Garbage Collection
Terminology

• Heap – a finite pool of data cells, can be organized in many ways

• Roots - Pointers from the program into the Heap.
  – We must keep track of these.
  – All pointers from global variables
  – All pointers from temporarys (often on the stack)

• Marking – Tracing the live data, starting at the roots. Leave behind a “mark” when we have visited a cell.
Things to keep in mind

- Costs – How much does it cost as function of
  - All data
  - Just the live data
- Overhead – Garbage collection is run when we have little or no space. What space does it require to run the collector?
- Complexity – How can we tell we are doing the right thing?
Structure of the Heap

Things to note in a Mark and sweep collector
- The Freelist
- The Roots
- Links from function closures
- Links from data (like pair or list)
- Constants
Structure of the Heap

(fun f (x) (+ x 7))
(val x 45)
(val y
  (let (val x 6)
    (val y 2)
    in (\ (a)
      (+ x (+ y a))))))
(val z (pair 7 'c'))
Changes in the heap

- Intermediate result computation
  - (@ f (fst z))
- Assignment to things
- Garbage collection
Changes in the heap

- Intermediate result computation
- Assignment to things
  - \((:= y (\text{pair 7 'c'})\))
- Garbage collection
Garbage Collection

FreeList

1.0

f

12, f 12

2.0

x

11, 45

10, 6

y

8, pair

4, a

3, 44

9, 2

z

5, pair

7, 7

6, c

10, 6

5, pair

7, 7

6, c

12, f 12

11, 45

4, a

3, 44
Mark and Sweep

• Cells have room for several things beside data

```haskell
data HCell a = Cell { mark :: (IORef Bool),
    key :: Int,
    payload :: IORef a,
    allocLink :: IORef (HCell a),
    allLink :: HCell a }
| NullCell
```

• All cells start linked together on the free list
• Allocation takes 1 (or more cells) from the free list
• Garbage collection has two phases
  – Mark (trace all live data from the roots)
  – Sweep (visit every cell, and add unmarked cells to free list)
Mark phase (turns cells red in this picture).
Where do links into the heap reside?

- **In the environment**

  \[
  \text{interpE} :: \text{Env (Range Value)} \rightarrow \text{State} \rightarrow \text{Exp} \rightarrow \text{IO(Value,State)}
  \]

  -- the variables in scope
  -- the heap
  -- exp to interpret

- **Inside data values**

  \[
  \text{data Value} = \text{IntV Int} \\
  | \text{CharV Char} \\
  | \text{ConV String Int (Range Value)} \\
  | \text{FunV Vname (Env (Range Value)) [Vname] Exp}
  \]
Mark a cell

markCell markV NullCell = return NullCell
markCell markV (cell@(Cell m id p l1 l2)) =
    do { b <- readIORef m; help b }
    where help True = return cell
           help False =
               do { writeIORef m True
                    ; v <- readIORef p
                    ; v2 <- markV
                          (markRange markV) v
                    ; writeIORef p v2
                    ; return cell}
Sweeping through memory

sweep (H all free) NullCell = return (H all free)
sweep (H all free) (c@(Cell m id p l more)) =
    do { b <- readIORef m
          ; if b then do { writeIORef m False
                            ; sweep (H all free) more }
          else do { -- link it on the free
                        writeIORef l free
                            ; sweep (H all c) more }
Mark phase (turns cells red in this picture).
Two space collector

• The heap is divided into two equal size regions
• We allocate in the “active” region until no more space is left.
• We trace the roots, creating an internal linked list of just the live data.
• As we trace we compute where the cell will live in the new heap.
• We forward all pointers to point in the new inactive region.
• Flip the active and inactive regions
A heap Cell

data HCell a =
  Cell { mark :: Mutable Bool
         , payload :: Mutable a
         , forward :: Mutable Addr
         , heaplink::: Mutable Addr
         , showR::: a -> String }
The Heap

data Heap a =
  Heap
  { heapsize :: Int,
    nextActive :: Addr,
    active :: (Array Int (HCell a)),
    inactive :: (Array Int (HCell a)),
    nextInActive :: Mutable Addr,
    liveLink :: Mutable Addr }
(val tim (+ 1 2))
(fun h (x) (+ x tim))

(fun map (f xs) (if (ispair xs)
    (pair (@ f (fst xs))
        (@ map f (snd xs)))
    xs))

(fun plus1 (x) (+ x 1))

(val g (@map plus1))

(val ans (@g (pair 1 (pair 2 (pair 3 0)))) )

in

ans { should yield (2.(3.(4.0))) }
(val tim (+ 1 2))
(fun h (x) (+ x tim))

(fun map (f xs) (if (ispair xs)
    (pair (@ f (fst xs))
        (@ map f (snd xs)))
    xs))

(fun plus1 (x) (+ x 1))

(val g (@map plus1))

(val ans (@g (pair 1 (pair 2 (pair 3 0)))))

in

ans { should yield (2.(3.(4.0))) }
markAddr :: (GCRecord a) -> Addr -> IO Addr
markAddr (rec@(GCRec heap markpay showV )) index = mark cell
  where cell = active heap ! index
  nextFreeInNewHeap = nextInActive heap
  markedList = liveLink heap
mark (Cell m payld forward reachable showr) =
  do { mark <- readIORef m
       ; if mark
           then do readIORef forward
                   else do {
                   -- Set up recursive marking
                   ; new <- fetchAndIncrement nextFreeInNewHeap
                   ; next <- readIORef markedList
                   ; writeIORef markedList index

                   -- Update the fields of the cell, showing it is marked
                   ; writeIORef m True
                   ; writeIORef forward new
                   ; writeIORef reachable next

                   -- recursively mark the payload
                   ; v <- readIORef payld
                   ; v2 <- markpay (markRange rec) v

                   -- copy payload in the inactive Heap with
                   -- all payload pointers relocated .
                   ; writeIORef (payload ((inactive heap) ! new)) v2
                   -- finally return the Addr where this cell will be relocated to.
                   ; return new }}