A Functional Space Model

COS 326
David Walker
Princeton University

slides copyright 2017 David Walker
permission granted to reuse these slides for non-commercial educational purposes
Because Halloween draws nigh:

Serial killer or programming languages researcher?

http://www.malevole.com/mv/misc/killerquiz/
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
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:

```ocaml
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]
Space Model

Type:

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

Representation:

- Leaf
- Node(3, left, right)

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:

```
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:

```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
def 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:

```
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:

```ml
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
The garbage collector reclaims unreachable data structures on the heap.

let fiddle (t: tree) =
insert t 21

John McCarthy invented g.c. 1960
Net space allocated

The garbage collector reclaims unreachable data structures on the heap.

let fiddle (t: tree) =
    insert t 21

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

Let \( fiddle \) be defined as:

\[
\text{let } fiddle \ (t: \text{tree}) = \text{insert } t \ 21
\]

If \( t \) is dead (unreachable), then all these nodes will be reclaimed!
Net space allocated

The garbage collector reclaims unreachable data structures on the heap.

```ml
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?

```plaintext
let faddle (t: tree) = (t, 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"

allocates nothing when arg is Some i

allocates an option when arg is Some i
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 cadd (c1:int*int) (c2:int*int) : int*int =
  let (x1,y1) = c1 in
  let (x2,y2) = c2 in
  (x1+x2, y1+y2)

define the cadd function to extract components: it is a read

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

double does not allocate
FUNCTION CLOSURES
Closures (A reminder)

Nested functions like bar often contain free variables:

Here's bar on its own:

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

y is free in the definition of bar
But what about nested, higher-order functions?

bar again:

\[
\text{let bar } x = x + y
\]

bar's representation:

\[
\text{let f2 (n, env) = n + env.y}
\]

\{
\text{y = 1}
\}

environment

code

closure
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 = two 4-byte values = 8 bytes total +
- the cost of the environment (4 bytes in this case).

```plaintext
let f2 (n, env) = n + env.y
{y = 1}
```
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 (pair + environment)