de linguagens tiram vantagem disto para otimizar chamadas em cauda ao usar o mesmo registro de ativação para a função chamada
  - Não é preciso empilhar/desempilhar um novo registro de ativação
  - Funções chamadas retornam diretamente ao chamador original

Como funciona isto?

Exemplo

- Exemplo da otimização de chamada em cauda ao aproveitar o mesmo registro de ativação

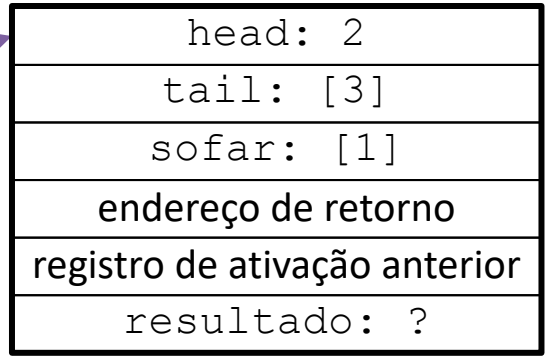
Registro de ativação atual




```
fun reverse x =
  let
    fun rev (nil, sofar) = sofar
    |   rev (head::tail, sofar) =
      rev (tail, head::sofar);
  in
    rev(x, nil)
  end;
```

Exemplo

- Exemplo da otimização de chamada em cauda ao aproveitar o mesmo registro de ativação

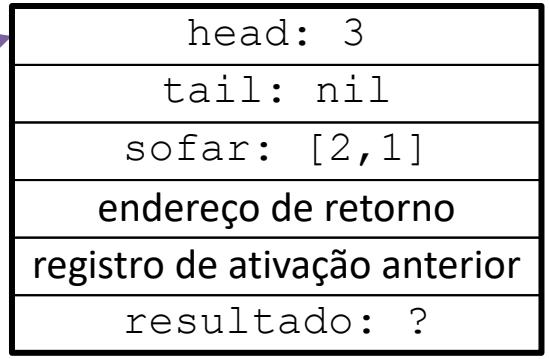



```
fun reverse x =
  let
    fun rev (nil, sofar) = sofar
    |   rev (head::tail, sofar) =
      rev (tail, head::sofar);
  in
    rev(x, nil)
  end;
```

Exemplo

- Exemplo da otimização de chamada em cauda ao aproveitar o mesmo registro de ativação

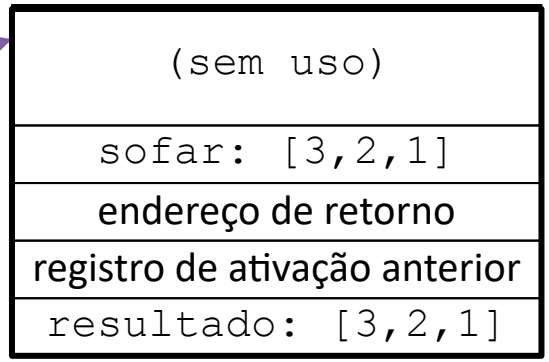
Registro de ativação atual




```
fun reverse x =
  let
    fun rev (nil, sofar) = sofar
    |   rev (head::tail, sofar) =
      rev (tail, head::sofar);
  in
    rev(x, nil)
  end;
```

Exemplo

- Exemplo da otimização de chamada em cauda ao aproveitar o mesmo registro de ativação




```
fun reverse x =
  let
    fun rev (nil, sofar) = sofar
    |   rev (head::tail, sofar) =
      rev (tail, head::sofar);
  in
    rev(x, nil)
  end;
```

Modelo de Custo de Otimização de Chamadas em Cauda

- Neste modelo, chamadas em cauda são significativamente mais rápida que chamadas que não são em cauda

- Este tipo de chamada consome muito menos espaço (já que aproveita o mesmo registro de ativação)
  - Funções recursivas em cauda gastam espaço constante
  - Funções não-recursivas em cauda gastam espaço, no mínimo, linear na profundidade da recursão

Aplicação

- Considere a função `length` para calcular o tamanho de uma lista

```
fun length nil = 0
|   length (_::tail) =
        1 + length tail;
```

Essa função é recursiva em cauda? Como criar uma variação mais eficiente desta função?

- O modelo de custos guia programadores a criar funções recursiva em cauda

Dica: use um parâmetro como acumulador ao fazer a conversão

```
fun length list =
  let
    fun len (nil, sofar) = sofar
    |   len (_::tail, sofar) = len (tail, sofar+1);
  in
    len (list, 0)
  end;
```

Embora mais extensa, esta solução executa mais rápido e toma menos espaço

Aplicabilidade

- Implementado em virtualmente todos os sistemas de linguagens funcionais (algumas garantidas explicitamente pela sua especificação)

- Também implementado por bons compiladores para a maioria das linguagens modernas, como C, C++ e outros

Cuidado: atualmente não implementado pelo sistema de linguagens de Java

- Uma otimização similar é feita pela maioria dos sistemas compilados de Prolog
  - Porém, pode ser complicado identificar chamadas recursivas em cauda

```
p :- q(X), r(X).
```

A chamada do predicado `r` não é (necessariamente) uma chamada em cauda, por causa da possibilidade de *backtracking*

MODELO DE CUSTO PARA PESQUISAS EM PROLOG

Linguagens de Programação

Pesquisa em Prolog

- O sistema de busca de Prolog resolve as cláusulas de Horn da esquerda para a direita

- O sistema tenta casar regras do banco de dados na ordem em que aparecem tentando unificar a cabeça de cada regra com o termo alvo atual

- Em caso de falha é feito *backtracking*
  - Podem haver mais de uma regra que pode ser unificada a cada termo alvo, o sistema tenta quantas forem necessárias

Aplicação

- Considere o predicado `grandfather` em Prolog

```
grandfather(X,Y) :-
  parent(X,Z),
  parent(Z,Y),
  male(X).
```

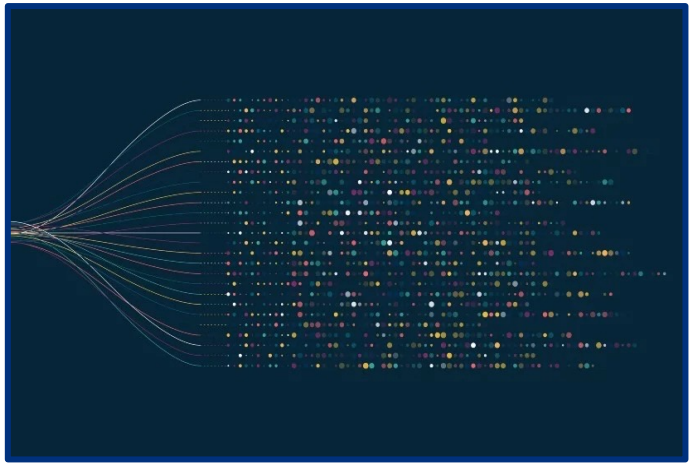
Qual o problema deste predicado? Como criar uma variação mais eficiente dele?

- O modelo de custos guia programadores a usar predicados que restringem o espaço de soluções mais cedo

```
grandfather(X,Y) :-
  male(X),
  parent(X,Z),
  parent(Z,Y).
```

Embora logicamente idêntica ao predicado anterior, esta solução pode ser muito mais rápida

MODELO DE CUSTO PARA ARRANJOS



Linguagens de Programação

Arranjos Multidimensionais

- Muitas linguagens de programação suportam arranjos multidimensionais

```
#define M 20000
#define N 10000
static char matrix[M][N];
```

```
int addup1() {
  int i, j, sum = 0;
  for (i = 0; i < M; i++) {
      for (j = 0; j < N; j++)
        sum += matrix[i][j];
  }
  return sum;
}
```

```
int addup2() {
  int i, j, sum = 0;
  for (j = 0; j < N; j++) {
    for (i = 0; i < M; i++)
        sum += matrix[i][j];
  }
  return sum;
}
```

Qual das duas funções em C é mais rápida, `addup1` ou `addup2`?

Acesso Sequencial

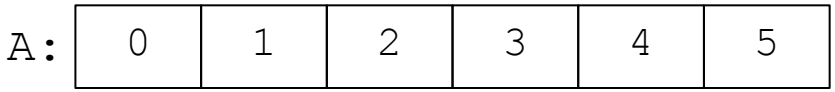
- O hardware da memória é geralmente otimizado para acesso sequencial
  - Se um programa acessa a palavra `i`, o hardware antecipa de várias maneiras que a palavra `i+1` será necessária em breve também

- Portanto, acessar elementos de um arranjo sequencialmente, na mesma ordem em que são armazenados na memória, é mais rápido que acessá-los não sequencialmente

Em que ordem os elementos de um arranjo são armazenados na memória?

Arranjo Unidimensional (1D) em Memória

- Um layout natural: um arranjo de n elementos pode ser armazenado em um bloco $n \times size$ palavras, tal que $size$ é o número de palavras por elemento

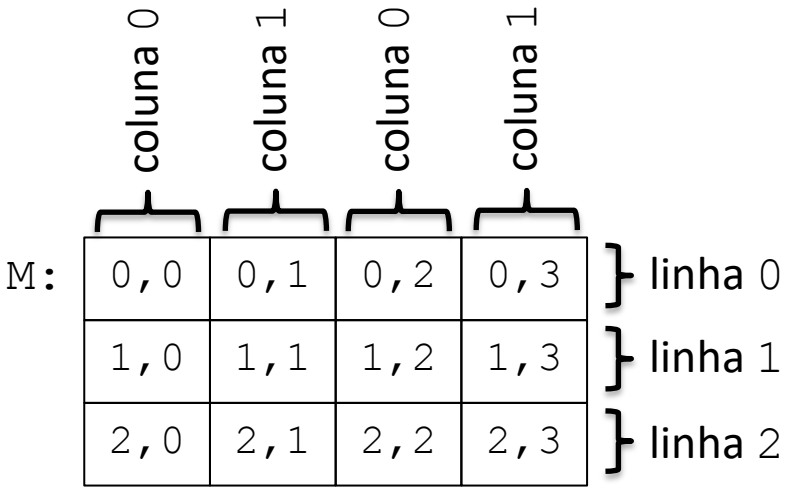


  - O endereço de memória `A[i]` pode ser computado como $base + i \times size$, tal que $base$ é o início do bloco de memória de `A` (assumindo que índices começam em 0)

O acesso sequencial é natural – difícil de evitar

Arranjo Bidimensional (2D) em Memória

- Normalmente são visualizados como um grid

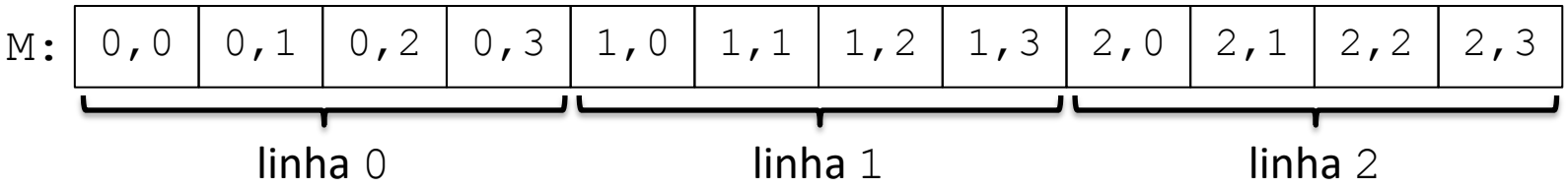


  - O acesso `M[i][j]` é dado pela linha `i` e coluna `j`

Como mapear este grid `3x4` em uma memória que é linear (sequencial)?

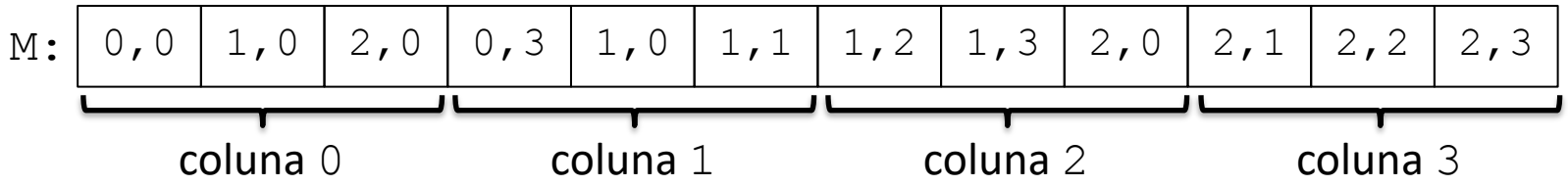
Arranjo Bidimensional (2D) em Memória

- Ordem por linhas (*row-major order*)



  - Uma linha inteira por vez; m por n usa $m \times n \times size$ palavras
  - O endereço `M[i][j]` é $base + (i \times n \times size) + (j \times size)$

- Ordem por colunas (*column-major order*)

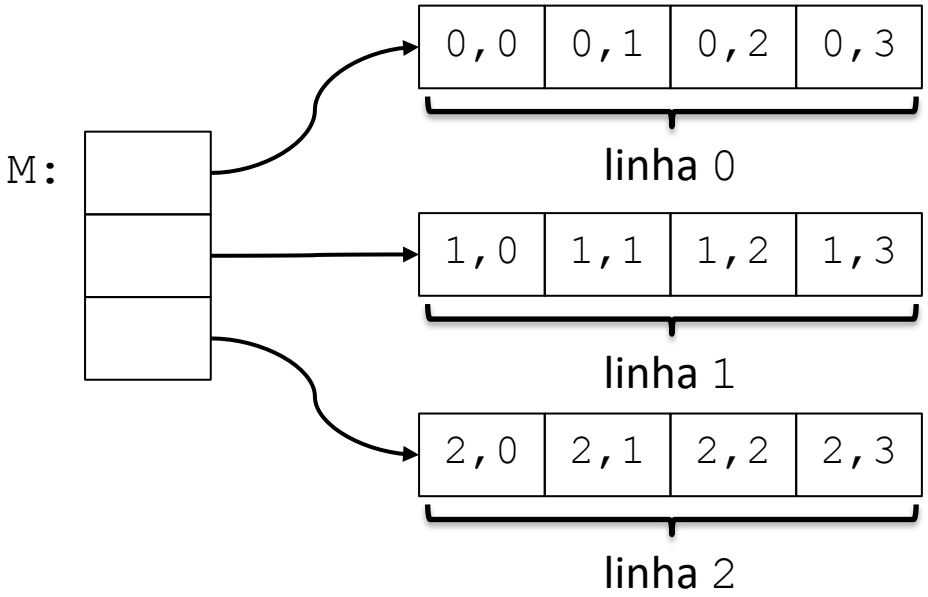


  - Uma coluna inteira por vez; m por n usa $m \times n \times size$ palavras
  - O endereço `M[i][j]` é $base + (i \times size) + (j \times m \times size)$

Qual destas estratégias C usa? Existe outra forma de representação além destas?

Arranjo Bidimensional (2D) em Memória

- Uma outra estratégia comum é tratar um arranjo bidimensional (2D) como um arranjos de ponteiros unidimensionais (1D)



Como fazer isto em C?

  - Cada linha pode ter um tamanho diferente; linhas não utilizadas não precisam ser alocadas
  - Acesso sequencial às linhas inteiras é eficiente, como dado pela ordem por linhas

Arranjos Multidimensionais

- Arranjos bidimensionais podem ser generalizados para dimensões superior (3D, 4D, ...)

- Por exemplo, generalização para acesso usando ordem por linhas

```
for each i0
  for each i1
     ...
       for each in-2
         for each in-1
           access A[i0][i1]...[in-2][in-1]
```

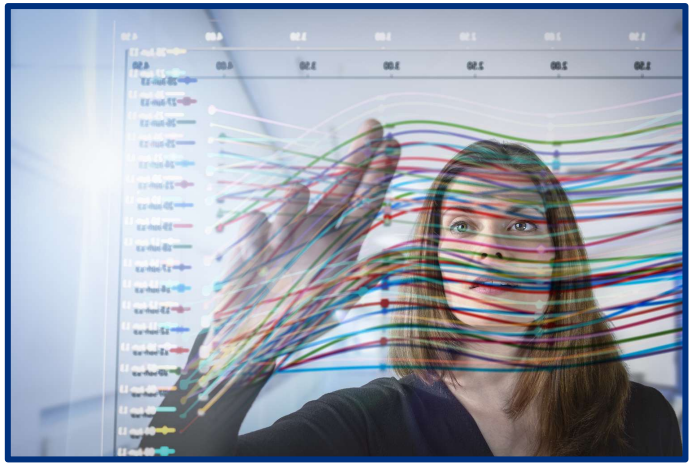
O índice mais a direita varia mais rápido (for mais interno)

Visibilidade de Arranjos

- Em C, a implementação dos arranjos é visível através de aritmética de ponteiros
  - Se `p` é o endereço de `M[i][j]`, então `p+1` é o endereço de `M[i][j+1]` em ordem por linhas
- Fortran também deixa visível
  - Alocações sobrepostas revelam ordem por colunas
- Ada usualmente usa ordem por linha, mas não é visível
  - Programas em Ada ainda funcionariam se o layout de memória fosse alterado

Porém, para todas estas linguagens, a visibilidade é parte do modelo de custos

MODELOS DE CUSTO ESPÚRIOS



Linguagens de Programação

Modelo de Custo Espúrio

- Considere o programa em C que recebe um tamanho via argumentos `size` (ex.: 10 milhões) e chama a função `max` $\texttt{size}^2$ vezes

```
int max(int i, int j) {
  return i > j ? i : j;
}

int main(int argc, char* argv[]) {
  int i, j;
  long long sum = 0;
  int size = atoi(argv[1]);

  for (i = 0; i < size; i++) {
    for (j = 0; i < size; i++)
      sum += max(i, j);
  }

  printf("Sum: %lld\n", sum);
  return 0;
}
```

Quão mais rápido este programa seria se usasse a computação direta `sum += (i>j?i:j)`?

Inlining

- Trocar uma chamada de função pelo corpo da função chamada é chamada de ***inlining***

- É útil para evitar o overhead da chamada da função: empilhar, chamar, desempilhar, retornar

- Normalmente é uma otimização pequena, mas para algo simples como `max` o overhead pode dominar o custo da execução do corpo da função

Modelo de Custo

- O custo da chamada da função é comparável ao custo de execução do corpo de uma função pequena

- Isto guia programadores a usar *inlining* em seus códigos (ou macros em C) ao invés de chamada de funções, principalmente para funções pequenas que são chamadas muitas vezes

Cuidado: este modelo de custo normalmente está errado, qualquer compilador de C de respeito faz *inlining* automaticamente (ex.: `gcc` com `-O2`)

O código anterior executa na mesma velocidade, independente se o *inlining* foi feito manualmente ou pelo compilador

Aplicabilidade

- Não é um fenômeno exclusivo de C, vários sistemas de linguagens para diferentes linguagens fazem *inlining*
  - É especialmente importante, e frequentemente implementado, por linguagens orientadas a objetos

- No geral é um erro tumultuar o programa com cópias manuais de corpos de função (*inlining*)
  - Isto torna o programa difícil de ler e de manter e não o torna mais rápido depois de otimizações

Nos primeiros 10 anos da linguagem, compiladores de C que faziam *inlining* não eram disponíveis: fazia sentido fazer *inlining* manualmente em códigos críticos

Dica: hoje em dia já não faz mais sentido fazer *inlining* manualmente

ISSO É TUDO, PESSOAL!