COMP90048: Workshop 7

Eleanor McMurtry, University of Melbourne

Logic Programming, but better

Intro

- In Prolog, it can be easy to accidentally write very inefficient code.

- There are two tricks: **tail recursion** and **asymptotic complexity**.

Tail recursion and the stack

- What happens when you call a function in C?

```
int n = 2;
int x;
x = factorial(n);
// x = 2

int factorial(int n) {
    return n <= 1
        ? 1
        : (n * factorial(n - 1);
}
```

The stack

Tail recursion and the stack

- What happens when you call a function in C?

```c
int n = 2;
int x;
x = factorial(n);
// x = 2

int factorial(int n) {
    return n <= 1
        ? 1
        : (n * factorial(n - 1);
}
```

The stack*

n (4 bytes)

* it grows downwards by convention

Tail recursion and the stack

- What happens when you call a function in C?

```
int n = 2;
int x;
x = factorial(n);
// x = 2

int factorial(int n) {
    return n <= 1
        ? 1
        : (n * factorial(n - 1);
}
```

The stack

n (4 bytes)
x (4 bytes)

Tail recursion and the stack

- What happens when you call a function in C?

```
int n = 2;
int x;
x = factorial(n);
// x = 2

int factorial(int n) {
    return n <= 1
        ? 1
        : (n * factorial(n - 1);
}
```

The stack

n (4 bytes)
x (4 bytes)
return address (8 bytes)
n (4 bytes)

Tail recursion and the stack

- What happens when you call a function in C?

```
int n = 2;
int x;
x = factorial(n);
// x = 2

int factorial(int n) {
    return n <= 1
        ? 1
        : (n * factorial(n - 1);
}
```

the calling function's **stack frame** => extra memory used

The stack

`n (4 bytes)`
`x (4 bytes)`
`return address (8 bytes)`
`n (4 bytes)`

Tail recursion and the stack

- What happens when you call a function in C?

```
int n = 2;
int x;
x = factorial(n);
// x = 2

int factorial(int n) {
    return n <= 1
        ? 1
        : (n * factorial(n - 1);
}
```

The stack

n (4 bytes)
x (4 bytes)
return address (8 bytes)
n (4 bytes)

Tail recursion and the stack

- What happens when you call a function in C?

```
int n = 2;
int x;
x = factorial(n);
// x = 2

int factorial(int n) {
    return n <= 1
        ? 1
        : (n * factorial(n - 1);
}
```

The stack

n (4 bytes)
x (4 bytes)
return address (8 bytes)
n (4 bytes)
return address (8 bytes)
n - 1 (4 bytes)

Tail recursion and the stack

- What happens when you call a function in C?

```
int n = 2;
int x;
x = factorial(n);
// x = 2

int factorial(int n) {
    return n <= 1
        ? 1
        : (n * factorial(n - 1);
}
```

The stack

n (4 bytes)
x (4 bytes)
return address (8 bytes)
n (4 bytes)
return address (8 bytes)
n - 1 (4 bytes)

Tail recursion and the stack

- What happens when you call a function in C?

```
int n = 2;
int x;
x = factorial(n);
// x = 2

int factorial(int n) {
    return n <= 1
        ? 1
        : (n * factorial(n - 1);
}
```

The stack

	n (4 bytes)
	x (4 bytes)
	return address (8 bytes)
	n (4 bytes)
popped	~~return address (8 bytes)~~
	~~n = 1 (4 bytes)~~

Tail recursion and the stack

- What happens when you call a function in C?

```c
int n = 2;
int x;
x = factorial(n);
// x = 2

int factorial(int n) {
    return n <= 1
        ? 1
        : (n * factorial(n - 1);
}
```

The stack

n (4 bytes)
x (4 bytes)
return address (8 bytes)
n * 1 (4 bytes)

Tail recursion and the stack

- What happens when you call a function in C?

```
int n = 2;
int x;
x = factorial(n);
// x = 2

int factorial(int n) {
    return n <= 1
          ? 1
          : (n * factorial(n - 1);
}
```

The stack

n (4 bytes)
x (4 bytes)
~~return address (8 bytes)~~
~~n * 1 (4 bytes)~~

Tail recursion and the stack

- What happens when you call a function in C?

```
int n = 2;
int x;
x = factorial(n);
// x = 2

int factorial(int n) {
    return n <= 1
        ? 1
        : (n * factorial(n - 1);
}
```

The stack

n (4 bytes)
x (4 bytes)

Tail recursion and the stack

- We've got two stack entries per call...

```
int n = 2;
int x;
x = factorial(n);
// x = 2

int factorial(int n) {
    return n <= 1
        ? 1
        : (n * factorial(n - 1);
}
```

The stack

n (4 bytes)
x (4 bytes)
return address (8 bytes)
n (4 bytes)
return address (8 bytes)
n - 1 (4 bytes)

Tail recursion and the stack

- Now with tail recursion!

```
int n = 2;
int x;
x = factorial(n);
// x = 2

int factorial(int n) {
    return n <= 1
        ? 1
        : (n * factorial(n - 1);
}
```

The stack

n (4 bytes)
x (4 bytes)
return address (8 bytes)
n (4 bytes)

Tail recursion and the stack

- Now with tail recursion!

```
int n = 2;
int x;
x = factorial(n);
// x = 2

int factorial(int n) {
    return n <= 1
        ? 1
        : (n * factorial(n - 1);
}
```

doesn't need to create new stack frame

The stack

The stack
n (4 bytes)
x (4 bytes)
return address (8 bytes)
n (4 bytes)
n - 1 (4 bytes)

Tail recursion and the stack

- Now with tail recursion!

```
int n = 2;
int x;
x = factorial(n);
// x = 2

int factorial(int n) {
    return n <= 1
        ? 1
        : (n * factorial(n - 1);
}
```

The stack

n (4 bytes)
x (4 bytes)
return address (8 bytes)
n (4 bytes)
n - 1 (4 bytes)

Tail recursion and the stack

- Now with tail recursion!

```
int n = 2;
int x;
x = factorial(n);
// x = 2

int factorial(int n) {
    return n <= 1
        ? 1
        : (n * factorial(n - 1);
}
```

The stack

n (4 bytes)
x (4 bytes)
return address (8 bytes)
n * 1 (4 bytes)
n - 1 (4 bytes)

Tail recursion and the stack

- Now with tail recursion!

```
int n = 2;
int x;
x = factorial(n);
// x = 2

int factorial(int n) {
    return n <= 1
        ? 1
        : (n * factorial(n - 1);
}
```

The stack

n (4 bytes)
x (4 bytes)
~~return address (8 bytes)~~
~~n * 1 (4 bytes)~~
~~n - 1 (4 bytes)~~

Tail recursion and the stack

- By reducing stack use, we save both memory and time.

```
int n = 2;
int x;
x = factorial(n);
// x = 2

int factorial(int n) {
    return n <= 1
        ? 1
        : (n * factorial(n - 1);
}
```

The stack

n (4 bytes)
x (4 bytes)

Tail recursion and the stack

- By reducing stack use, we save both memory and time.

The stack (no tail recursion)

`n (4 bytes)`
`x (4 bytes)`
`return address (8 bytes)`
`n (4 bytes)`
`return address (8 bytes)`
`n - 1 (4 bytes)`

The stack (tail recursion)

`n (4 bytes)`
`x (4 bytes)`
`return address (8 bytes)`
`n * 1 (4 bytes)`
`n - 1 (4 bytes)`

Tail recursion and the stack

- We can do this because **recursion is the last step.**

```
int factorial(int n) {
    return n <= 1
        ? 1
        : (n * factorial(n - 1);
}
```

Tail recursion in Prolog

- We introduce an **accumulator** (and a helper predicate).

```
prodlist(List,Prod):-
    prodlist_acc(List, 1, Prod).

prodlist_acc([], Prod, Prod).
% to be continued
```

- The idea: `prodlist_acc` holds when its accumulator is the product.

Tail recursion in Prolog

- The same trick applies in Prolog.

```
prodlist([], 1).

prodlist([N|Ns], Prod):-
    prodlist(Ns, Prod0),
    Prod0 is N * Prod.
```

Tail recursion in Prolog

- We introduce an **accumulator** (and a helper predicate).

```
prodlist(List, Prod):-
    prodlist_acc(List, 1, Prod).

prodlist_acc([], Prod, Prod).
prodlist_acc([N|Ns], Acc, Prod):-
    Prod0 is Acc * N,
    prodlist_acc(Ns, Prod0, Prod).
```

Avoiding append

- Similarly to Haskell, `append/3` is O(n). We try to avoid it where we can. Consider:

```
reverse([], []).

reverse([A|BC], CBA):-
    reverse(BC, CB),
    append(CB, [A], CBA).
```

Avoiding append

- We can use the accumulator approach here too!

```
reverse(ABC, CBA):-
    reverse_acc(ABC, [], CBA).

reverse_acc([], A, A).

reverse_acc([B|CD], Acc, DCBAcc):-
    reverse_acc(CD, [B|Acc], DCBAcc).
```

`DCBAcc` is the reversed list, with `Acc` on the end

Avoiding append

- We can use the accumulator approach here too!

```
reverse(ABC, CBA):-
    reverse_acc(ABC, [], CBA).

reverse_acc([], A, A).

reverse_acc([B|CD], Acc, DCBAcc):-
    reverse_acc(CD, [B|Acc], DCBAcc).
```

`DCBAcc` is the reversed list, with `Acc` on the end

peel `B` from the **front** of the original list `BCD` and stick it at the **back** of the new list `Acc`

Avoiding append

- We can use the accumulator approach here too!

```
reverse([], A, A).

reverse([B|CD], Acc, DCBAcc):-
    reverse(CD, [B|Acc], DCBAcc).
```

- This is a general method for working with lists.

Using the Prolog debugger to understand `reverse_acc/3`

```
[trace]  ?- reverse([a, b, c], List).
   Call: (8) reverse([a, b, c], _6824) ? creep
   Call: (9) reverse_acc([a, b, c], [], _6824) ? creep
   Call: (10) reverse_acc([b, c], [a], _6824) ? creep
   Call: (11) reverse_acc([c], [b, a], _6824) ? creep
   Call: (12) reverse_acc([], [c, b, a], _6824) ? creep
   Exit: (12) reverse_acc([], [c, b, a], [c, b, a]) ? creep
   Exit: (11) reverse_acc([c], [b, a], [c, b, a]) ? creep
   Exit: (10) reverse_acc([b, c], [a], [c, b, a]) ? creep
   Exit: (9) reverse_acc([a, b, c], [], [c, b, a]) ? creep
   Exit: (8) reverse([a, b, c], [c, b, a]) ? creep
List = [c, b, a].
```

peel `a` off the front

Using the Prolog debugger to understand `reverse_acc/3`

```
[trace]  ?- reverse([a, b, c], List).
   Call: (8) reverse([a, b, c], _6824) ? creep
   Call: (9) reverse_acc([a, b, c], [], _6824) ? creep
   Call: (10) reverse_acc([b, c], [a], _6824) ? creep
   Call: (11) reverse_acc([c], [b, a], _6824) ? creep
   Call: (12) reverse_acc([], [c, b, a], _6824) ? creep
   Exit: (12) reverse_acc([], [c, b, a], [c, b, a]) ? creep
   Exit: (11) reverse_acc([c], [b, a], [c, b, a]) ? creep
   Exit: (10) reverse_acc([b, c], [a], [c, b, a]) ? creep
   Exit: (9) reverse_acc([a, b, c], [], [c, b, a]) ? creep
   Exit: (8) reverse([a, b, c], [c, b, a]) ? creep
List = [c, b, a].
```

← peel `a` off the front

← accumulator done

Using the Prolog debugger to understand `reverse_acc/3`

```
[trace]  ?- reverse([a, b, c], List).
   Call: (8) reverse([a, b, c], _6824) ? creep
   Call: (9) reverse_acc([a, b, c], [], _6824) ? creep
   Call: (10) reverse_acc([b, c], [a], _6824) ? creep
   Call: (11) reverse_acc([c], [b, a], _6824) ? creep
   Call: (12) reverse_acc([], [c, b, a], _6824) ? creep
   Exit: (12) reverse_acc([], [c, b, a], [c, b, a]) ? creep
   Exit: (11) reverse_acc([c], [b, a], [c, b, a]) ? creep
   Exit: (10) reverse_acc([b, c], [a], [c, b, a]) ? creep
   Exit: (9) reverse_acc([a, b, c], [], [c, b, a]) ? creep
   Exit: (8) reverse([a, b, c], [c, b, a]) ? creep
List = [c, b, a].
```

← peel `a` off the front

← accumulator done

← unify accumulator with List

Continue with Grok Workshop 7 (Week 8).