Garbage Collection

ICS 280

Joachim Feise

jfeise@ics.uci.edu

What is Garbage Collection?

- automatic reclamation of computer storage
- objects not reachable via any pointer are considered garbage
- live objects are preserved
- Two phases:
  - garbage detection
  - reclaiming the storage

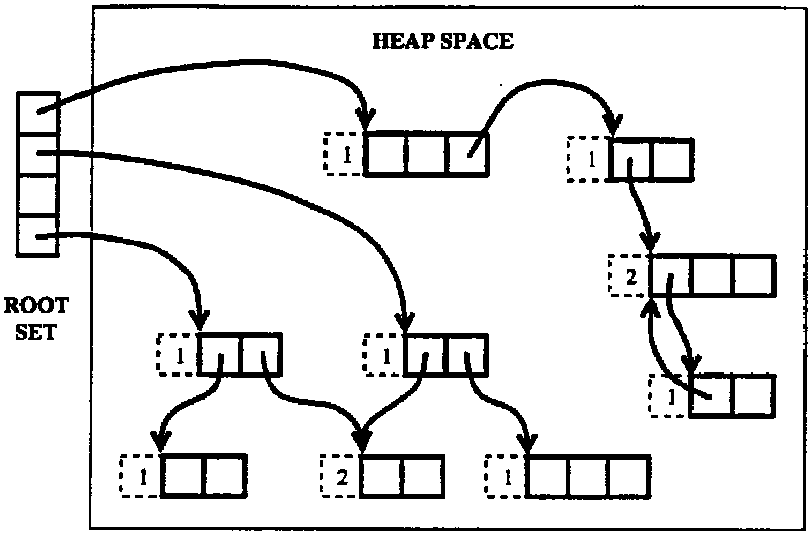
Basic Techniques

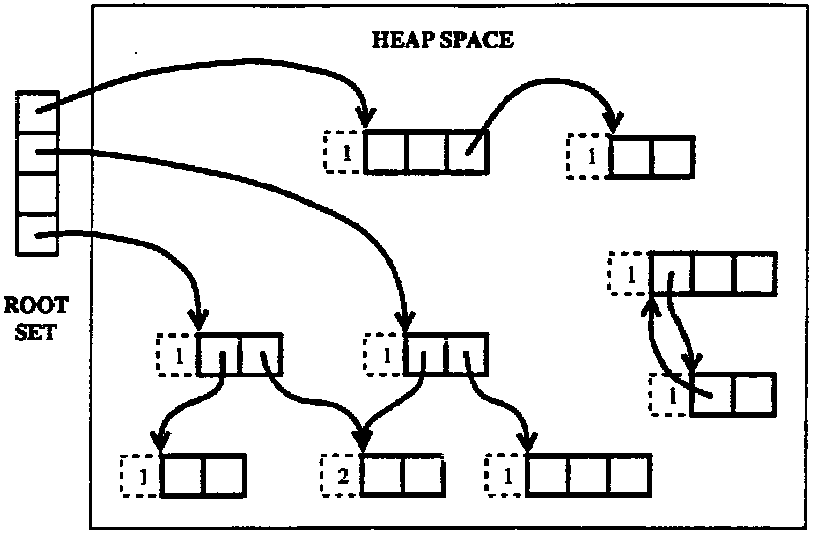
- Reference counting
  - each object has associated count of the references (pointers) to it
  - object's memory may be reclaimed when count reaches zero
  - incremental, interleaved closely with program execution

Basic Techniques (cont.)

- Reference counting problems
  - Problem with cycles
    - reference counts may never reach zero
    - programmers may need to avoid using cyclic data structures
  - Efficiency problems
    - short-lived stack variables can cause big overhead
  - Treatment: Deferred Reference Counting
    - adjust reference counts only now and then

Cycle Problem Illustrated





Basic Techniques (cont.)

- Mark-Sweep Collection
  - traversing pointer graph, marking the objects that are reached
  - sweeping memory to find all unmarked objects and reclaim their memory

Basic Techniques (cont.)

- Mark-Sweep problems
  - variable-size objects can cause memory fragmentation
  - cost is proportional to heap size
    - all live objects must be marked
    - all garbage objects must be collected
  - locality of reference is lost
    - can cause problems with virtual memory

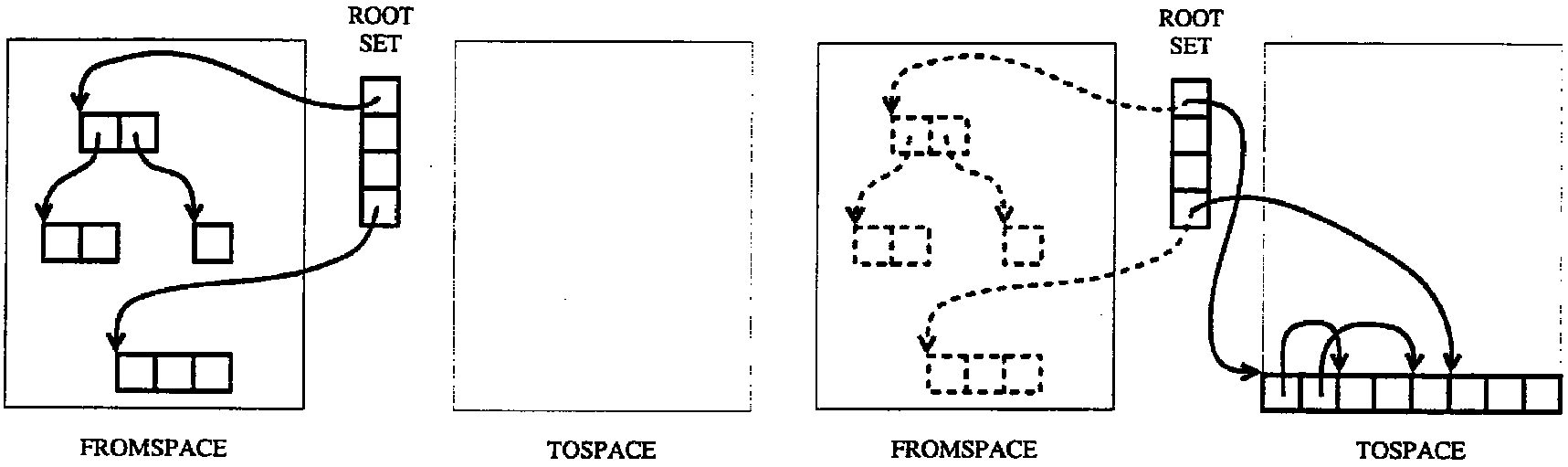
Basic Techniques (cont.)

- Mark-Compact Collection
  - traverses and marks reachable objects
  - live objects are moved until all are contiguous
  - rest of memory is single contiguous free space
  - eliminates fragmentation problem
  - makes allocation easy by incrementing pointer into free space
  - still, several passes over the data necessary

Basic Techniques (cont.)

- Copying Garbage Collection
  - moves all live objects into one area
  - rest of heap is then available
  - integration of data traversal and copying process
  - Example: semispace collector
    - heap is divided into two contiguous semispaces
    - only one is in use
    - GC copies live data to other semispace

Semispace Collector Illustrated



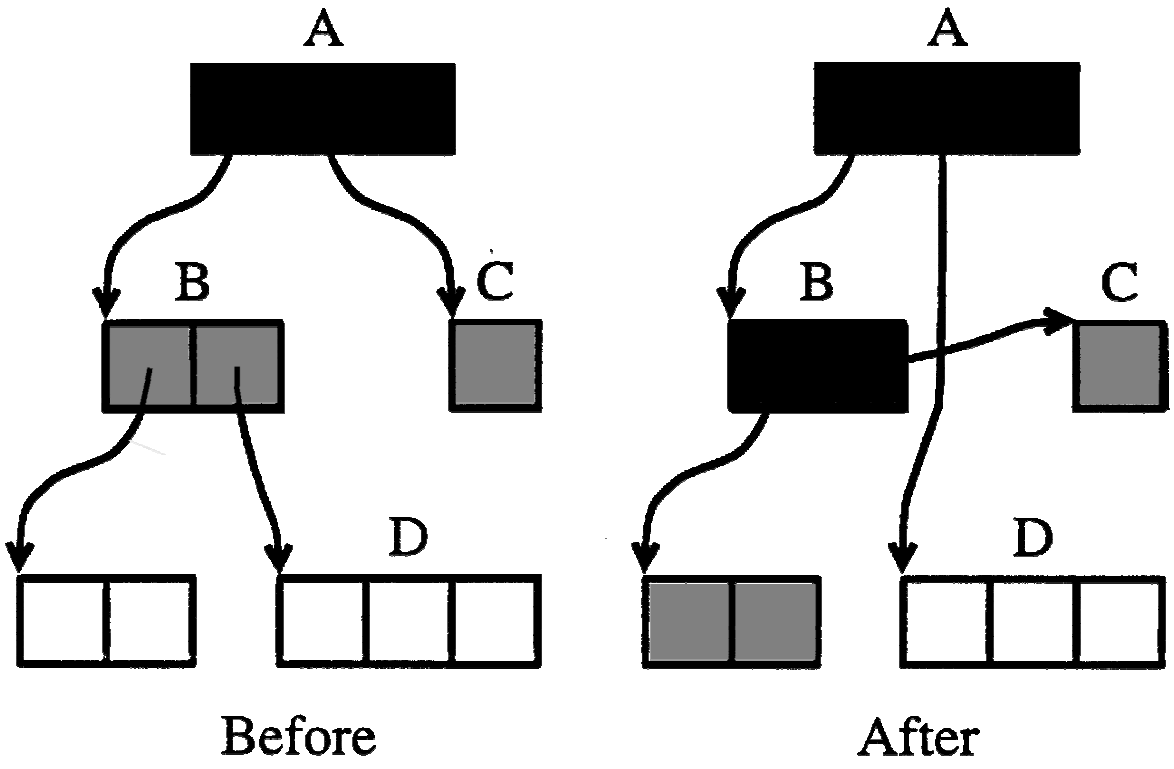
Basic techniques (cont.)

- Non-Copying Implicit Collection
  - spaces are seen as sets
  - two pointer link objects in doubly-linked list
  - “color” field indicates which set the object belongs to
  - only pointer and color field changes are required to move objects between sets

Incremental Tracing Collectors

- Tricolor marking
  - using three colors to mark objects during traversal:
    - white: object unmarked
    - gray: object has been reached, but its descendants may not have been
    - black: direct descendants are traversed
  - Only black objects are live in the end
  - Coordination with application necessary

Tricolor Marking Illustrated



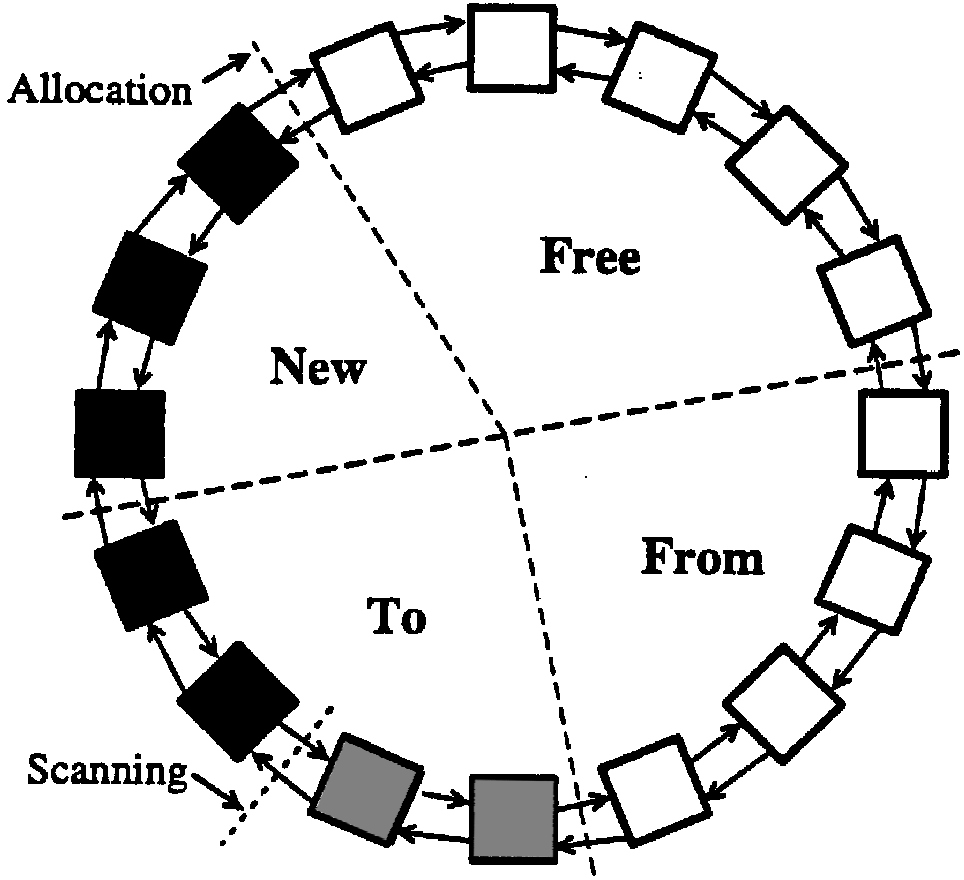
Incremental Collectors (cont.)

- Incremental Copying
  - read barrier for coordination with application
    - detects attempts to access pointers to white objects
    - hides temporary inconsistencies from application
  - objects allocated during collection are assumed to be live
    - are not claimed during current GC cycle

Incremental Collectors (cont.)

- The Treadmill
  - links lists into cyclic structure
  - divided into four sections:
    - New, Free, From, To
  - sections move around the cycle

Treadmill Illustrated



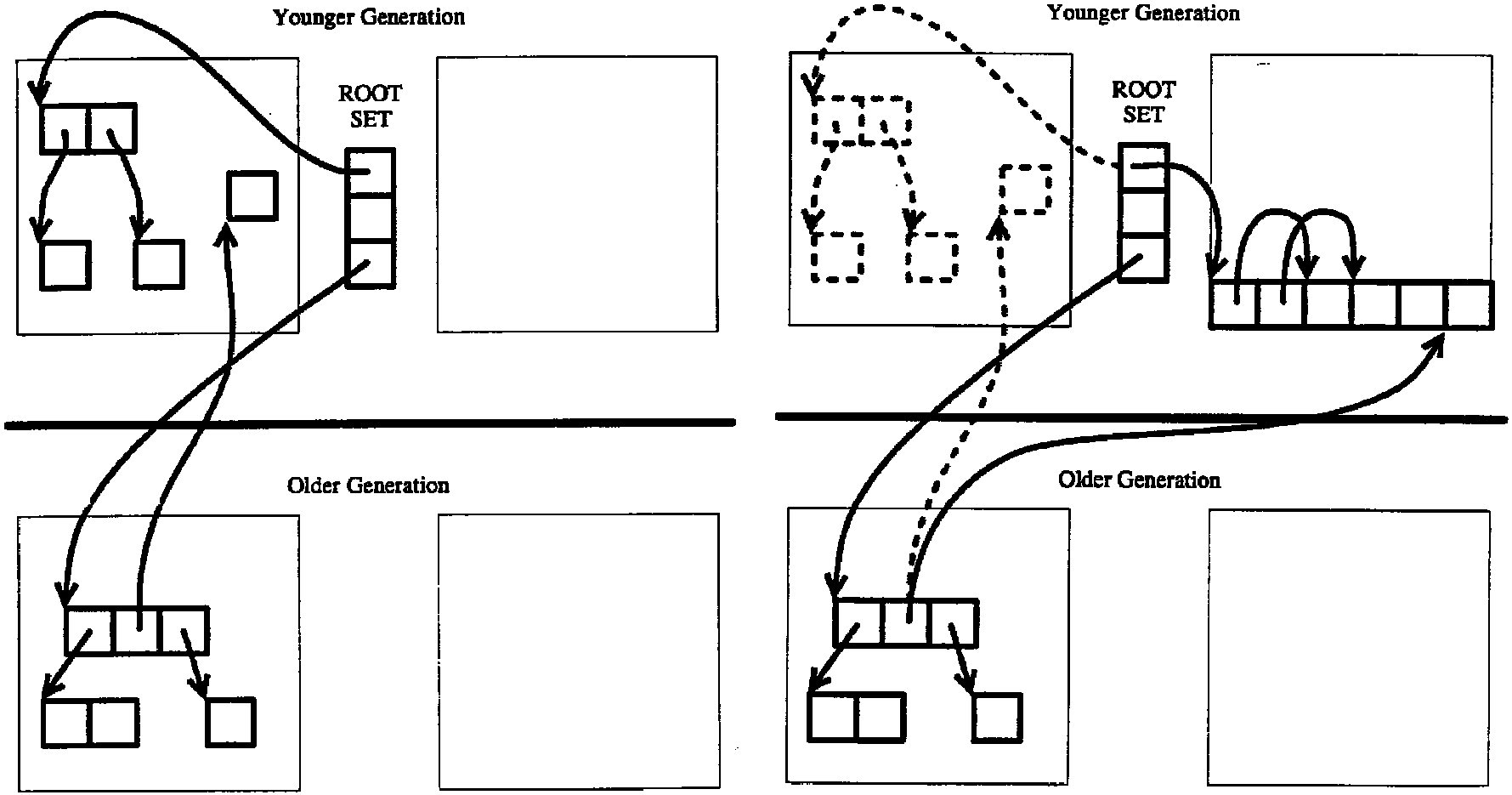
Incremental Collectors (cont.)

- Write-Barrier Algorithms
  - Snapshot-at-beginning
    - take a snapshot of the graph at the beginning of GC
    - if pointers are overwritten, GC can still find the objects
  - Incremental update
    - catch pointer writes into black (i.e., live) objects
    - change object status to gray

Generational Garbage Collection

- Observations:
  - Most objects live a very short time
  - Only a small percentage lives much longer
- Older objects are copied over and over
- Solution:
  - segregate objects into multiple areas by age
  - run GC less often on older objects
- Example: Multiple subheaps

Multiple Subheaps Illustrated



Tag-Free Garbage Collection

- Traditionally, GC (and type checking) required each datum to be tagged
- Strongly typed languages don't need tags
  - type checking is done at compile time
  - however, languages like ML keep tags for GC
  - space and time overhead

Tag-Free Garbage Collection (cont.)

- Compiler can generate code necessary to support GC
  - code is specific to program
  - compiler knows type of each datum, so no tagging is required
  - for each type in the program, there is a GC routine that manipulates objects of that type
  - for each procedure, compiler generates GC routines

Tag-Free GC (cont.)

- Advantages
  - more efficient use of heap space
  - more efficient execution
  - more accurate recognition of live data and garbage
- Disadvantage: increase in code size, but
  - simpler garbage routines
  - recognition of program points that can cause GC

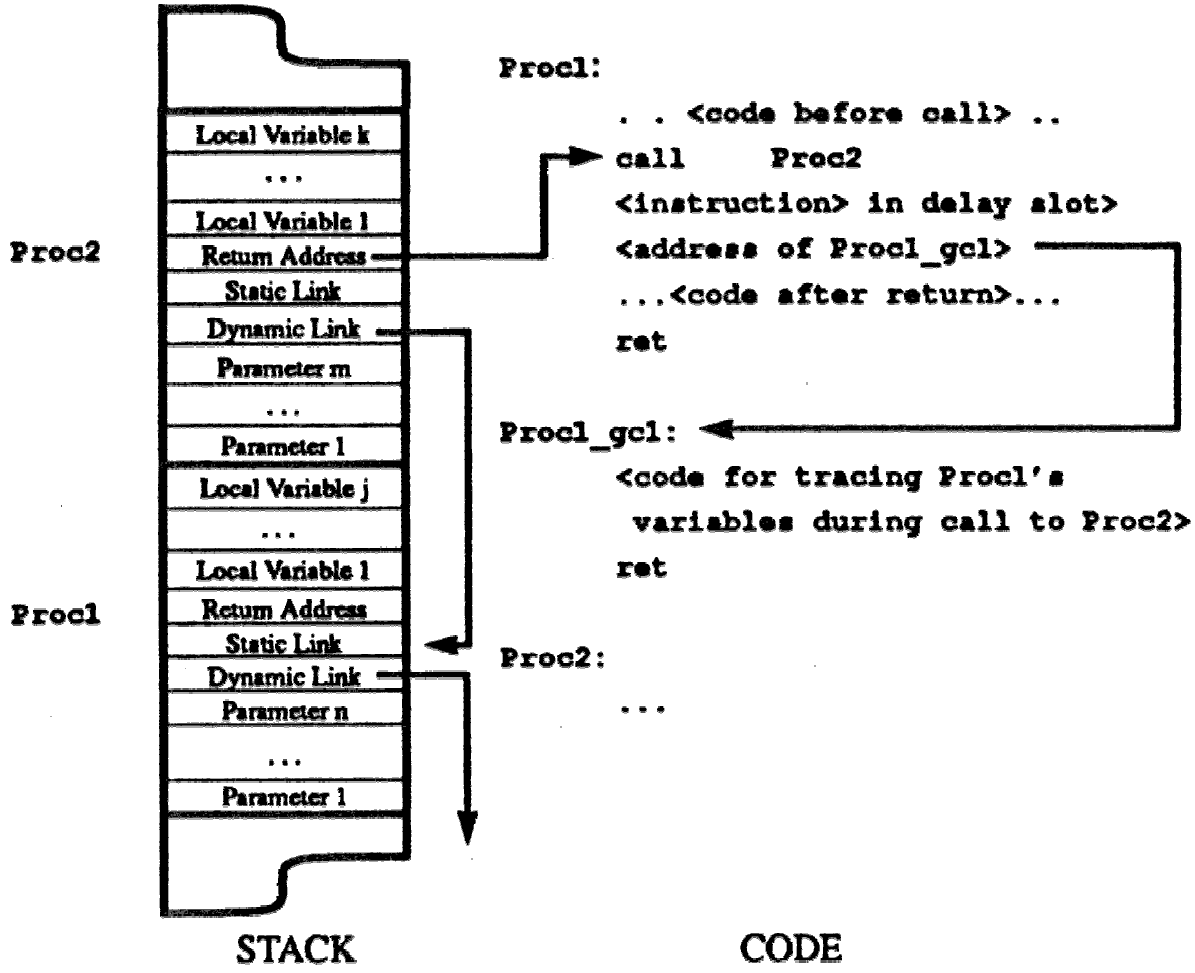
Interpretive Method

- each type has associated encoding of the type structure
- encoding is a parse-tree like representation called descriptor or template
- GC traverses descriptor to determine how to handle the substructures

Compiled Method

- gc routines generated by compiler
- needs to locate gc routines
  - use of table
    - problem: table update required for every creation of local variable on heap
  - better: use of return address pointers to determine which gc routine is associated with stack frame
    - observation: gc can only be initiated by call to a procedure (like cons, new, malloc)

Stack/Code Organization Illustrated



Polymorphism Support

- ML implementations execute the same code for all calls to a polymorphic function
  - gc routine can not know precisely all variable structures
  - calling procedures can be examined
    - problem: fair amount of stack traversing
  - better: stack traversal from oldest activation record to the most recent
    - may require initial traversal to perform pointer-reversal

Extension to Languages with Tasking

- Ada model: multiple tasks operating in a shared memory environment
- all tasks must be suspended during GC
  - tasks suspended immediately upon allocation attempt might not be in consistent state for GC
  - solution: tasks are suspended only on procedure calls
    - might allow some processes to run for a long time while others are suspended

Compiler Support for GC in Statically Typed Languages

- Requirements
  - avoidance of use of special hardware support
  - use of highly-optimizing compiler
    - no defeat or disallowance of compiler optimizations
      - challenge since compiler/optimizer may introduce complex pointer manipulation
  - avoidance of tagging
  - compiler knows which global variables, stack locations and registers contain pointers

Compiler Support for GC (cont.)

- Low-level requirements of collector
  - determine size of objects on heap
  - locate pointers in heap objects
  - locate pointers in global variables
  - find all references in stack and registers
  - find objects referred to using pointer arithmetic
  - update values obtained using pointer arithmetic when objects are moved

Implementation for use in Modula-3

- type descriptors in heap objects
- statically typed language makes compile-time location of pointers in global variables easy
- stack and register assignment may vary even within a procedure
- pointer update and following is complicated if pointer is untidy

Untidy Pointers

- introduced by language features or optimizations
  - strength reduction
  - virtual array origin
  - CSE
  - double indexing
- usually involves pointer arithmetic
  - derived values are created by pointer arithmetic
  - base values are values participating in derivation

Use of Tables for GC

- construct tables at compile time to assist in locating and updating all pointers
- one set of tables per gc-point
  - gc-points: where gc can occur
- three kinds of tables:
  - stack pointers: live tidy pointers in stack frame
  - register pointers: live tidy pointers in registers
  - derivations: live derived values

Use of Tables for GC (cont.)

- GC needs to locate the tables
  - use return addresses from stack frames to search a table that maps gc-points to gc tables
- use of register tables requires additional information about saved registers
- derivation tables are needed to update derived values when base values change

Derived Value Updates

- Two-step process
  - example: `a := b1 + b3 - b2 + E`
  - calculate E by applying the inverse operation for each base value: `a := a - b1 - b3 + b2`
  - note: derived value must be updated before any of its base values
  - after gc, reconstruct derived values from updated base values

Derivation Table Assumptions

- the base values are live whenever values derived from them are live
  - allows to update derived values in the first place
- operations used in the derivation have inverses
  - current implementation handles + and - only
- Extension to non-invertible operations would require redesign of tables

Complications

- base value may die before derived value does
- multiple derivations of a value reaching a gc-point
- indirect references used as base values in a derivation

Complications Illustrated

SOURCE

```
A: REF ARRAY [1..10] OF INTEGER;
FOR i := 1 TO LAST (A^) DO
  s := s + A^[i];
END;
```

OPTIMIZED

```
for (i = 1; i <= 10; i++) {
  s = s + *A++;
  <gc-point>
}
```

SOURCE

```
a:  REF ARRAY [1..5] OF
     REF ARRAY [5..9] OF INTEGER;
foo (a^[2]^[6]);
```

COMPILED

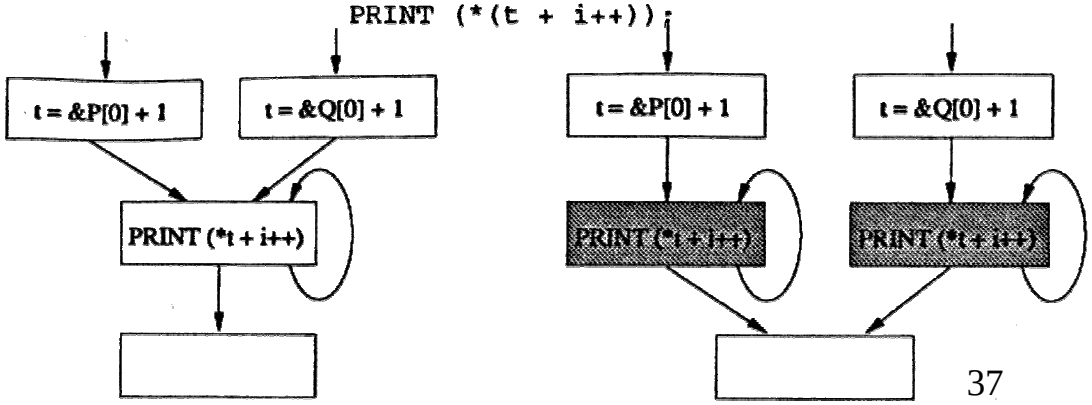
```
foo ( *(a + sizeof (int))
      + sizeof (int))
```

SOURCE

```
i := 1;
WHILE (cond)
  IF (inv) THEN
    PRINT (P[i]);
  ELSE
    PRINT (Q[i]);
  END;
  INC (i);
END;
```

OPTIMIZED

```
i = 1;
if (inv)
  t = &P[0] + 1;
else
  t = &Q[0] + 1;
while (cond)
  PRINT (*(t + i++));
```



Complications Resolved

- dead base problem
  - consider use of derived value as use of each of its base values
- ambiguous derivations
  - introduce path variables or use path splitting
- indirect references
  - preserving intermediate reference in stack slot or register

Implementation Issues

- table can get very large (45% of the size of optimized code)
  - remedies: use of delta tables
  - table compression
  - yields reduction to 16% of code size
- execution time overhead
  - ratio of stack tracing time to total gc time estimated between 1.7% and 6%

Benchmark Statistics

Program	Size	NGC	NPTRS	NDEL	NREG	NDER
typereg	3154	59	87	58	26	3
typereg-opt	2289	52	122	39	41	0
FieldList	4594	51	103	45	18	11
FieldList-opt	3330	82	319	61	70	11
takl	457	8	11	8	6	0
takl-opt	437	9	18	6	9	0
destroy	1240	12	14	11	2	0
destroy-opt	552	14	18	4	13	0

	Full Info		*δ-main*			
Program	Plain	Packing	Plain	Previous	Packing	PP
typereg	45.5	14.3	35.0	28.2	12.3	10.6
typereg-opt	51.4	17.2	41.6	35.5	16.0	14.0
FieldList	30.3	11.1	16.4	14.8	6.1	5.6
FieldList-opt	64.7	22.9	53.0	47.6	20.8	18.7
takl	51.6	17.9	41.1	34.1	16.0	14.2
takl-opt	55.8	19.7	43.9	37.5	17.6	15.6
destroy	17.1	5.9	17.1	15.2	6.6	6.1
destroy-opt	46.4	17.4	42.8	38.4	18.1	16.5