

Garbage Collection

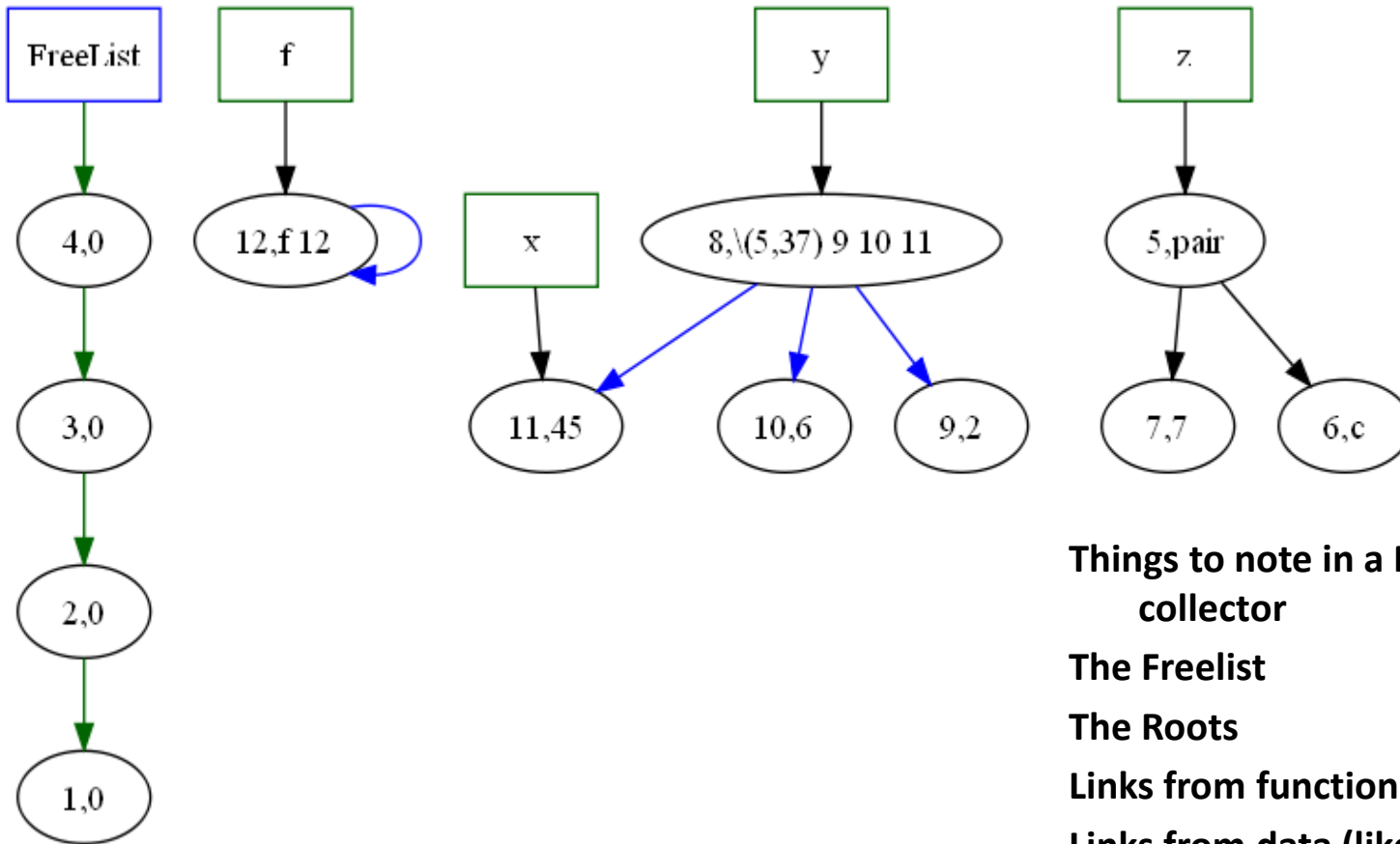
Terminology

- 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 temporaries (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

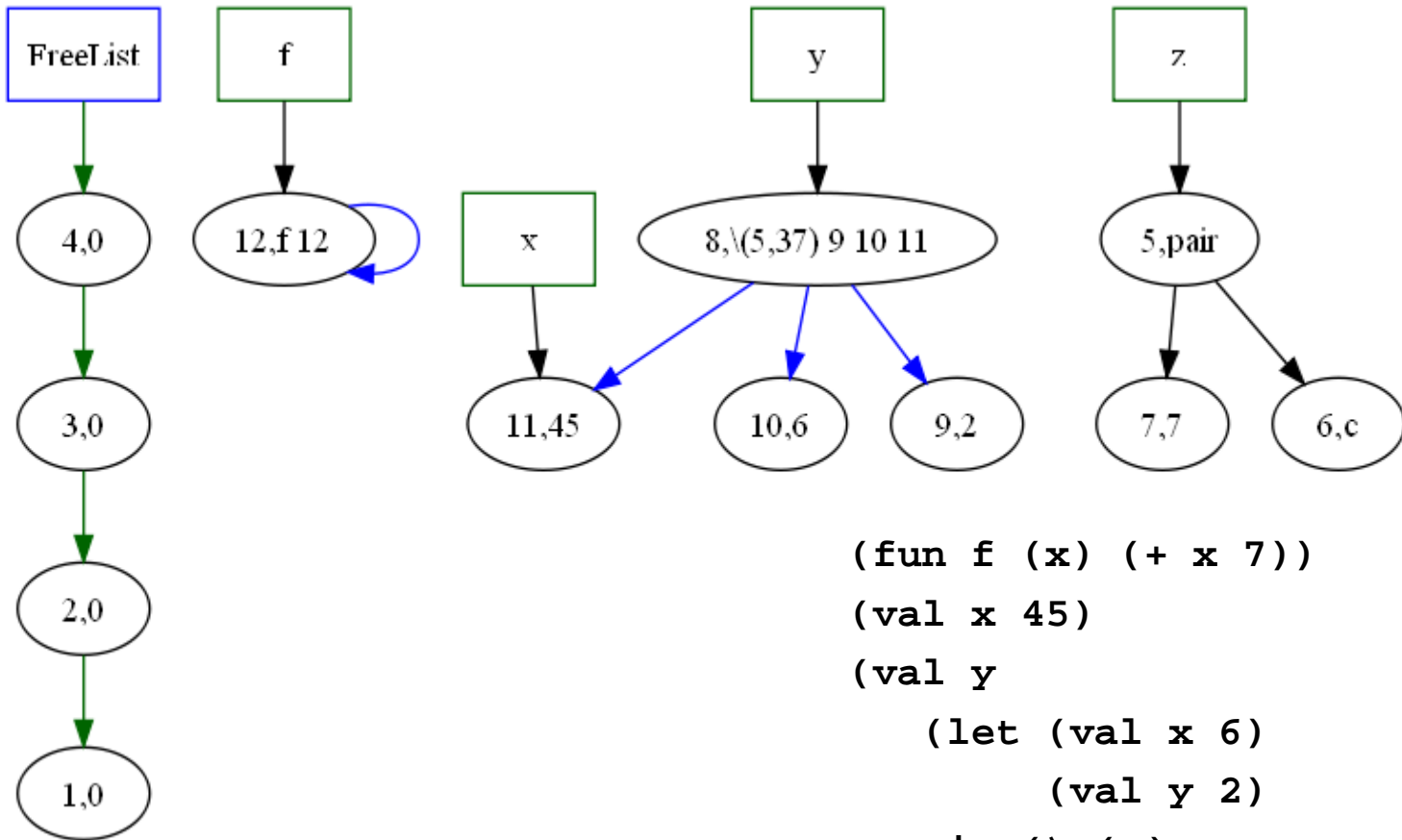
The Roots

Links from function closures

Links from data (like pair or list)

Constants

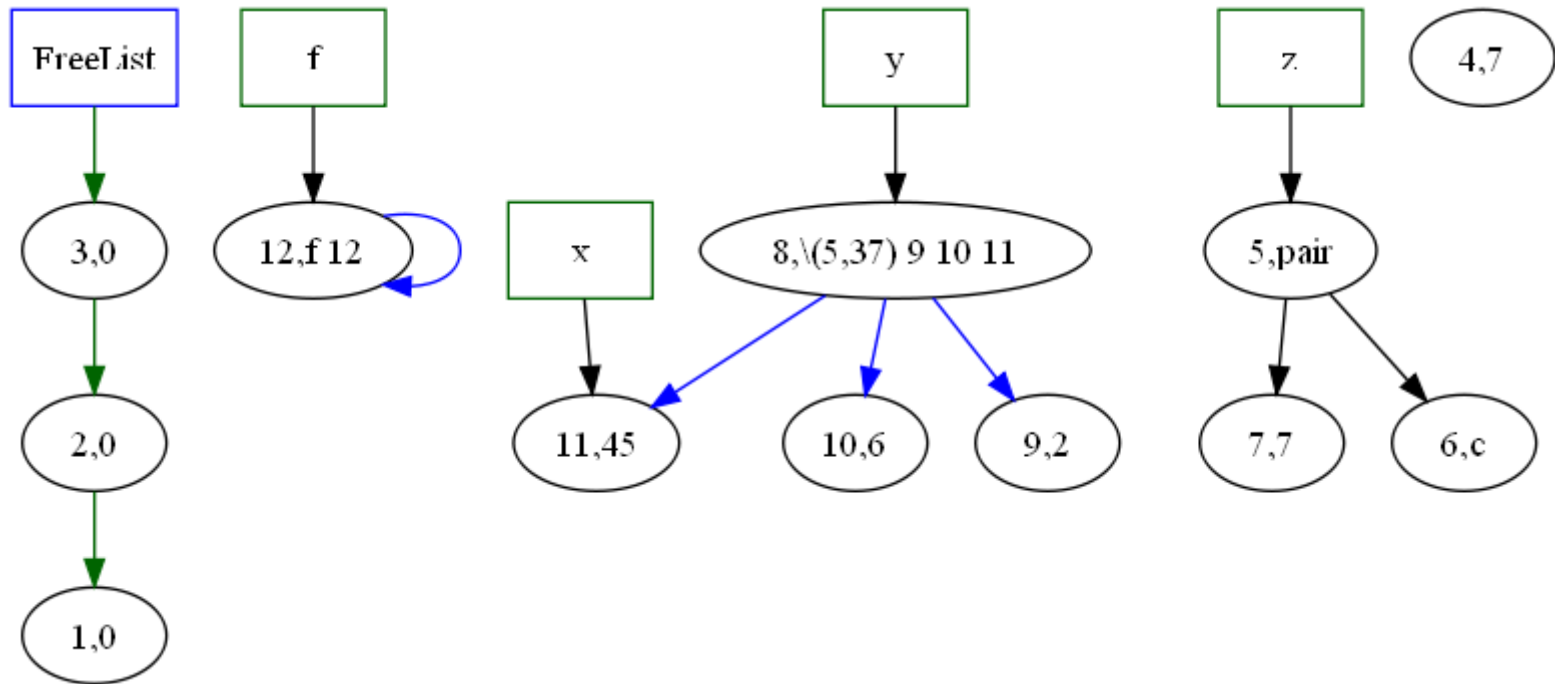
Structure of the Heap



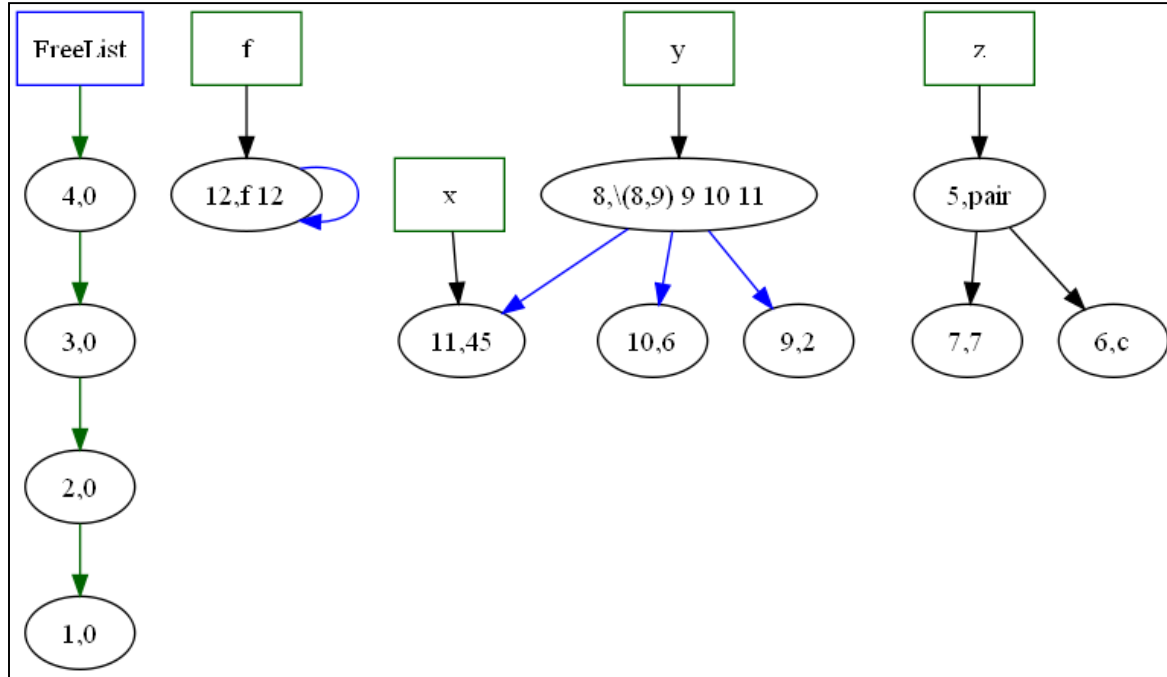
```
(fun f (x) (+ x 7))
(val x 45)
(val y
  (let (val x 6)
        (val y 2)
    in (\ (a)
        (+ x (+ y a))))))
(val z (pair 7 'c'))
```

Changes in the heap

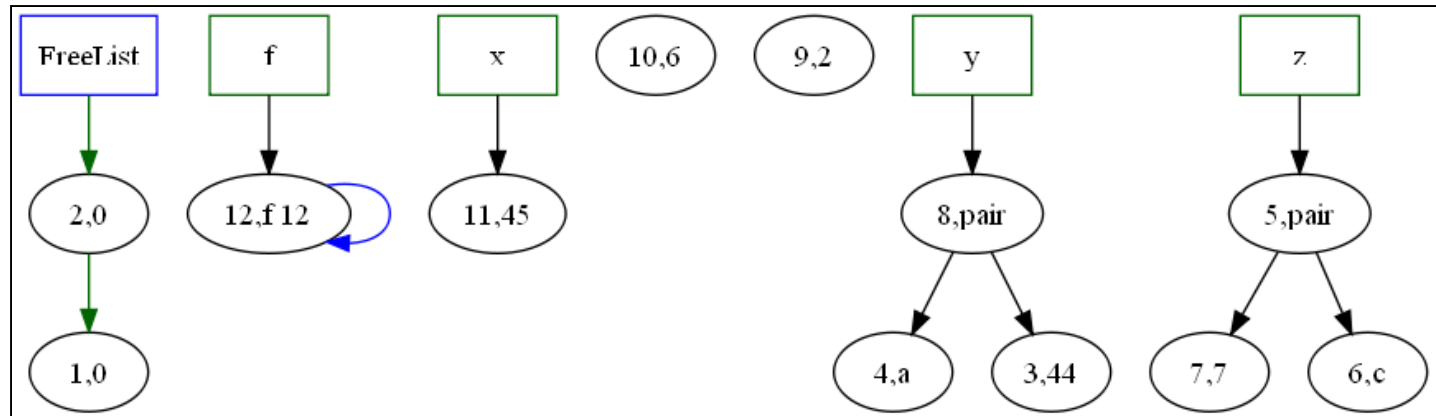
- Intermediate result computation
 - (@ f (fst z))
- Assignment to things
- Garbage collection



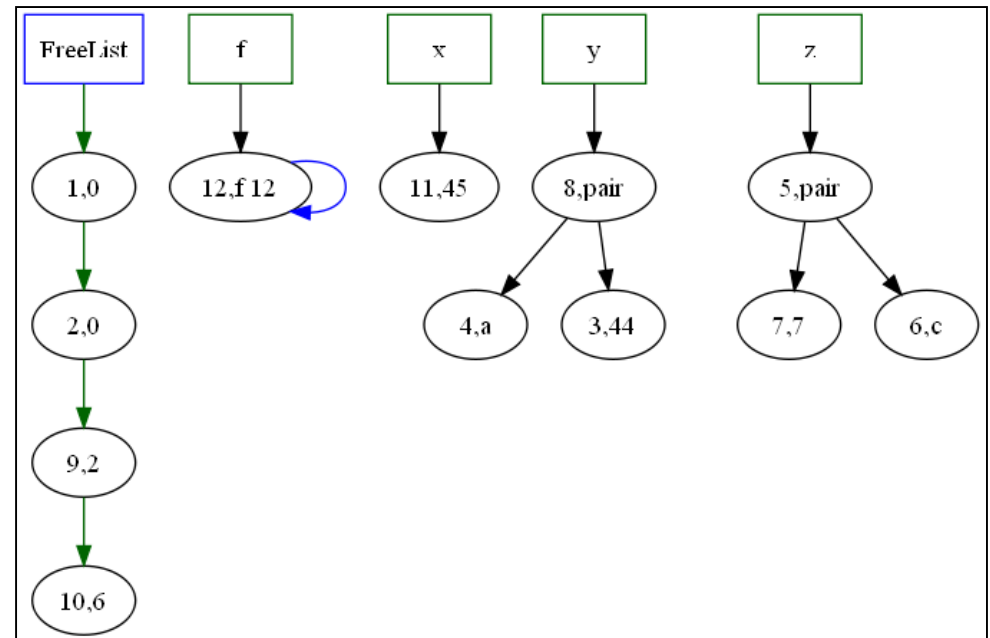
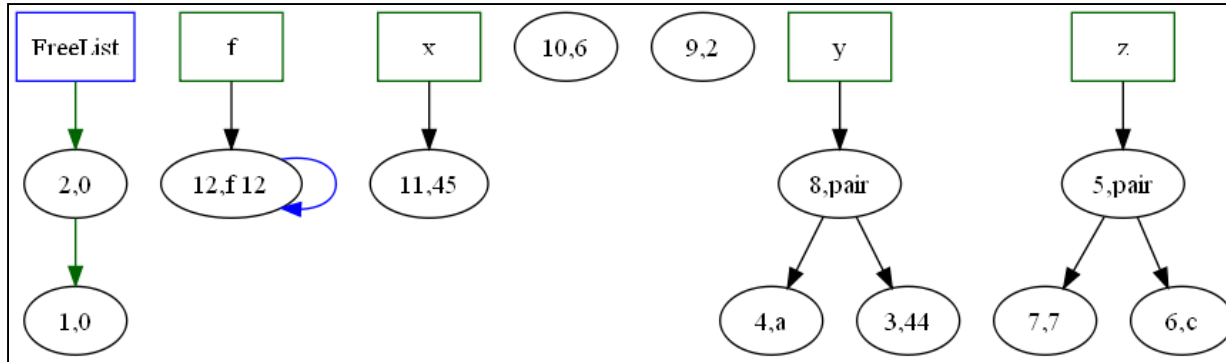
Changes in the heap



- Intermediate result computation
- Assignment to things
 - **(:= y (pair 44 'a'))**
- Garbage collection



Garbage Collection



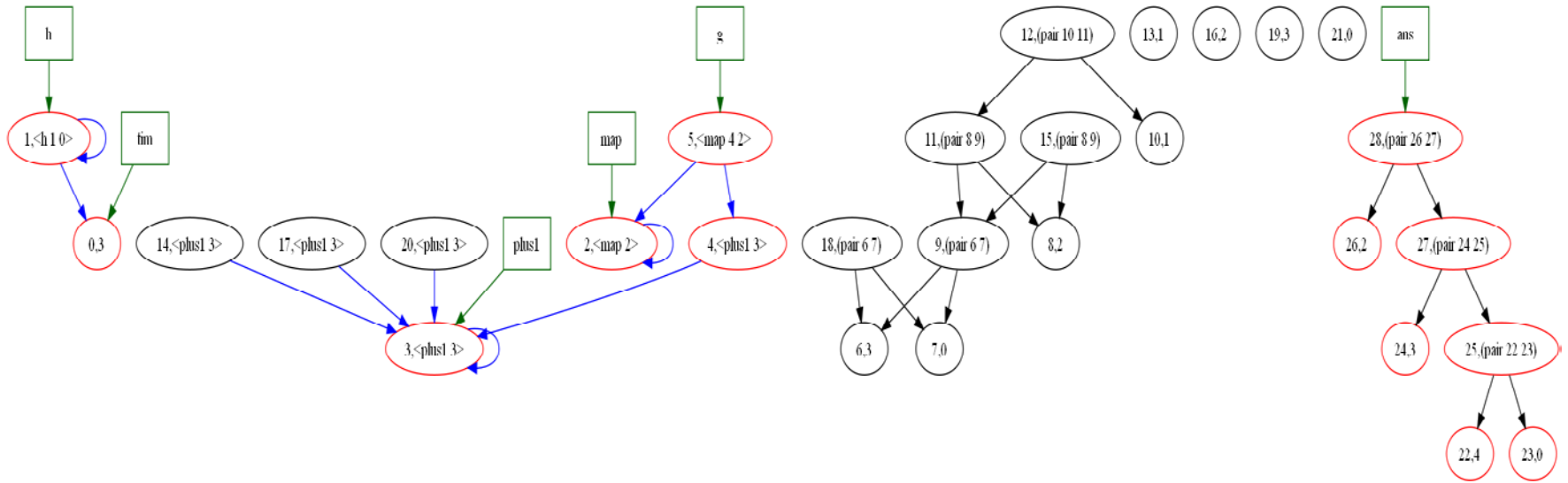
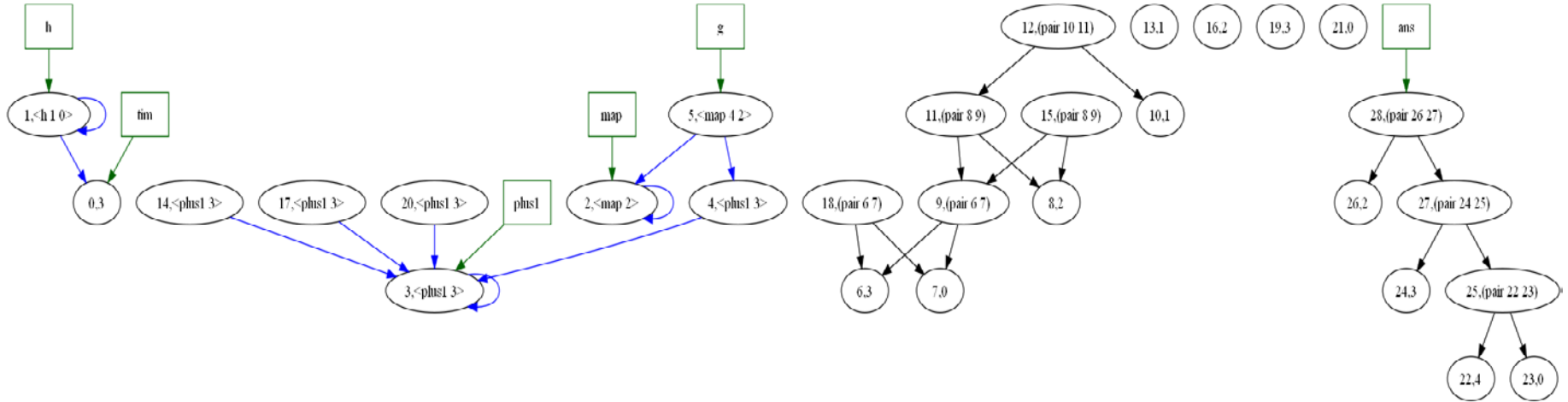
Mark and Sweep

- Cells have room for several things beside data

```
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

```
interpE :: Env (Range Value)      -- the variables in scope
         -> State                  -- the heap
         -> Exp                    -- exp to interpret
         -> IO(Value,State)
```

- Inside data values

```
data Value
  = IntV Int
  | CharV Char
  | ConV String Int (Range Value)
  | FunV Vname (Env (Range Value)) [Vname] Exp
```

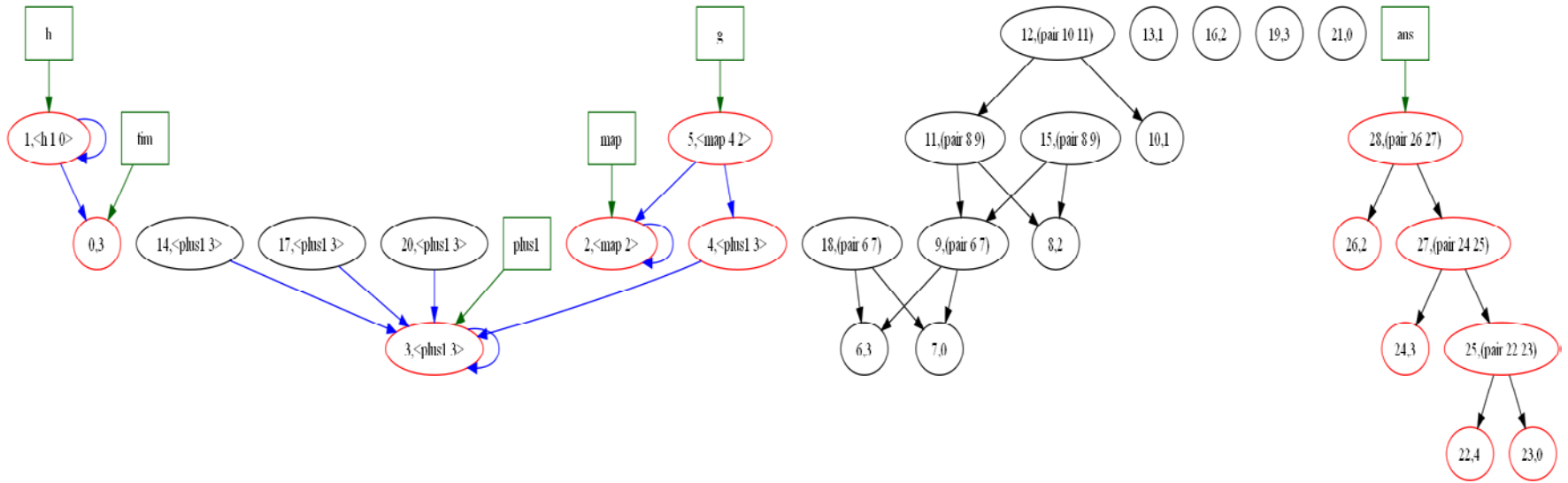
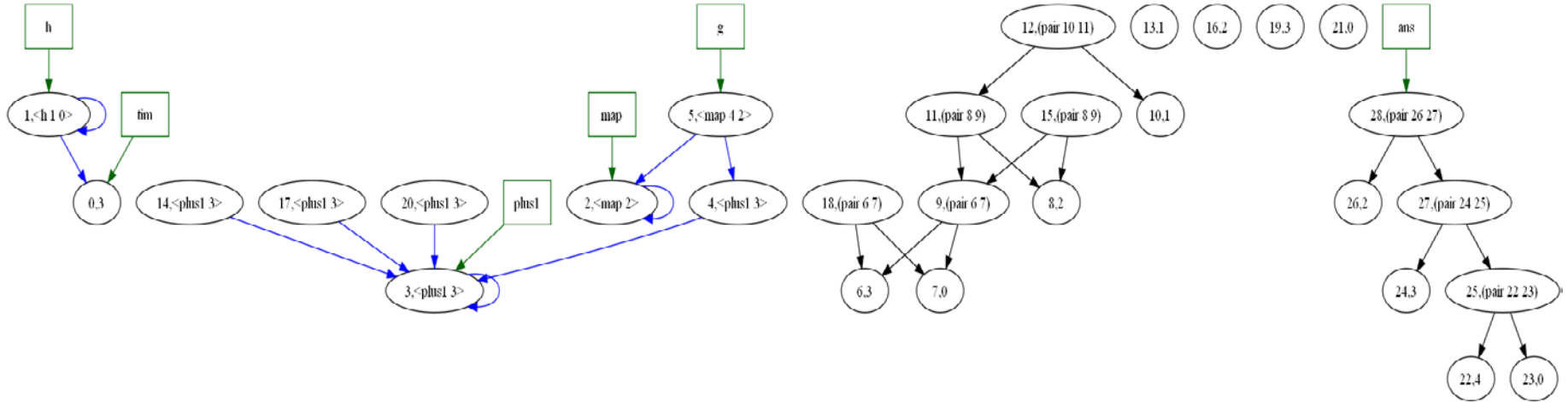
Mark a cell

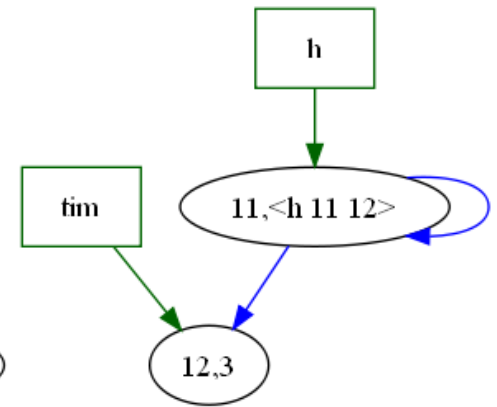
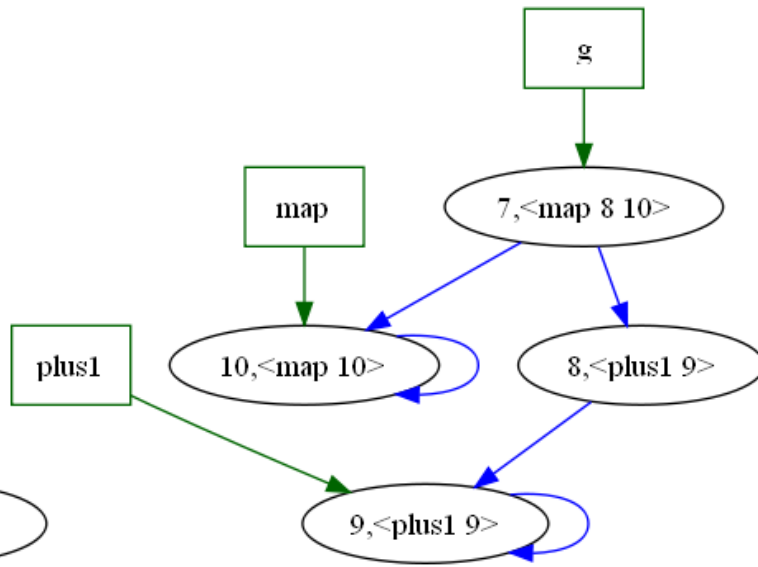
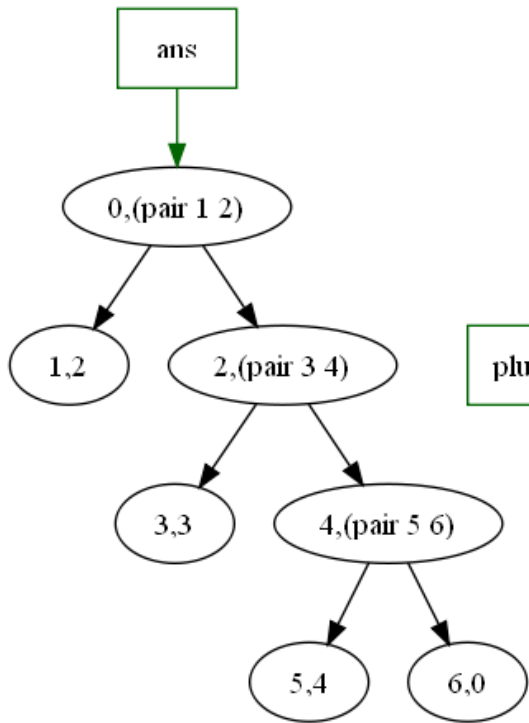
```
markCell markV NullCell = return NullCell
markCell markV (cell@(Cell m id p l1 l2)) =
    do { b <- readIORef m; help b }
where help True = return cell
      help False =
        do { writeIORef m True
            ; v <- readIORef p
            ; v2 <- markV
              (markRange markV) v
   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

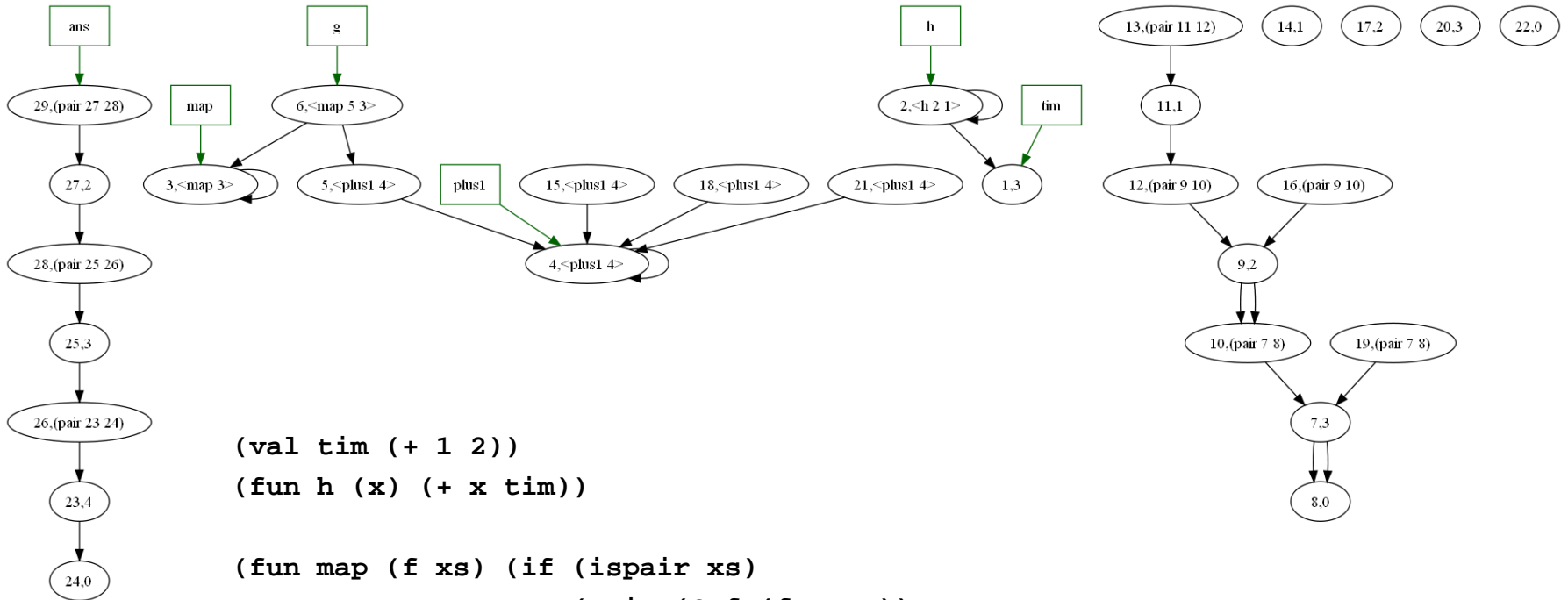
(fun plus1 (x) (+ x 1))

(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))) }

```

```

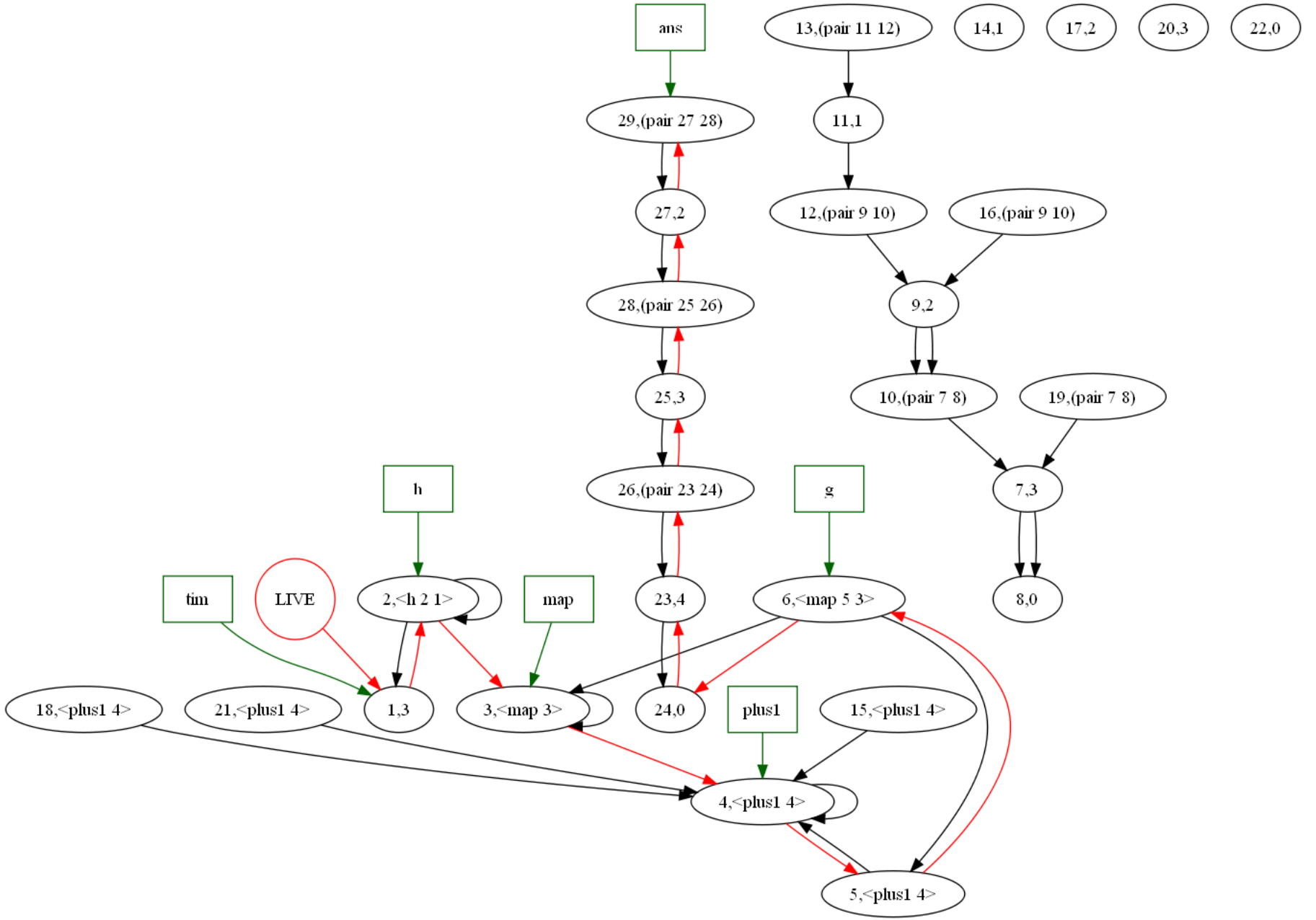
markAddr :: (GCRecord a) -> Addr -> IO Addr
markAddr (rec@(GCRec heap markpay showV )) index = mark cell
  where cell = active heap ! index
        nextFreeInNewHeap = nextInActive heap
        markedList = liveLink heap
mark (Cell m payld forward reachable showr) =
  do { mark <- readIORef m
      ;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

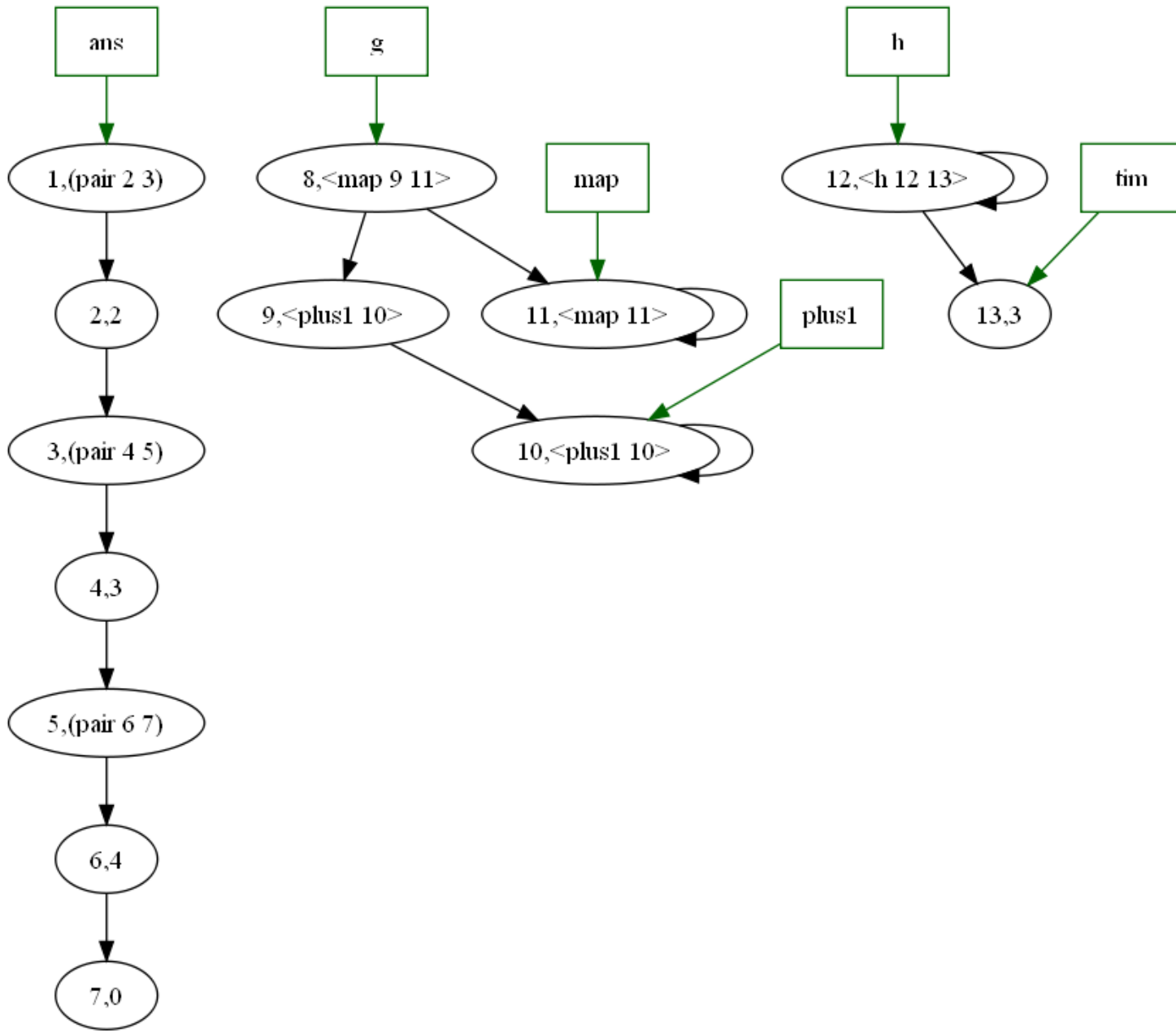
-- Update the fields of the cell, showing it is marked
; writeIORef m True
; writeIORef forward new
; writeIORef reachable next

-- 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 }}

```





Kinds of collectors

- Mark and sweep
- Two space collectors
- Relocating collectors
- Reference counting collectors
- Generational collectors

Reference counting collectors

- Every cell contains a reference count.
- It is incremented whenever a new pointer is added to a cell, and decremented whenever a pointer is changed from pointing at the cell to some other cell.
- Cells whose reference counts drop to zero are garbage and are reclaimed.

Reference Counting

- Advantages

- Simple
- Garbage is collected incrementally when it becomes free

- Disadvantages

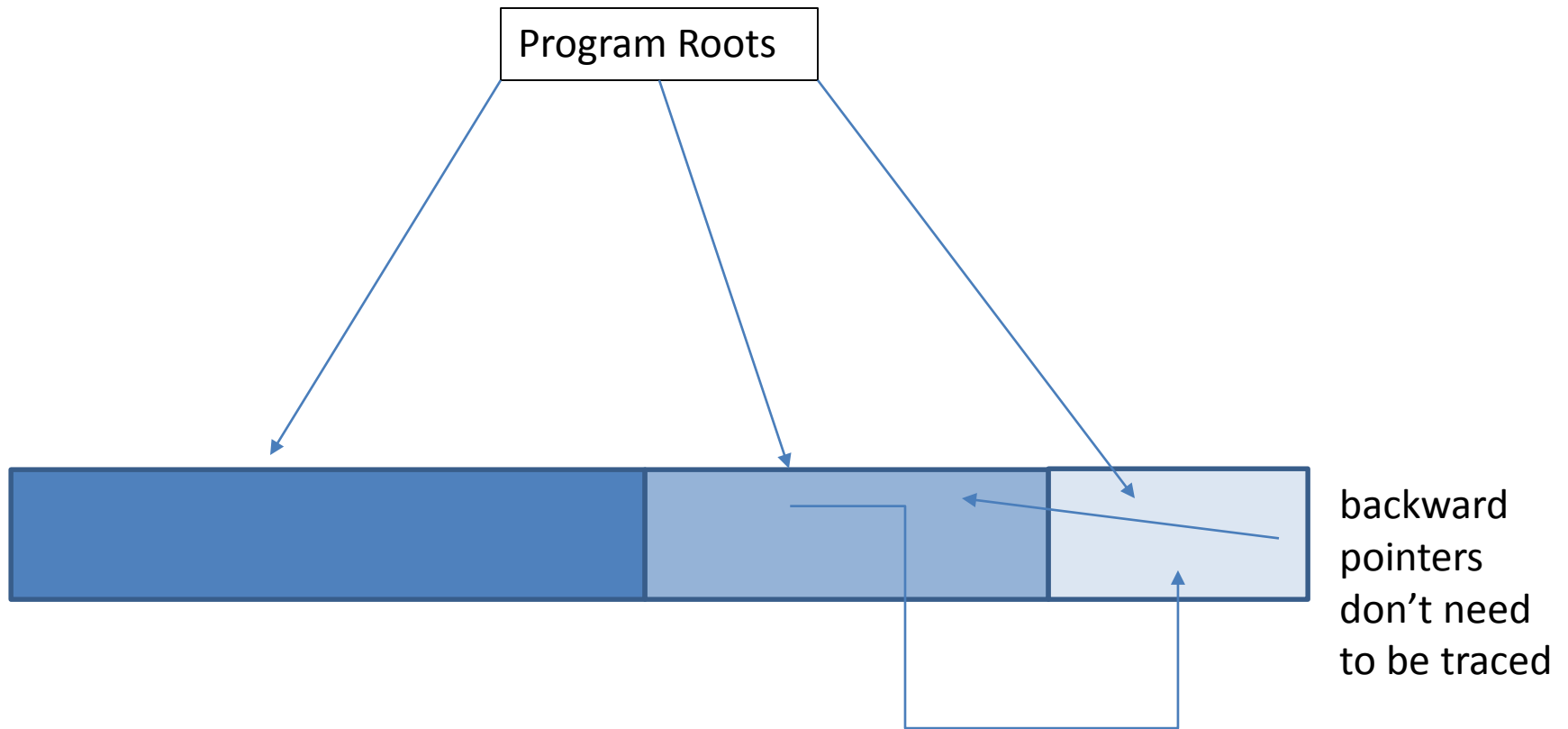
- Circular structures are never collected
- No upper bound on performing a pointer operation. (A cell may become free, and then all the cells it points to must be decremented, and they may become free)
- Live cells become fragmented in memory (little spatial locality)

Generational Collectors

- Assumptions
 - Most newly allocated cells become garbage quickly
 - Cells that survive 1 or 2 collections tend to be long lived
 - Old cells seldom (if ever) point to newer cells.
 - No need to spend time tracing pointers to old cells as one can assume that they are still reachable

Strategy

- Divide memory into (different sized) regions.
- Each region holds cells of approximately the same age.
- Allocate cells in the newest region (usually relatively small, often called the nursery)
- When space in the newest region runs out, collect cells in only that region
 - Only trace the roots into the newest region
 - Assume everything in older regions is reachable
 - Special code to handle pointers from old regions to newer regions
 - Collect reachable cells in the newest region and promote them to an older region.



Most garbage is in the newest region.
Only backward pointers within the collecting region need to be traced.
Forward pointers into the collecting region must be handled just like program roots

Generational

Advantages

- Small collection times
 - Tracing only the live data
 - In only a (relatively small) region
 - No need to touch unreachable cells
 - Compacts live cells for better special locality

Disadvantages

- Code can be complex
- Many forward pointers can wreck the otherwise good performance

Other issues

- Concurrent Collection
 - Separate processes
 - Race conditions
 - Approximate collection can be liveable
- Finalization
 - When an object becomes garbage, it may free up other cells. Sometimes this can be done automatically, but other time specialized knowledge is needed. Finalizers allow programmers to add this knowledge

Overall Advantages of Garbage collectors

- Relieves programmers of an error prone task
- Removes dangling pointers
- Stops memory leaks
- Avoids “double frees”
 - Freeing a cell already freed (and subsequently re-allocated)
- Efficient implementation of “persistent” data structures
 - immutable data structures, can keep around old versions in case they are needed

Overall Disadvantages

- Stop-the-world mentality
 - At any time the system may pause for GC
- Timing and duration of GC times is unpredictable
- Unpredictable performance of the same code on the same data

Comparisons

- Mark and trace is simple but can have long pauses. Tracing times are proportional to memory size
- Reference Count systems are simple but can't deal with cyclic structures, and all pointer manipulation operations can have unbounded upper limits
- Two space collectors have times proportional to live memory. Can compact memory for better spatial locality, but use twice as much space.

Comparisons continued

- Generational collectors can have the smallest pause times (proportional to live memory that is traced). Compact memory. With small nurserys don't have as much spatial overhead as two space collectors.
- Concurrent collectors. Run continuously. Code complicated to deal correctly with race conditions. Safe to leave some garbage uncollected if one can collect it later (when race conditions no longer apply).