A Functional Space Model

Speaker: David Walker
COS 326
Princeton University

slides copyright 2020 David Walker and Andrew W. Appel
permission granted to reuse these slides for non-commercial educational purposes
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]
```

```
0  3  4  5  0
```
Type:

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

Representation:

Leaf

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
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)
```

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

```ocaml
define 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)
```

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)
```

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.

~ log n, if tree with n nodes is balanced
The garbage collector reclaims unreachable data structures on the heap.

```plaintext
let fiddle (t: tree) = insert t 21
```
The garbage collector reclaims unreachable data structures on the heap.

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

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

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

If `t` is dead (unreachable),

Then all these nodes will be reclaimed!
The garbage collector reclaims unreachable data structures on the heap.

```ocaml
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)
Net space allocated

But what if you want to keep the old tree?

let faddle \( (t: \text{tree}) = (t, \text{insert } t \ 21) \)

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: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 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 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)

no allocation

no allocation

allocates 2 pairs (unless the compiler happens to optimize...).
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

let cadd (c1:int*int) (c2:int*int) : int*int =
let (x1,y1) = c1 in
let (x2,y2) = c2 in
(x1+x2, y1+y2)
FUNCTION CLOSURES
Closures (A reminder)

Nested functions like bar often contain free variables:

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

Here's bar on its own:

```plaintext
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)
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)
Exercise

```ml
let rec gen n =  
  if n <= 0 then  
    []  
  else  
    n::gen (n-1)

let rec goo n =  
  if n <= 0 then  
    []  
  else  
    (fun () -> gen n)::goo (n-1)

let rec gah n =  
  let n <= 0 then  
    []  
  else  
    let l = gen n in  
    (fun () -> l)::goo (n-1)
```

Assume 8-byte words. Estimate the size of the data structure generated by a call to \texttt{goo} (respectively \texttt{gah}) in terms of their arguments \texttt{n}. Explain your work. Discuss.