Cost Models

Which Is Faster?

```
Y=[1|X]

append(X,[1],Y)
```

- Every experienced programmer has a cost model of the language: a mental model of the relative costs of various operations
- Not usually a part of a language specification, but very important in practice

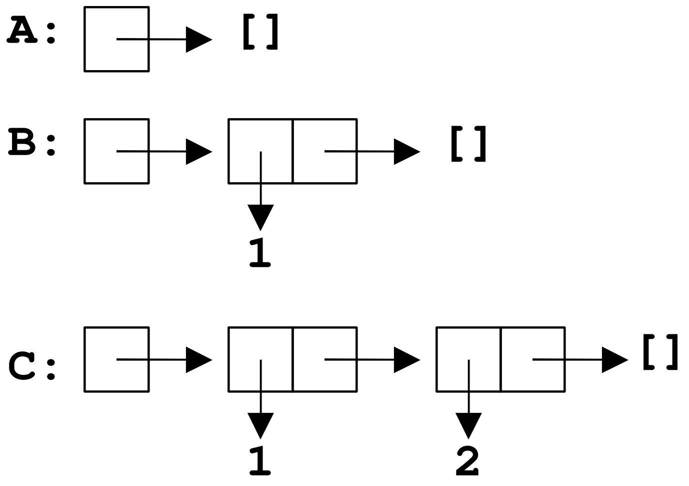
Outline

- A cost model for lists
- A cost model for function calls
- A cost model for Prolog search
- A cost model for arrays
- Spurious cost models

The Cons-Cell List

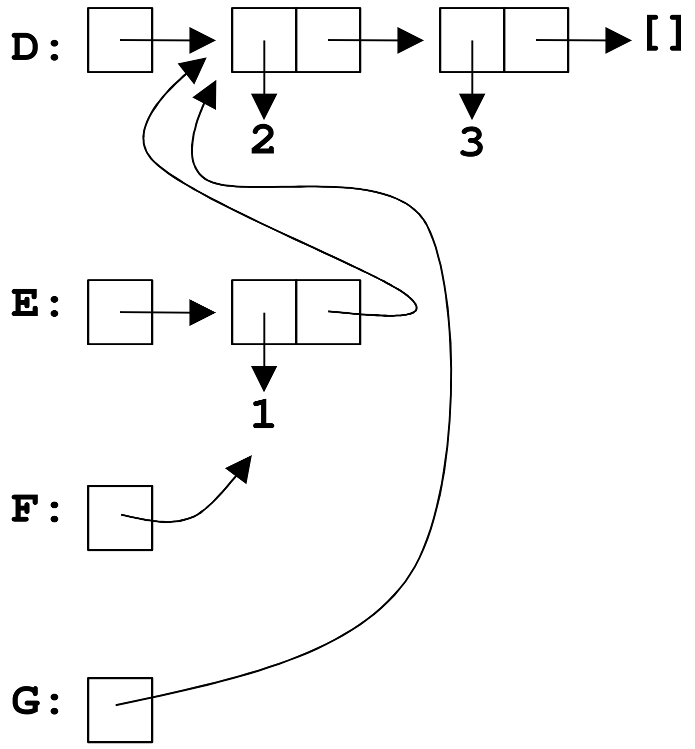
- Used by ML, Prolog, Lisp, and many other languages
- We also implemented this in Java

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



Shared List Structure

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



How Do We Know?

- How do we know Prolog shares list structure—how do we know **`E=[1|D]`** does not make a copy of term **`D`**?
- It observably takes a constant amount of time and space
- This is not part of the formal specification of Prolog, but is part of the cost model

Computing Length

- **`length(X,Y)`** can take no shortcut—it must count the length, like this in ML:

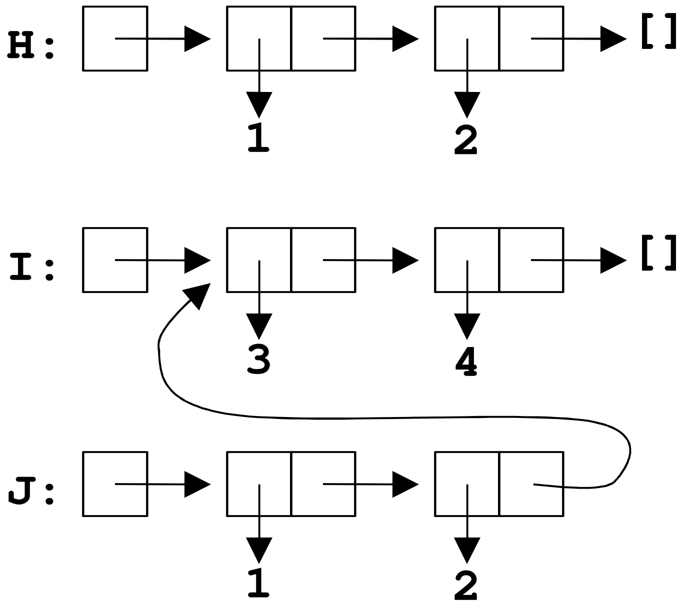
```
fun length nil = 0
|   length (head::tail) = 1 + length tail;
```

- Takes time proportional to the length of the list

Appending Lists

- **`append(H,I,J)`** can also be expensive: it must make a copy of **`H`**

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



Appending

- **`append`** must copy the prefix:

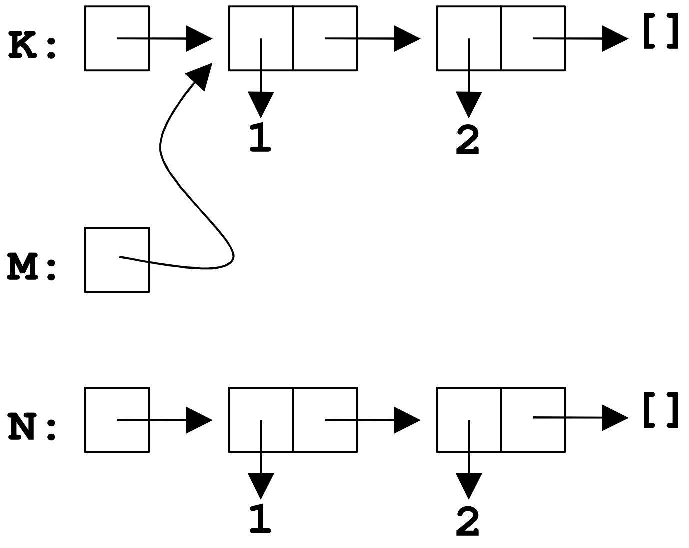
```
append([],X,X).
append([Head|Tail],X,[Head|Suffix]) :-
  append(Tail,X,Suffix).
```

- Takes time proportional to the length of the first list

Unifying Lists

- Unifying lists can also be expensive, since they may or may not share structure:

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



Unifying Lists

- To test whether lists unify, the system must compare them element by element:

```
xequal([],[]).
xequal([Head|Tail1],[Head|Tail2]) :-
  xequal(Tail1,Tail2).
```

- It might be able to take a shortcut if it finds shared structure, but in the worst case it must compare the entire structure of both lists

Cons-Cell Cost Model Summary

- Consing takes constant time
- Extracting head or tail takes constant time
- Computing the length of a list takes time proportional to the length
- Computing the result of appending two lists takes time proportional to the length of the first list
- Comparing two lists, in the worst case, takes time proportional to their size

Application

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

The cost model guides programmers away from solutions like this, which grow lists from the rear

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

This is much faster: linear time instead of quadratic

Exposure

- Some languages expose the shared-structure cons-cell implementation:
  - Lisp programs can test for equality (**`equal`**) or for shared structure (**`eq`**, constant time)
- Other languages (like Prolog and ML) try to hide it, and have no such test
- But the implementation is still visible in the sense that programmers know and use the cost model

Outline

- A cost model for lists
- A cost model for function calls
- A cost model for Prolog search
- A cost model for arrays
- Spurious cost models

Reverse in ML

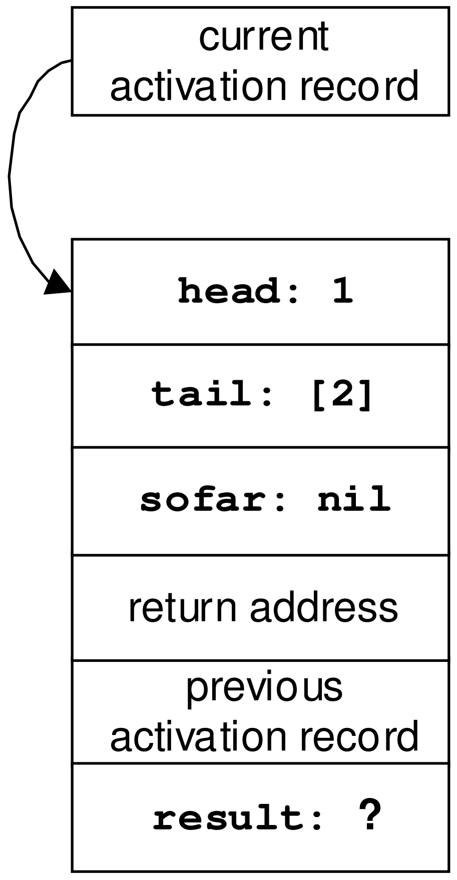
- Here is an ML implementation that works like the previous Prolog **`reverse`**

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

Example

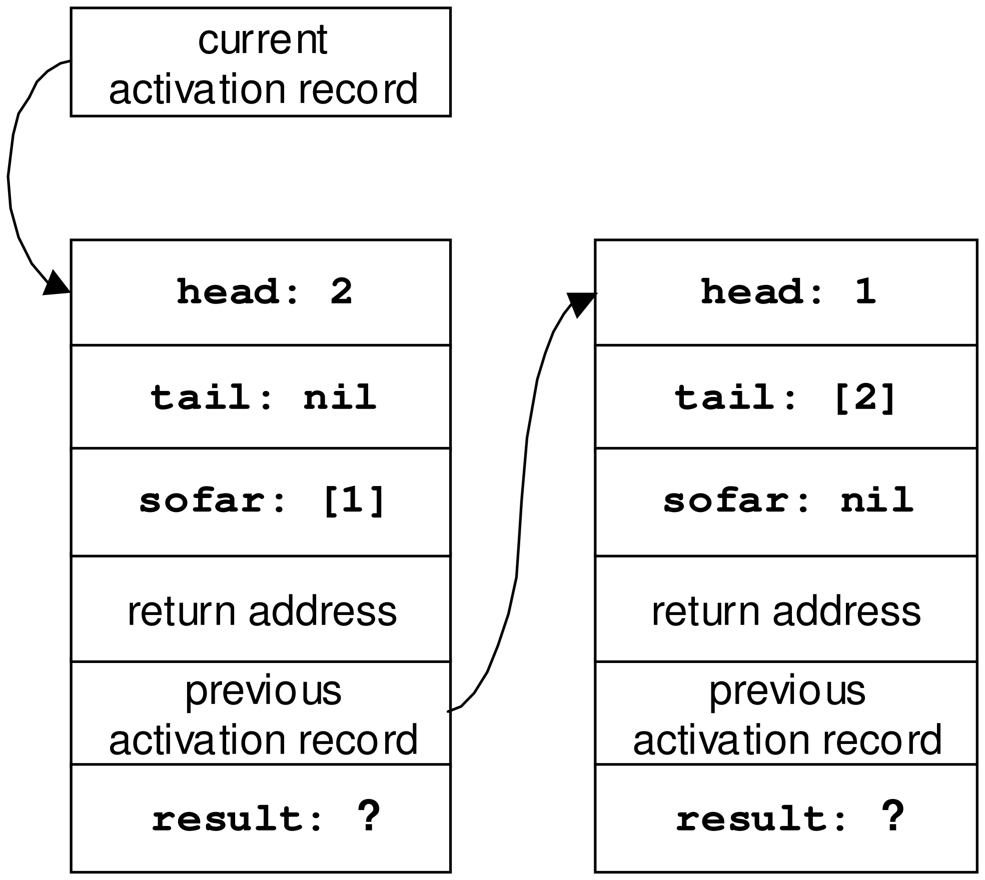
```
fun rev(nil,sofar) = sofar
|   rev(head::tail,sofar) =
          rev(tail,head::sofar);
```

We are evaluating **`rev([1,2],nil)`**. *This shows the contents of memory just before the recursive call that creates a second activation.*



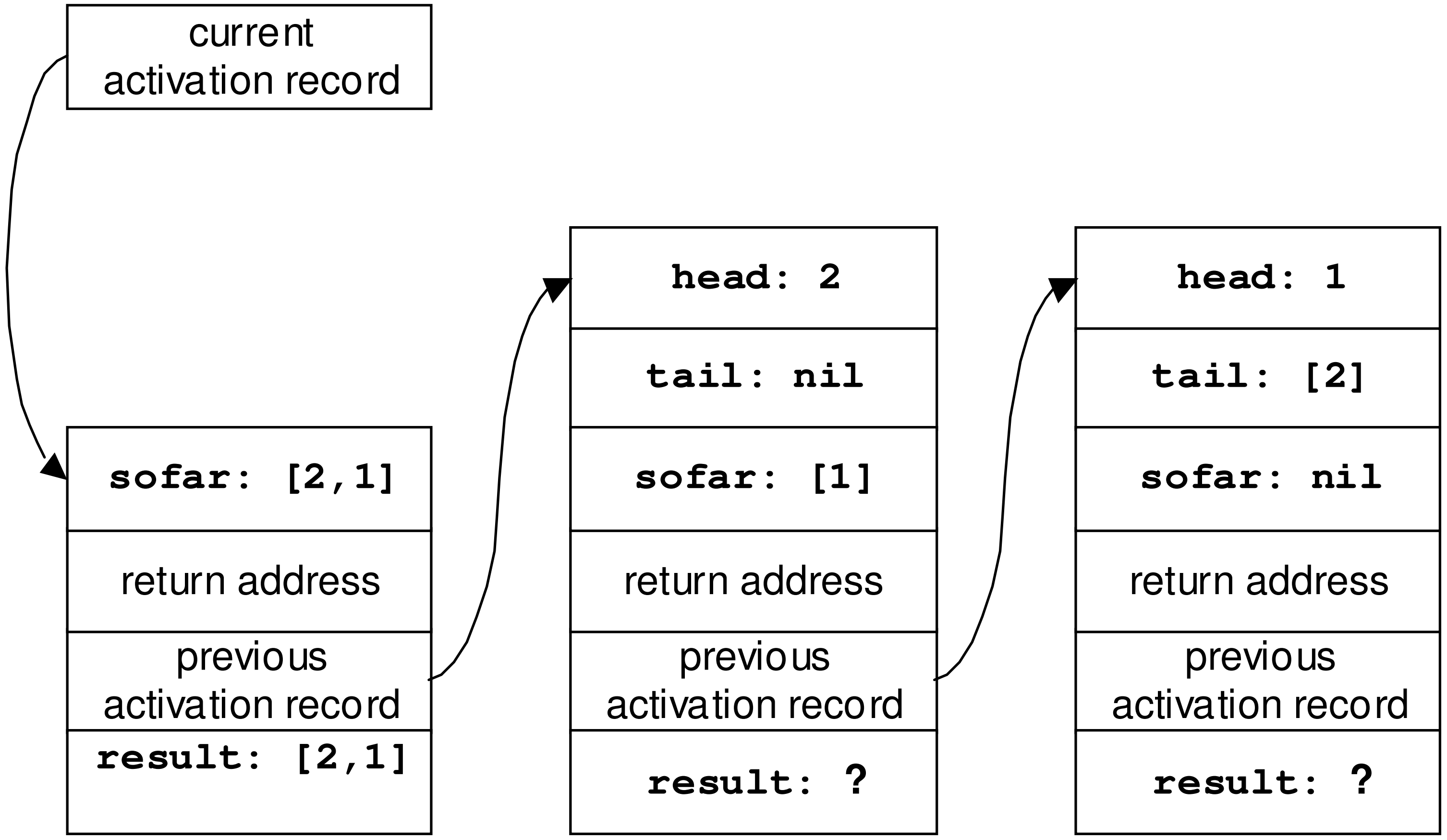
This shows the contents of memory just before the third activation.

```
fun rev(nil,sofar) = sofar
|   rev(head::tail,sofar) =
        rev(tail,head::sofar);
```



This shows the contents of memory just before the third activation returns.

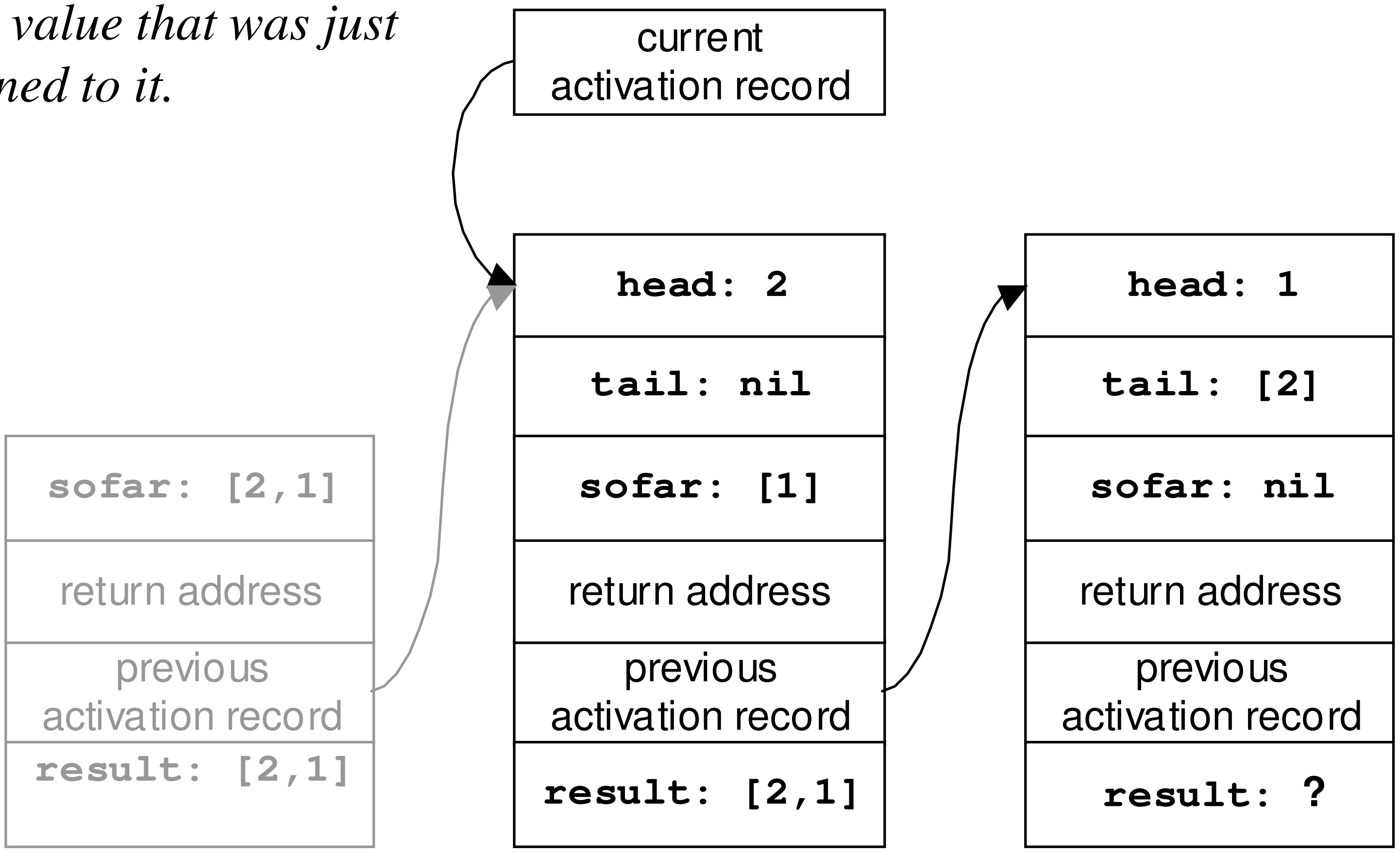
```
fun rev(nil,sofar) = sofar
|   rev(head::tail,sofar) =
        rev(tail,head::sofar);
```



This shows the contents of memory just before the second activation returns.

All it does is return the same value that was just returned to it.

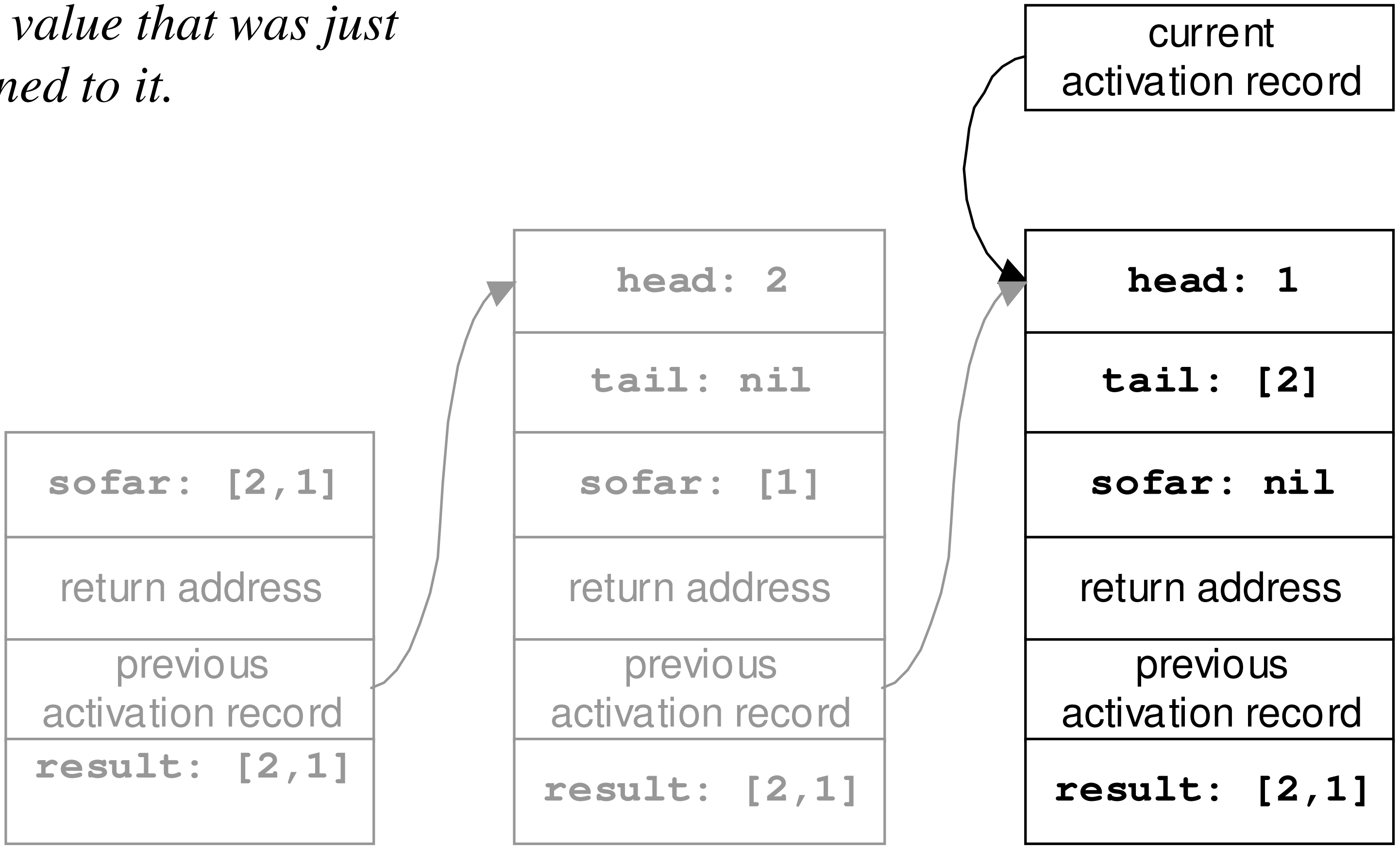
```
fun rev(nil,sofar) = sofar
|   rev(head::tail,sofar) =
        rev(tail,head::sofar);
```



This shows the contents of memory just before the first activation returns.

All it does is return the same value that was just returned to it.

```
fun rev(nil,sofar) = sofar
|   rev(head::tail,sofar) =
         rev(tail,head::sofar);
```



Tail Calls

- A function call is a *tail call* if the calling function does no further computation, but merely returns the resulting value (if any) to its own caller
- All the calls in the previous example were tail calls

Tail Recursion

- A recursive function is *tail recursive* if all its recursive calls are tail calls
- Our **`rev`** function is tail recursive

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

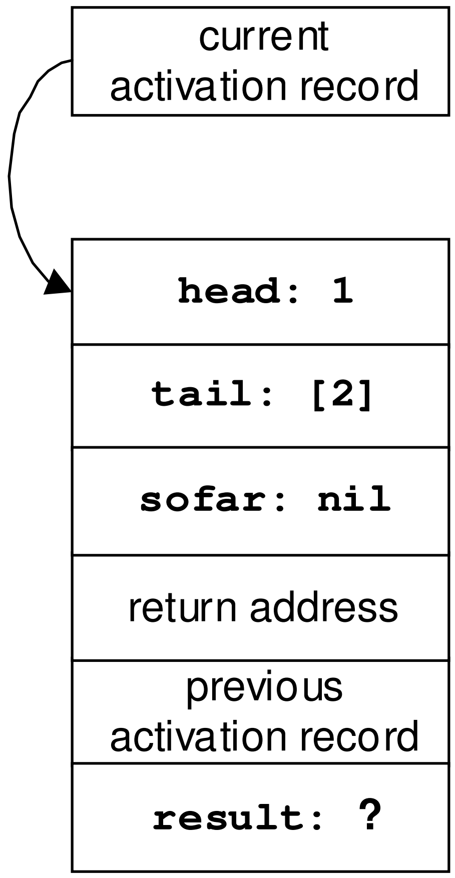
Tail-Call Optimization

- When a function makes a tail call, *it no longer needs its activation record*
- Most language systems take advantage of this to optimize tail calls, by using the same activation record for the called function
  - No need to push/pop another frame
  - Called function returns directly to original caller

Example

```
fun rev(nil,sofar) = sofar
|   rev(head::tail,sofar) =
          rev(tail,head::sofar);
```

We are evaluating **`rev([1,2],nil)`**. *This shows the contents of memory just before the recursive call that creates a second activation.*

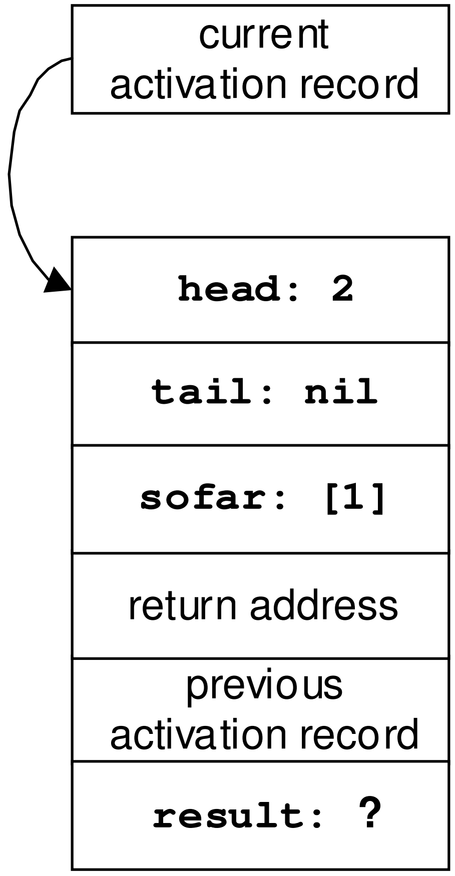


```
fun rev(nil,sofar) = sofar
|   rev(head::tail,sofar) =
          rev(tail,head::sofar);
```

Just before the third activation.

Optimizing the tail call, we reused the same activation record.

The variables are overwritten with their new values.



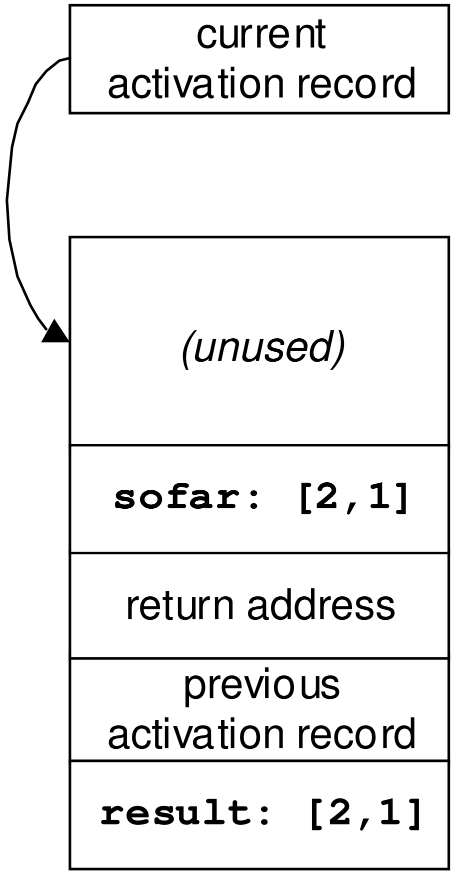
```
fun rev(nil,sofar) = sofar
|   rev(head::tail,sofar) =
        rev(tail,head::sofar);
```

Just before the third activation returns.

Optimizing the tail call, we reused the same activation record again. We did not need all of it.

The variables are overwritten with their new values.

Ready to return the final result directly to **`rev`***'s original caller* (**`reverse`**).



Tail-Call Cost Model

- Under this model, tail calls are significantly faster than non-tail calls
- And they take up less space
- The space consideration may be more important here:
  - tail-recursive functions can take constant space
  - non-tail-recursive functions take space at least linear in the depth of the recursion

Application

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

The cost model guides programmers away from non-tail-recursive solutions like this

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

Although longer, this solution runs faster and takes less space

An accumulating parameter.

Often useful when converting to tail-recursive form

Applicability

- Implemented in virtually all functional language systems; explicitly guaranteed by some functional language specifications
- Also implemented by good compilers for most other modern languages: C, C++, etc.
- One exception: not currently implemented in Java language systems

Prolog Tail Calls

- A similar optimization is done by most compiled Prolog systems
- But it can be a tricky to identify tail calls:

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

- Call of **`r`** above is not (necessarily) a tail call because of possible backtracking
- For the last condition of a rule, when there is no possibility of backtracking, Prolog systems can implement a kind of tail-call optimization

Outline

- A cost model for lists
- A cost model for function calls
- A cost model for Prolog search
- A cost model for arrays
- Spurious cost models

Prolog Search

- We know all the details already:
  - A Prolog system works on goal terms from left to right
  - It tries rules from the database in order, trying to unify the head of each rule with the current goal term
  - It backtracks on failure—there may be more than one rule whose head unifies with a given goal term, and it tries as many as necessary

Application

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

The cost model guides programmers away from solutions like this. Why do all that work if **`X`** *is not male?*

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

Although logically identical, this solution may be much faster since it restricts early.

General Cost Model

- Clause order in the database, and condition order in each rule, can affect cost
- Can’t reduce to simple guidelines, since the best order often depends on the query as well as the database

Outline

- A cost model for lists
- A cost model for function calls
- A cost model for Prolog search
- A cost model for arrays
- Spurious cost models

Multidimensional Arrays

- Many languages support them
- In C:
  ```
  int a[1000][1000];
  ```
- This defines a million integer variables
- One **`a[i][j]`** for each pair of **`i`** and **`j`** with $0 \leq$ **`i`** < 1000 and $0 \leq$ **`j`** < 1000

Which Is Faster?

```
int addup1
    (int a[1000][1000]) {
  int total = 0;
  int i = 0;
  while (i < 1000) {
    int j = 0;
    while (j < 1000) {
      total += a[i][j];
      j++;
    }
    i++;
  }
  return total;
}
```

```
int addup2
    (int a[1000][1000]) {
  int total = 0;
  int j = 0;
  while (j < 1000) {
    int i = 0;
    while (i < 1000) {
      total += a[i][j];
      i++;
    }
    j++;
  }
  return total;
}
```

Varies **`j`** *in the inner loop:*
`a[0][0]` *through* **`a[0][999]`**, *then* **`a[1][0]`** *through* **`a[1][999]`**, …

Varies **`i`** *in the inner loop:*
`a[0][0]` *through* **`a[999][0]`**, *then* **`a[0][1]`** *through* **`a[999][1]`**, …

Sequential Access

- Memory hardware is generally optimized for sequential access
- If the program just accessed word *i*, the hardware anticipates in various ways that word *i*+1 will soon be needed too
- So *accessing array elements sequentially, in the same order in which they are stored in memory, is faster than accessing them non-sequentially*
- In what order are elements stored in memory?

1D Arrays In Memory

- For one-dimensional arrays, a natural layout
- An array of n elements can be stored in a block of $n \times size$ words
  - $size$ is the number of words per element
- The memory address of $A[i]$ can be computed as $base + i \times size$:
  - $base$ is the start of A's block of memory
  - (Assumes indexes start at 0)
- Sequential access is natural—hard to avoid

2D Arrays?

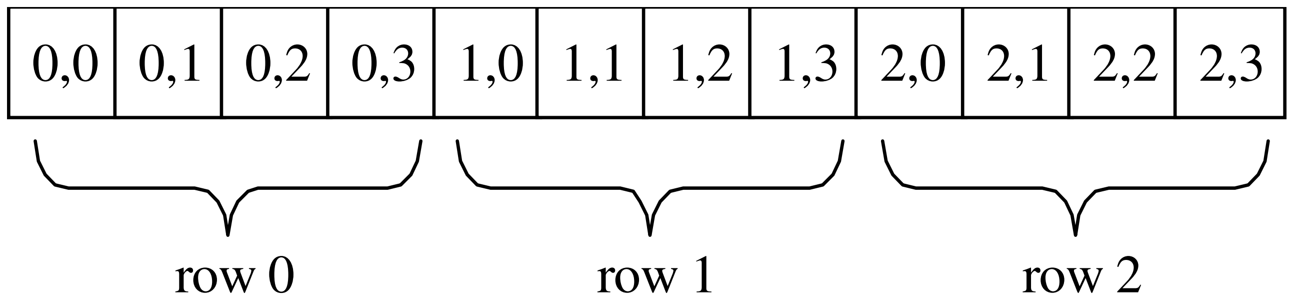
- Often visualized as a grid
- *A*[*i*][*j*] is row *i*, column *j*:

	column 0	column 1	column 2	column 3
row 0	0,0	0,1	0,2	0,3
row 1	1,0	1,1	1,2	1,3
row 2	2,0	2,1	2,2	2,3

A 3-by-4 array: 3 rows of 4 columns

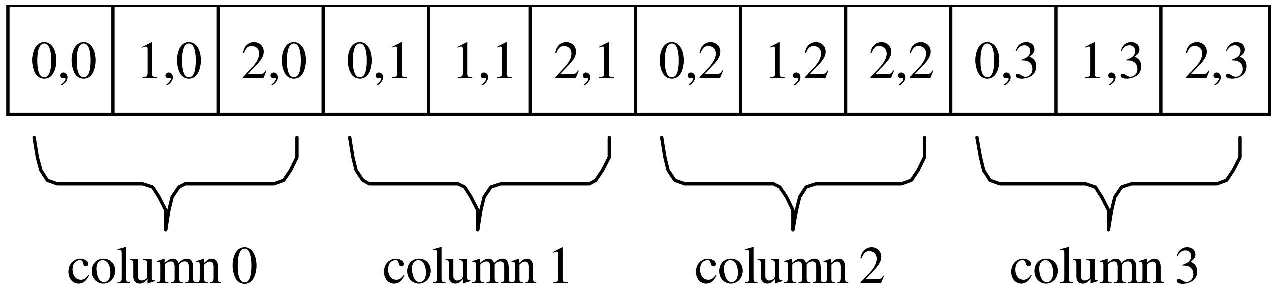
- Must be mapped to linear memory…

Row-Major Order



- One whole row at a time
- An *m*-by-*n* array takes $m \times n \times size$ words
- Address of $A[i][j]$ is
  $base + (i \times n \times size) + (j \times size)$

Column-Major Order



- One whole column at a time
- An m-by-n array takes $m \times n \times size$ words
- Address of $A[i][j]$ is
  $base + (i \times size) + (j \times m \times size)$

So Which Is Faster?

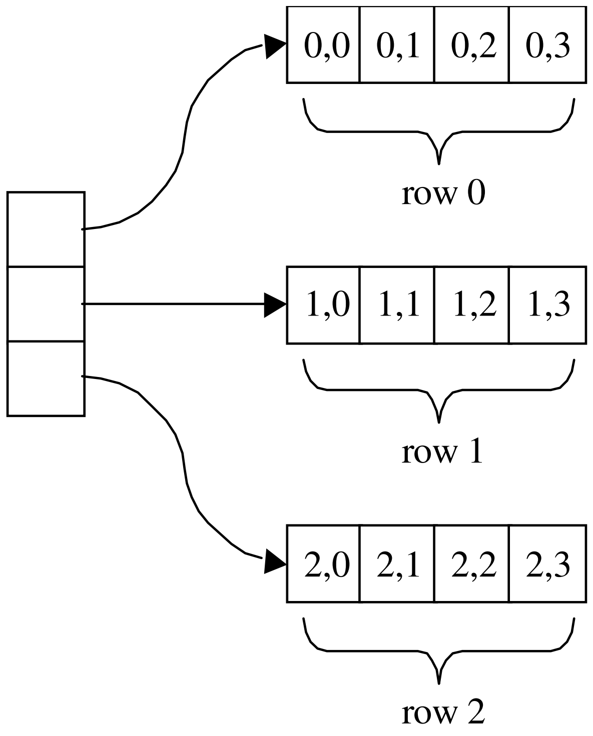
```
int addup1
    (int a[1000][1000]) {
  int total = 0;
  int i = 0;
  while (i < 1000) {
    int j = 0;
    while (j < 1000) {
      total += a[i][j];
      j++;
    }
    i++;
  }
  return total;
}
```

```
int addup2
    (int a[1000][1000]) {
  int total = 0;
  int j = 0;
  while (j < 1000) {
    int i = 0;
    while (i < 1000) {
      total += a[i][j];
      i++;
    }
    j++;
  }
  return total;
}
```

C uses row-major order, so this one is faster: it visits the elements in the same order in which they are allocated in memory.

Other Layouts

- Another common strategy is to treat a 2D array as an array of pointers to 1D arrays
- Rows can be different sizes, and unused ones can be left unallocated
- Sequential access of whole rows is efficient, like row-major order



Higher Dimensions

- 2D layouts generalize for higher dimensions
- For example, generalization of row-major (*odometer order*) matches this access order:

```
for each $i_0$
  for each $i_1$
    . . .
      for each $i_{n-2}$
        for each $i_{n-1}$
          access $A[i_0][i_1]\ldots[i_{n-2}][i_{n-1}]$
```

- Rightmost subscript varies fastest

Is Array Layout Visible?

- In C, it is visible through pointer arithmetic
  - If `p` is the address of `a[i][j]`, then `p+1` is the address of `a[i][j+1]`: row-major order
- Fortran also makes it visible
  - Overlaid allocations reveal column-major order
- Ada usually uses row-major, but hides it
  - Ada programs would still work if layout changed
- But for all these languages, it is visible as a part of the cost model

Outline

- A cost model for lists
- A cost model for function calls
- A cost model for Prolog search
- A cost model for arrays
- Spurious cost models

Question

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

int main() {
  int i,j;
  double sum = 0.0;
  for (i=0; i<10000; i++) {
    for (j=0; j<10000; j++) {
      sum += max(i,j);
    }
  }
  printf("%d\n", sum);
}
```

If we replace this with a direct computation,

```
sum += (i>j?i:j)
```

how much faster will the program be?

Inlining

- Replacing a function call with the body of the called function is called *inlining*
- Saves the overhead of making a function call: push, call, return, pop
- Usually minor, but for something as simple as **`max`** the overhead might dominate the cost of the executing the function body

Cost Model

- Function call overhead is comparable to the cost of a small function body
- This guides programmers toward solutions that use inlined code (or macros, in C) instead of function calls, especially for small, frequently-called functions

Wrong!

- Unfortunately, this model is often wrong
- Any respectable C compiler *can perform inlining automatically*
- (Gnu C does this with **`–O2`** for small functions)
- Our example runs at exactly the same speed whether we inline manually, or let the compiler do it

Applicability

- Not just a C phenomenon—many language systems for different languages do inlining
- (It is especially important, and often implemented, for object-oriented languages)
- Usually it is a mistake to clutter up code with manually inlined copies of function bodies
- It just makes the program harder to read and maintain, but no faster after automatic optimization

Cost Models Change

- For the first 10 years or so, C compilers that could do inlining were not generally available
- It made sense to manually inline in performance-critical code
- Another example is the old **`register`** declaration from C

Conclusion

- Some cost models are language-system-specific: does this C compiler do inlining?
- Others more general: tail-call optimization is a safe bet for all functional language systems and most other language systems
- All are an important part of the working programmer’s expertise, though rarely part of the language specification
- No substitute for good algorithms!