Compiler Construction

Lecture 24: Code Generation V (Implementation of Dynamic Data Structures)

Thomas Noll

Lehrstuhl für Informatik 2
(Software Modeling and Verification)

RWTH Aachen University

noll@cs.rwth-aachen.de

http://www-i2.informatik.rwth-aachen.de/i2/cc10/

Winter semester 2010/11

Outline

Variant Records

Example 24.1 (Variant records in Pascal)

```
TYPE Coordinate = RECORD
                    nr: INTEGER;
                    CASE type: (cartesian, polar) OF
                      cartesian: (x, y: REAL);
                      polar: (r : REAL; phi: INTEGER )
                    END
                  END;
VAR pt: Coordinate;
pt.type := cartesian; pt.x := 0.5; pt.y := 1.2;
```

Implementation:

- Allocate memory for “biggest” variant
- Share memory between variant fields

Dynamic Arrays

Example 24.2 (Dynamic arrays in Pascal)

```
FUNCTION Sum(VAR a: ARRAY OF REAL): REAL;
  VAR
    i: INTEGER; s: REAL;
  BEGIN
    s := 0.0; FOR i := 0 to HIGH(a) do s := s + a[i] END; Sum := s
  END
```

Implementation:

- Memory requirements unknown at compile time but determined by actual function/procedure parameters
  $\Longrightarrow$ no heap required
- Use array descriptor with following fields as parameter value:
  - starting memory address of array
  - size of array
  - lower index of array (possibly fixed by 0)
  - upper index of array (actually redundant)
- Use data stack or index register to access array elements

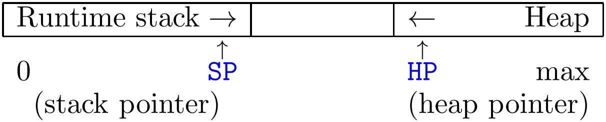
Outline

Dynamic Memory Allocation I

- Dynamically manipulated data structures (lists, trees, graphs, ...)
- So far: creation of (static) objects by declaration
- Now: creation of (dynamic) objects by explicit memory allocation
- Access by (implicit or explicit) pointers
- Deletion by explicit deallocation or garbage collection
  (= automatic deallocation of unreachable objects)
- Implementation: runtime stack not sufficient
  (lifetime of objects generally exceeds lifetime of procedure calls)

$\Longrightarrow$ new data structure: heap

- Simplest form of organization:



Dynamic Memory Allocation II

- New instruction: `NEW` ("`malloc`", ...)
  - allocates n memory cells where n = topmost value of runtime stack
  - returns address of first cell
  - formal semantics
    (`SP` = stack pointer, `HP` = heap pointer, `<.>` = dereferencing):

```
if HP - <SP> > SP
  then HP := HP - <SP>; <SP> := HP
  else error("memory overflow")
```

- But: collision check required for every operation which increases `SP` (e.g., expression evaluations)
- Efficient solution: add extreme stack pointer `EP`
  - points to topmost `SP` which will be used in the computation of current procedure
  - statically computable at compile time
  - set by procedure entry code
  - modified semantics of `NEW`:

```
if HP - <SP> > EP
  then HP := HP - <SP>; <SP> := HP
  else error("memory overflow")
```

Outline

Memory Deallocation

Releasing memory areas that have become unused

- explicitly by programmer
- automatically by runtime system (garbage collection)

Management of deallocated memory areas by free list (usually doubly-linked list)

- goal: reduction of fragmentation (= heap memory splitted in large number of non-contiguous free areas)
- coalescing of contiguous areas
- allocation strategies: first-fit vs. best-fit

Explicit Deallocation

- Manually releasing memory areas that have become unused
  - Pascal: `dispose`
  - C: `free`
- Problems with manual deallocation:
  - memory leaks:
    - failing to eventually delete data that cannot be referenced
    - critical for long-running/reactive programs (operating systems, server code, ...)
  - dangling pointer dereference:
    - referencing of deleted data
    - may lead to runtime error (if deallocated pointer reset to nil) or produce side effects (if deallocated pointer keeps value and storage reallocated)

$\Longrightarrow$ Adopt programming conventions (object ownership) or use automatic deallocation

Outline

Garbage Collection

- Garbage = data that cannot be referenced (anymore)
- Garbage collection = automatic deallocation of unreachable data
- Supported by many programming languages:
  - object-oriented: Java, Smalltalk
  - functional: Lisp (first GC), ML, Haskell
  - logic: Prolog
  - scripting: Perl
- Design goals for garbage collectors:
  - execution time: no significant increase of application run time
  - space usage: avoid memory fragmentation
  - pause time: minimize maximal pause time of application program caused by garbage collection (especially in real-time applications)

Preliminaries

- Object = allocated entity
- Object has type known at runtime, defining
  - size of object
  - references to other objects

  $\Longrightarrow$ excludes type-unsafe languages that allow manipulation of pointers (C, C++)
- Reference always to address at beginning of object ($\Longrightarrow$ all references to an object have same value)
- Mutator = application program modifying objects in heap
  - creation of objects by acquiring storage
  - introduce/drop references to existing objects
- Objects become garbage when not reachable by mutator

Reachability of Objects

- Root set = heap data that is directly accessible by mutator
  - for Java: static field members and variables on stack
  - yields directly reachable objects
- Every object with a reference that is stored in a reachable object is indirectly reachable
- Mutator operations that affect reachability:
  - object allocation: memory manager returns reference to new object
    - creates new reachable object
  - parameter passing and return values: passing of object references from calling site to called procedure or vice versa
    - propagates reachability of objects
  - reference assignment: assignments `p := q` where with references `p` and `q`
    - creates second reference to object referred to by `q`, propagating reachability
    - destroys orginal reference in `p`, potentially causing unreachability
  - procedure return: removes local variables
    - potentially causes unreachability of objects
- Objects becoming unreachable can cause more objects to become unreachable

Identifying Unreachable Objects

Principal approaches:

- Catch program steps that turn reachable into unreachable objects
  $\Longrightarrow$ reference counting
- Periodically locate all reachable objects; others then unreachable
  $\Longrightarrow$ mark-and-sweep

Outline

Reference-Counting Garbage Collectors I

Working principle:

- Add reference count field to each heap object (= number of references to that object)
- Mutator operations maintain reference count:
  - object allocation: set reference count of new object to 1
  - parameter passing: increment reference count of each object passed to procedure
  - reference assignment `p := q`: decrement/increment reference count of object referred to by `p`/`q`, respectively
  - procedure return: decrement the reference count of each object that a local variable refers to (multiple decrement if sharing)
- Moreover: transitive loss of reachability
  - when reference count of object becomes zero
    $\implies$ decrement reference count of each object pointed to (and add object storage to free list)

Example 24.3

(on the board)

Reference Counting Garbage Collectors II

Advantage: Incrementality

- collector operations spread over mutator's computation
  - short pause times (good for real-time/interactive applications)
  - immediate collection of garbage (low space usage)
- exception: transitive loss of reachability (removing a reference may render many objects unreachable)
- but: recursive modification can be deferred

Disadvantages:

- Incompleteness:
  - cannot collect unreachable, cyclic data structures (cf. Example 24.3)
- High overhead:
  - additional operations for assignments and procedure calls/exits
  - proportional to number of mutator steps (and not to number of heap objects)

Conclusion: use for real-time/interactive applications

Outline

Mark-and-Sweep Garbage Collectors I

Working principle:

- Mutator runs and makes allocation requests
- Collector runs periodically
  (typically when space exhausted/below threshold)
  - computes set of reachable objects
  - reclaims storage for objects in complement set

Mark-and-Sweep Garbage Collectors II

Algorithm 24.4 (Mark-and-sweep garbage collection)

Input: *heap Heap, root set Root, free list Free*

Procedure:

1. *(* Marking phase *)*
   for each o in Heap, (initialize reachability bit *)*
   let $r_o :=$ true *iff o referenced by Root*
2. *let* $W := \{o \mid r_o = \text{true}\}$ *(* working set *)*
3. *while* $o \in W \neq \emptyset$ *do*
   1. *let* $W := W \setminus \{o\}$
   2. *for each* o' *referenced by o with* $r_{o'} = \text{false}$,
      let $r_{o'} = \text{true}; W := W \cup \{o'\}$
4. *(* Sweeping phase *)*
   for each o in Heap with $r_o =$ false, *add o to Free*

Output: *modified free list*

Example 24.5

(on the board)

Mark-and-Sweep Garbage Collectors III

Advantages:

- Completeness: identifies all unreachable objects
- Time complexity proportional to number of objects in heap

Disadvantage: “stop-the-world” style

$\Longrightarrow$ may introduce long pauses into mutator execution
(sweeping phase inspects complete heap)

Conclusion: refine to short-pause garbage collection

- Incremental collection: divide work in time by interleaving mutation and collection
- Partial collection: divide work in space by collecting subset of garbage at a time

(see Chapter 7 of A.V. Aho, M.S. Lam, R. Sethi, J.D. Ullman: *Compilers – Principles, Techniques, and Tools; 2nd ed.*, Addison-Wesley, 2007)