A Functional Space Model

Speaker: David Walker
COS 326
Princeton University
Interlude

Can you tell a coder from a cannibal? A mathematician from a murderer? Try to spot who liked hacking away at corpses rather than computers

https://vole.wtf/coder-serial-killer-quiz/
Understanding the space complexity of functional programs

- At least two interesting components:
  - the amount of *live space* at any instant in time
  - the *rate of allocation*
    - a function call may not change the amount of live space by much but may allocate at a substantial rate
    - because functional programs act by generating new data structures and discarding old ones, they often allocate a lot
      » OCaml garbage collector is optimized with this in mind
      » *interesting fact*: at the assembly level, the number of writes by a functional program is roughly the same as the number of writes by an imperative program

- *What takes up space?*
  - conventional first-order data: tuples, lists, strings, datatypes
  - function representations (closures)
  - the call stack
CONVENTIONAL DATA
OCaml Representations for Data Structures

Type:

```
type triple = int * char * int
```

Representation:

(3, 'a', 17)
OCaml Representations for Data Structures

Type:

```
type mylist = int list
```

Representation:

- `[ ]`
- `[3; 4; 5]`
Space Model

Type:

```
type tree = Leaf | Node of int * tree * tree
```

Representation:

- **Leaf**
  - 0

- **Node(3, left, right)**

Actually like this in Ocaml:

```
Node 3 left right
```
Allocating space

In C, you allocate when you call “malloc”

In Java, you allocate when you call “new”

What about ML?
Whenever you *use a constructor*, space is allocated:

```ocaml
let rec insert (t:tree) (i:int) =
  match t with
  Leaf -> Node (i, Leaf, Leaf)
| Node (j, left, right) ->
  if i <= j then
    Node (j, insert left i, right)
  else
    Node (j, left, insert right i)
```
Whenever you use a constructor, space is allocated:

```
let rec insert (t:tree) (i:int) =
  match t with
  Leaf -> Node (i, Leaf, Leaf)
| Node (j, left, right) ->
    if i <= j then
      Node (j, insert left i, right)
    else
      Node (j, left, insert right i)
```

Consider:

```
insert t 21
```
Whenever you use a constructor, space is allocated:

```ocaml
defined let rec insert (t:tree) (i:int) = 
    match t with 
    | Leaf -> Node (i, Leaf, Leaf) 
    | Node (j, left, right) -> 
        if i <= j then 
            Node (j, insert left i, right) 
        else 
            Node (j, left, insert right i)
```

Consider: 
```
insert t 21
```
Whenever you use a constructor, space is allocated:

```ocaml
define function insert (t:tree) (i:int) =
    match t with
    Leaf -> Node (i, Leaf, Leaf)
| Node (j, left, right) ->
    if i <= j then
        Node (j, insert left i, right)
    else
        Node (j, left, insert right i)
```

Consider:
```
insert t 21
```
Whenever you use a constructor, space is allocated:

```ocaml
definition insert(t:tree) (i:int) =
  match t with
  | Leaf       -> Node (i, Leaf, Leaf)
  | Node (j, left, right) ->
    if i <= j then
      Node (j, insert left i, right)
    else
      Node (j, left, insert right i)
```

Consider:

```ocaml
insert t 21
```
Whenever you use a constructor, space is allocated:

```ocaml
let rec insert (t:tree) (i:int) =
  match t with
  | Leaf -> Node (i, Leaf, Leaf)
  | Node (j, left, right) ->
    if i <= j then
      Node (j, insert left i, right)
    else
      Node (j, left, insert right i)
```

Consider:

```
insert t 21
```
Whenever you use a constructor, space is allocated:

```
let rec insert (t:tree) (i:int) =
    match t with
    | Leaf -> Node (i, Leaf, Leaf)
    | Node (j, left, right) ->
        if i <= j then
            Node (j, insert left i, right)
        else
            Node (j, left, insert right i)
```

Consider:
```
insert t 21
```
Whenever you use a constructor, space is allocated:

```ocaml
let rec insert (t:tree) (i:int) =
  match t with
  | Leaf -> Node (i, Leaf, Leaf)
  | Node (j, left, right) ->
    if i <= j then
      Node (j, insert left i, right)
    else
      Node (j, left, insert right i)
```

Total space allocated is proportional to the height of the tree.

$\sim \log n$, if tree with $n$ nodes is balanced
Net space allocated

The garbage collector reclaims unreachable data structures on the heap.

let fiddle (t: tree) =
    insert t 21

John McCarthy invented GC
1960
(PhD Princeton 1951, student of Alonzo Church)
The garbage collector reclaims unreachable data structures on the heap.

```fsharp
let fiddle (t: tree) = insert t 21
```

If `t` is dead (unreachable),
The garbage collector reclaims unreachable data structures on the heap.

If \( t \) is dead (unreachable),

Then all these nodes will be reclaimed!

```
let fiddle (t: tree) =
  insert t 21
```
Net space allocated

The garbage collector reclaims unreachable data structures on the heap.

let fiddle (t: tree) =
   insert t 21

Net new space allocated:
   1 node

(just like “imperative” version of binary search trees)
But what if you want to keep the old tree?

```
let faddle (t: tree) = (t, insert t 21)
```
But what if you want to keep the old tree?

```
let faddle (t: tree) = (t, insert t 21)
```

[Diagram showing a tree structure]

Net new space allocated:
log(N) nodes

but note: “imperative” version would have to copy the old tree, space cost N new nodes!
let check_option (o:int option) : int option =
  match o with
  | Some _ -> o
  | None -> failwith "found none"

let check_option (o:int option) : int option =
  match o with
  | Some j -> Some j
  | None -> failwith "found none"
let check_option (o:int option) : int option =
    match o with
    | Some _ -> o
    | None -> failwith "found none"

let check_option (o:int option) : int option =
    match o with
    | Some j -> Some j
    | None -> failwith "found none"
let cadd \((c1: \text{int}*\text{int})\) \((c2: \text{int}*\text{int})\) : \text{int}*\text{int} =
let \((x_1, y_1) = c1\) in
let \((x_2, y_2) = c2\) in
\((x_1+x_2, y_1+y_2)\)

let double \((c1: \text{int}*\text{int})\) : \text{int}*\text{int} =
let \(c2 = c1\) in
cadd \(c1\) \(c2\)

let double \((c1: \text{int}*\text{int})\) : \text{int}*\text{int} =
cadd \(c1\) \(c1\)

let double \((c1: \text{int}*\text{int})\) : \text{int}*\text{int} =
let \((x_1, y_1) = c1\) in
cadd \((x_1, y_1)\) \((x_1, y_1)\)
let cadd (c1:int*int) (c2:int*int) : int*int =
let (x1,y1) = c1 in
let (x2,y2) = c2 in
(x1+x2, y1+y2)

let double (c1:int*int) : int*int =
let c2 = c1 in
cadd c1 c2

let double (c1:int*int) : int*int =
cadd c1 c1

let double (c1:int*int) : int*int =
let (x1,y1) = c1 in
cadd (x1,y1) (x1,y1)
let cadd (c1:int*int) (c2:int*int) : int*int =
let (x1,y1) = c1 in
let (x2,y2) = c2 in
(x1+x2, y1+y2)

let double (c1:int*int) : int*int =
let c2 = c1 in
cadd c1 c2

let double (c1:int*int) : int*int =
cadd c1 c1

let double (c1:int*int) : int*int =
let (x1,y1) = c1 in
cadd (x1,y1) (x1,y1)
```ocaml
define cadd (c1:int*int) (c2:int*int) : int*int =
  let (x1,y1) = c1 in
  let (x2,y2) = c2 in
  (x1+x2, y1+y2)

let double (c1:int*int) : int*int =
  let c2 = c1 in
  cadd c1 c2

let double (c1:int*int) : int*int =
  cadd c1 c1

let double (c1:int*int) : int*int =
  let (x1,y1) = c1 in
  cadd (x1,y1) (x1,y1)
```
let cadd (c1:int*int) (c2:int*int) : int*int =
let (x1,y1) = c1 in
let (x2,y2) = c2 in
(x1+x2, y1+y2)

let double (c1:int*int) : int*int =
let c2 = c1 in
cadd c1 c2

let double (c1:int*int) : int*int =
cadd c1 c1

let double (c1:int*int) : int*int =
let (x1,y1) = c1 in
cadd (x1,y1) (x1,y1)
let cadd (c1:int*int) (c2:int*int) : int*int = 
let (x1,y1) = c1 in 
let (x2,y2) = c2 in 
(x1+x2, y1+y2)

let double (c1:int*int) : int*int = 
let (x1,y1) = c1 in 
cadd c1 c1

double does not allocate

extracts components: it is a read
FUNCTION CLOSURES
Closures (A reminder)

Nested functions like bar often contain free variables:

```ocaml
let foo y =  
  let bar x = x + y in  
  bar
```

Here's bar on its own:

```ocaml
let bar x = x + y
```

y is free in the definition of bar

To implement bar, the compiler creates a closure, which is a pair of code for the function plus an environment holding the free variables.
But what about nested, higher-order functions?

bar again:

```
let bar x = x + y
```

bar's representation:

```
let f2 (n, env) =
  n + env.y

{y = 1}
```
But what about nested, higher-order functions?

To estimate the (heap) space used by a program, we often need to estimate the (heap) space used by its closures.

Our estimate will include the cost of the pair:

- **two pointers = 2 words** (8 bytes each, or 4 bytes each on some machines)
- the cost of the environment (1 word in this case).
- but not: the cost of the code (because the same code is reused in every closure of this function)
Space Model Summary

Understanding space consumption in FP involves:

• understanding the difference between
  • live space
  • rate of allocation

• understanding where allocation occurs
  • any time a constructor is used
  • whenever closures are created

• understanding the costs of
  • data types (fairly similar to Java)
  • costs of closures (cost of a pair of pointers + environment)
Assume 8-byte words. Estimate the size of the data structure generated by a call to `goo` (respectively `gah`) in terms of their arguments `n`. Explain your work. Discuss.