

Final step:

- once there are no SimpWL, MoveWL, FreezeWL, SpillWL, it is time to color the nodes
- Pop nodes from the Stack in order, assigning colors
- For each node: the possible colors are colors not already given to the neighbors
- If there is a color, pick a color, any color
- Otherwise, add a *real* spill
- Once finished, if there are any real spills, start over



Assign_colors

```
let rec assign_colors ra =
  node_iter ra NodeStack (fun node ->
    let remove_color colors w = ... in
    match List.fold_left remove_color
      Backend.registers node.node_neighbors
    with
    color :: _ ->
      node.node_color <- Some color;
      node_reclassify ra node NodeColored
    | [] ->
      node_reclassify ra node NodeSpilled)
```



Remove color

- Start with the full set of registers
- Remove the color of each neighboring node

```
let remove_color colors w =
  let w' = node_alias w in
  match w'.node_class with
  NodeColored
  | NodePrecolored ->
    (match w'.node_color with
     Some color ->
       List_util.remove color colors
     | None ->
       raise (Invalid_argument "assign_colors"))
  | _ ->
    colors
```



Spill color assignment

- Once all the colors have been assigned, the Spilled nodes have to be spilled
- The easy way:
 - Just call (*Backend.spset_spill spset v*) for any node that was spilled
 - This will assign *v* to a new spill register *i(%ebp)*, in sequence *0(%ebp)*, *4(%ebp)*, *8(%ebp)*, ...



A better spill method

- Treat spills as an alternative register set, and spill the same way as `assign_colors`
- Start with an empty register file
 - Make up new colors as you go along



Assigning spill colors

```
(* Generate new colors *)
let colors =
  node_fold ra NodeSpilled (fun colors node ->
    let remove_color colors w = ... in
    match List.fold_left remove_color
      colors node.node_neighbors with
    color :: _ ->
      node.node_color <- Some color;
      colors
    | [] ->
      let color = new_symbol_string "spill" in
      node.node_color <- Some color;
      color :: colors) []
in
```



Choosing spill colors

```
(* Generate new colors *)
let remove_color colors w =
  let w' = node_alias w in
  match w'.node_class with
  | NodeSpilled ->
    (match w'.node_color with
     | Some color ->
       List_util.remove color colors
     | None ->
       colors)
  | _ ->
    (* Others do not interfere *)
    colors
```

in



Spilling spill colors

```
(* Spill all the new colors *)
let spset = List.fold_left Backend.spset_spill spset colors in
```

```
(* Assign all the spills *)
node_fold ra NodeSpilled (fun spset node ->
  let v = var_of_node ra node in
  match node.node_color with
  | Some v' ->
    Backend.spset_add spset v v'
  | None ->
    raise (Invalid_argument ...))
```



Garbage

- What is garbage?
 - Any object that can be removed from the heap without affecting the results of the computation
- Semantic garbage
 - Optimal collection is not computable
 - Usually, garbage is computed syntactically
 - Type inference can be used to collect some objects that are semantically garbage



Programming languages

- We'll simplify and use functional languages for now
- Lambda-calculus:
 - Variables: x, y, z
 - Functions: $(\text{fun } x \rightarrow e)$ or $(\text{lambda } x. e)$
 - Function application: $e_1(e_2)$
- Evaluation is by substitution
 - $(\text{lambda } x. e_1) e_2 \rightarrow e_1[e_2 / x]$



Programs: GC

Programs :

(variables)	$x, y, z \in \text{Var}$	
(integers)	$i \in \text{Int}$	$::= \dots, -2, -1, 0, 1, 2, \dots$
(expressions)	$e \in \text{Exp}$	$::= x \mid i \mid (e_1, e_2) \mid \pi_i e \mid \lambda x. e \mid e_1 e_2$
(heap values)	$h \in \text{Hval}$	$::= i \mid (x_1, x_2) \mid \lambda x. e$
(heaps)	$H \in \text{Heap}$	$::= \{x_1 = h_1; \dots; x_n = h_n\}$
(programs)	$P \in \text{Prog}$	$::= \text{letrec } H \text{ in } e$
(answers)	$A \in \text{Ans}$	$::= \text{letrec } H \text{ in } x$

Contexts :

(contexts)	$E \in \text{Ctx}$	$::= [] \mid (E, e) \mid (x, E) \mid \pi_i E \mid E e \mid x E$
(instructions)	$I \in \text{Inst}$	$::= h \mid \pi_i x \mid x y$



Free variables

$$\begin{aligned} FV(i) &= \{\} \\ FV(x) &= \{x\} \\ FV((e_1, e_2)) &= FV(e_1) \cup FV(e_2) \\ FV(\pi_i e) &= FV(e) \\ FV(\lambda x. e) &= FV(e) - \{x\} \\ FV(e_1 e_2) &= FV(e_1) \cup FV(e_2) \end{aligned}$$

$$\begin{aligned} FV(\{x_1 = h_1; \dots; x_n = h_n\}) &= (\bigcup_{i=1}^n FV(h_i)) - \{x_1, \dots, x_n\} \\ FV(\text{letrec } H \text{ in } e) &= (FV(H) \cup FV(e)) - \text{Dom}(H) \end{aligned}$$



Evaluating programs

- Operational semantics (Plotkin)
- Specify a *relation* that corresponds to evaluation

(alloc) $\text{letrec } H \text{ in } E[h] \xrightarrow{\text{alloc}} \text{letrec } H \cup \{x = h\} \text{ in } E[x]$
(proj) $\text{letrec } H \text{ in } E[\pi_i; x] \xrightarrow{\text{proj}} \text{letrec } H \text{ in } E[x_i] \quad (H(x) = \langle x_1, x_2 \rangle)$
(app) $\text{letrec } H \text{ in } E[x \ y] \xrightarrow{\text{app}} \text{letrec } H \cup \{z = H(y)\} \text{ in } E[e] \quad (H(x) = \lambda z. e)$



Evaluation example

$\text{letrec } \{x = 1\} \text{ in } (\lambda y. \pi_1 \ y) \langle x, x \rangle \xrightarrow{\text{alloc}} \text{letrec } \{x = 1; z = \lambda y. \pi_1 \ y\} \text{ in } z \langle x, x \rangle$
 $\xrightarrow{\text{alloc}} \text{letrec } \{x = 1; z = \lambda y. \pi_1 \ y; w = \langle x, x \rangle\} \text{ in } z \ w$
 $\xrightarrow{\text{app}} \text{letrec } \{x = 1; z = \lambda y. \pi_1 \ y; w = \langle x, x \rangle; y = \langle x, x \rangle\} \text{ in } \pi_1 \ y$
 $\xrightarrow{\text{app}} \text{letrec } \{x = 1; z = \lambda y. \pi_1 \ y; w = \langle x, x \rangle; y = \langle x, x \rangle\} \text{ in } x$



Definitions

- $P \xrightarrow{G} P'$ means P rewrites to P' using one of the rules in G
- $P \xrightarrow{G}^* P'$ is the transitive closure
- $G + r$ is the union of G with the rule r
- A program P is *canonical* if there is no P' where $P \xrightarrow{G} P'$
- $P \Downarrow_G P'$ means $P \xrightarrow{G}^* P'$ and P' is canonical
- $P \Uparrow_G$ means P diverges



Stuck

- A program is *stuck* if it is not a value, and it can't be evaluated further
- (A type system can be used to guarantee that this won't happen)
- **letrec** H **in** $E[\pi_i x]$ and $x \notin \text{Dom}(H) \vee x \neq \langle x_1, x_2 \rangle$
- **letrec** H **in** $E[x y]$ and $x \notin \text{Dom}(H) \vee x \neq \lambda z.e$



Program equivalence

$(P_1, G_1) \approx (P_2, G_2)$ means both programs diverge or they evaluate to the same value

- $P_1 \Downarrow_{G_1}$ **letrec** H_1 **in** x and $H_1(x) = i$
- $P_2 \Downarrow_{G_2}$ **letrec** H_2 **in** y and $H_2(y) = i$

(if $G_1 = G_2 = R$ we write $P_1 \approx P_2$)



Semantic garbage

For a program $P = \text{letrec } H \cup \{x = h\} \text{ in } e$:

x is **garbage** if $P \approx \text{letrec } H \text{ in } e$

Garbage undecidable:

$P = \text{letrec } H \cup \{x_1 = 1, x_2 = 2, x = \langle x_1, x_2 \rangle\} \text{ in } (\lambda y. \pi_1 x) e$



Syntactic garbage

- Free-variable rule fv

$\text{letrec } H_1 \cup H_2 \text{ in } e \xrightarrow{fv} \text{letrec } H_1 \text{ in } e$

if $\text{Dom}(H_2) \cap \text{FV}(\text{letrec } H_1 \text{ in } e) = \{\}$



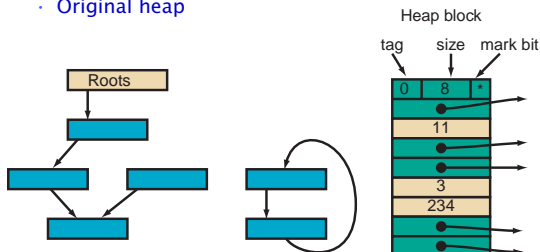
Mark-sweep

- Each program has a set of roots
 - Contents of the registers
 - Contents of the stack
- GC is performed in two phases
 - Mark: traverse the heap from the roots, marking each block encountered
 - Sweep: traverse the heap linearly, removing any block that is not marked (and clearing mark bits)



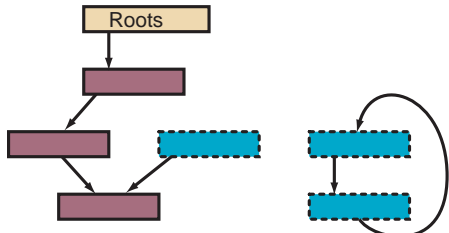
Mark-sweep

- Original heap



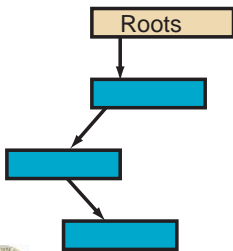
Mark phase

- Traverse the heap, marking all reachable blocks



Sweep phase

- Scan the heap, removing unmarked blocks



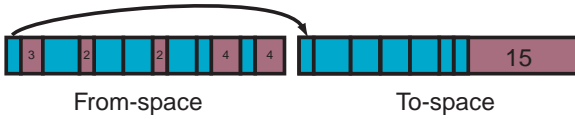
Cost of mark-sweep

- Mark: linear in number of *live* blocks
- Sweep: linear in total size of the heap
- Allocation
 - *First-fit, best-fit, etc*
 - *Fragmentation*



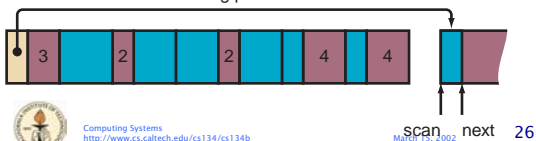
Copying collection

- Split heap into two parts:
 - From-space: *the currently used space*
 - To-space: *destination space in a collection*



Copying

- *scan*: a pointer to the next block in to-space to be scanned
- *next*: the next free location in to-space
 - *while scan < next do*
 - for each pointer *p* in the block
 - if *p* has been forwarded, update the pointer
 - otherwise, copy the block and leave a forwarding pointer
 - move scanning the next block in to-space



Copying

- Cost is linear in the amount of *live data*
- Allocation (constant time)
 - *next*: *next free location in from-space*
 - *limit*: *bounds of from-space*
 - let *alloc size* =
 - let *n* = *next* in
 - if *n* + *size* <= *limit* then
 - *next* := *n* + *size*;
 - return *n*
 - else
 - *gc* (); *alloc size*
- But only get to use *half* the heap!



Formal definition

- H_f is the *from*-space
- H_t is the *to*-space
- S is the *scan*-set

1. Pick a variable x in S bound in H_f
2. Scan the block h , looking for free variables
3. For each free variable $y \in FV(h)$, add y to S if it is not bound in H_t

$$\langle H_f \cup \{x = h\}, S \cup \{x\}, H_t \rangle \Rightarrow \langle H_f, S \cup (FV(h) - (Dom(H_t) \cup \{x\})), H_t \cup \{x = h\} \rangle$$



Collection

- Stop when there is nothing left to copy

letrec H in $e \xrightarrow{fva}$ **letrec** H' in e

$$\langle H, FV(e), \{\} \rangle \Rightarrow^* \langle H', S, H' \rangle$$

and $Dom(H'') \cap S = \{\}$



Next topics

- Generational collection
- Using type-inference to collect more garbage
- Tag-free garbage collection



Outline

- Generational garbage collection
- Using type inference
- Tag-free garbage collection



Cost of collection

- Suppose there are R words of reachable data
- Heap size is H
- Mark/sweep:
 - $c_1 R + c_2 H$ (for some constants c_1, c_2 , typically 10 and 3 instructions)
 - For each word allocated $\frac{c_1 R + c_2 H}{H - R}$
 - If H is large, cost is roughly c_2 (3 instructions per word allocated)



Cost of collection

- Copying collection takes time proportional to number of nodes marked
- Total cost of a collection is $c_3 R$ for some constant c_3 (typically 10 instructions)
- The amortized cost is then

$$\frac{c_3 R}{\frac{H}{2} - R}$$

- Note that there is no *inherent lower bound to the cost of garbage collection*

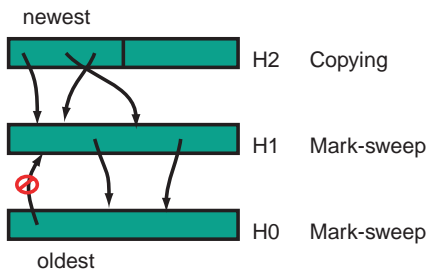


Generational collection

- Divide the heap into partitions (generations)
- Allocate from *newest* generation
- During GC, copy data to older heaps
- Invariant: no pointers from older to newer generations



Generational collection



Generational collection

A generational partition is a sequence of heaps H_1, H_2, \dots, H_n
 $H = H_1 \cup H_2 \cup \dots \cup H_n$
 $FV(H_i) \cap \text{Dom}(H_{i+1} \cup \dots \cup H_n) = \{\}$



Generational theorem

- Let $H_1, \dots, H_i, \dots, H_n$ be a generational partition of the heap $P = \text{letrec } H \text{ in } e$
- Suppose $H_i = H_i^1 \uplus H_i^2$
- $\text{Dom}(H_i^2) \cap \text{FV}(\text{letrec } H_i^1 \uplus \dots \uplus H_n \text{ in } e) = \{\}$
- Then $P \xrightarrow{\text{fv}} \text{letrec}(H - H_i^2) \text{ in } e$



Important properties

- Evaluation of closed, functional programs preserves generational partitions
- Free-variable-based collection works
- Performance: "objects tend to die young"
 - Concentrate collection on younger generations
 - Each older generation should be exponentially larger than younger generations



Performance

- Typically, the youngest generation is 10% live
- Cost is $c_3 R (10R - R)$ or about 1 instruction



Handling assignment

- Invariant assumption is that there are no pointers from older generations to younger generations
- Functional programs always preserve the invariant
- Assignment breaks it



Assignment

(expressions) $e \in Exp ::= \dots \mid e_1 := e_2$
(contexts) $E \in Ctxt ::= \dots \mid E := e \mid v ::= E$
(instructions) $I \in Instr ::= \dots \mid x := y$

$\text{letrec } H \cup \{x = h, y = h'\} \text{ in } E[x := y]$
 $\xrightarrow{\text{set}} \text{letrec } H \cup \{h = h', y = h'\} \text{ in } E[x]$

- Assignment breaks the generational invariant

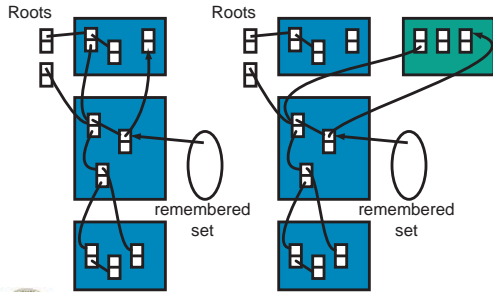


Assignment

- Handling pointers from old to young objects
- **Remembered list:** compiler generates code after each *store* to put the assigned pointer into a vector of *updated objects*. The GC treats the remembered list as a set of root pointers
- **Remembered set:** like a remembered list, but avoid adding an object multiple times by using a mark bit
- **Card marking:** divide memory into “cards” of size 2^k . Whenever an assignment happens to an object in card k , mark the card
- **Page marking:** use cards that are the same as the page size



Remembered lists



Garbage collection with type inference

- Consider

$\text{letrec}\{x_1 = 1, x_2 = 2, x_3 = \langle x_2, x_2 \rangle, x_4 = \langle x_1, x_3 \rangle\} \text{ in } \pi_1 x_4$

- Equivalent program

$\text{letrec}\{x_1 = 1, x_2 = 2, x_3 = 0, x_4 = \langle x_1, x_3 \rangle\} \text{ in } \pi_1 x_4$



Using type inference

- If an object in the heap can be given a polymorphic type, then that object is garbage
- Add types to the language

(types) $\tau \in \text{Type} ::= t \mid \text{int} \mid \tau_1 \times \tau_2 \mid \tau_1 \rightarrow \tau_2$



Typing rules for expressions

- Gamma is a type environment

$$\frac{}{\Gamma \cup \{x:\tau\} \vdash x : \tau} \quad \frac{\Gamma \cup \{x:\tau_1\} \vdash e : \tau_2}{\Gamma \vdash \lambda x.e : \tau_1 \rightarrow \tau_2}$$
$$\frac{}{\Gamma \vdash i : int} \quad \frac{\Gamma \vdash e : \tau_1 \times \tau_2}{\Gamma \vdash \pi_i e : \tau_i}$$
$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash (e_1, e_2) : \tau_1 \times \tau_2} \quad \frac{\Gamma \vdash e_1 : \tau_1 \rightarrow \tau_2 \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash e_1 e_2 : \tau_2}$$



Typing rules for heaps

$$\frac{\forall x \in Dom \Gamma'. \Gamma \cup \Gamma' \vdash H(x) : \Gamma'(x)}{\Gamma \vdash H : \Gamma'}$$
$$\frac{\vdash H : \Gamma \quad \Gamma \vdash e : \tau}{\vdash \text{letrec } H \text{ in } e : \tau}$$



Incremental collection

- Garbage collection is nice because the programmer doesn't have to worry about allocation
- Garbage collection is bad because a collection cycle may interrupt the process for a long time
- Incremental collection: try to interleave collection with mutation



Terminology

- The *collector* tries to collect garbage
- The *mutator* keeps changing the heap
- An *incremental* algorithm works only by explicit request
- A *concurrent* algorithm can operate at any point during or between instructions executed by the mutator



Tricolor marking

- **White objects are not yet visited by the collector**
- **Gray** object have been visited (marked or copied), but their children have not yet been examined
- **Black objects have been marked, and their children are also marked**



Incremental algorithm

- All objects are initially white
- When an object is changed from gray to black, it is removed from the queue of objects to be examined by the collector
- When an object is colored gray, it is placed in the queue of objects to be examined
- Invariants:
 - *No black object points to a white object*
 - *Every gray object is in the collector's queue*
- When there are no gray objects, all white objects are garbage



Maintaining the invariants

- Dijkstra: whenever the mutator stores a white pointer a into a black object b, a is colored gray
- Steele: whenever the mutator stores a white pointer a into a black object b, it colors b gray
- Boehm et.al. All-black pages are marked read-only. Whenever the mutator stores any value into a protected page, the fault handler colors the page gray
- Baker. Whenever the mutator read a pointer b to a white object, it colors b gray
- Appel, et.al. Whenever the mutator fetches a pointer b from a page containing any non-black object, the fault-handler colors every object black



Summary

- Incremental collection has a high overhead