



Compiler Construction

Lecture 18: Code Generation IV (Implementation of Dynamic Data Structures)

Summer Semester 2017

Thomas Noll

Software Modeling and Verification Group

RWTH Aachen University

<https://moves.rwth-aachen.de/teaching/ss-17/cc/>

Recap: Static Data Structures

Modified Syntax of EPL

Definition (Modified syntax of EPL)

The **modified syntax of EPL** is defined as follows (where $n \geq 1$):

\mathbb{Z} : z (* z is an integer *)
 \mathbb{B} : $b ::= \text{true} \mid \text{false}$ (* b is a Boolean *)
 \mathbb{R} : r (* r is a real number *)
Con : $c ::= z \mid b \mid r$ (* c is a constant *)
Ide : I, J (* I, J are identifiers *)
Type : $T ::= \text{bool} \mid \text{int} \mid \text{real} \mid I \mid \text{array}[z_1..z_2] \text{ of } T \mid$
 $\text{record } l_1 : T_1; \dots; l_n : T_n \text{ end}$
Var : $V ::= I \mid V[E] \mid V.I$
Exp : $E ::= c \mid V \mid E_1 + E_2 \mid E_1 < E_2 \mid E_1 \text{ and } E_2 \mid \dots$
Cmd : $C ::= V := E \mid C_1; C_2 \mid \text{if } E \text{ then } C_1 \text{ else } C_2 \mid \text{while } E \text{ do } C$
Dcl : $D ::= D_C D_T D_V$
 $D_C ::= \varepsilon \mid \text{const } l_1 := c_1; \dots; l_n := c_n;$
 $D_T ::= \varepsilon \mid \text{type } l_1 := T_1; \dots; l_n := T_n;$
 $D_V ::= \varepsilon \mid \text{var } l_1 : T_1; \dots; l_n : T_n;$
Pgm : $P ::= D C$

Pseudo-Dynamic Data Structures

Variant Records

Example 18.1 (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

Pseudo-Dynamic Data Structures

Dynamic Arrays

Example 18.2 (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 \implies **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

Heap Management

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)
- ⇒ new data structure: **heap**
- Simplest form of organisation:



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, $\langle . \rangle$ = dereferencing):

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

- But: collision check also required for every operation which increases SP
(in particular, for expression evaluations)
- Efficient solution: add **extreme stack pointer** EP
 - points to topmost SP which will be used in the computation of current procedure (except for calls)
 - statically computable at compile time for each procedure
 - set by procedure entry code upon call
 - modified semantics of `NEW`: `if HP - <SP> > EP`

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



Memory Deallocation

Releasing memory areas that have become unused

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

Management of deallocated memory areas (chunks) by **free list**

- usually organised as doubly-linked list, possible ordered by size
- goal: reduction of **fragmentation**
(= heap memory split in large number of non-contiguous chunks)
- **coalescing** of contiguous chunks
- allocation strategies: **first-fit** vs. **best-fit**

Memory Deallocation

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 anymore
 - critical for long-running/reactive programs (operating systems, server code, ...)
 - **dangling pointer dereference** (“use after free”):
 - 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 re-allocated later)
- ⇒ Adopt programming conventions (object ownership) or use **automatic deallocation**

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 runtime
 - space usage: avoid memory fragmentation
 - pause time: minimise maximal pause time of application program caused by garbage collection (especially in real-time applications)

Preliminaries

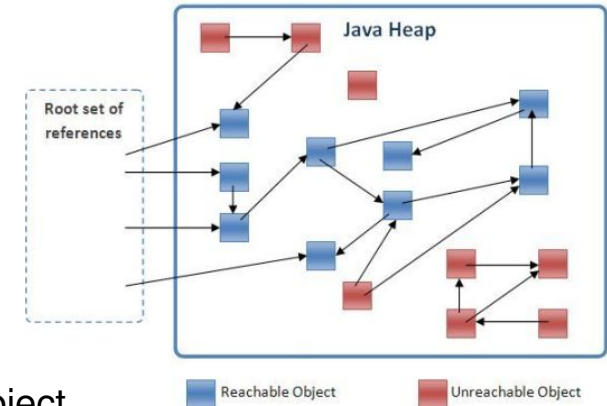
- **Object** = allocated entity
- Object has **type** known at runtime, defining
 - size of object
 - references to other objects

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

Garbage Collection

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 $p := q$ with references p and q
 - creates second reference to object referred to by q , propagating reachability
 - destroys original 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
⇒ **reference counting**
- Periodically locate all reachable objects; others then unreachable
⇒ **mark-and-sweep**

Reference-Counting Garbage Collection

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 (multiple increment if shared)
 - **reference assignment** $p := q$: decrement/increment reference count of object referred to by p/q
 - **procedure return**: decrement reference count of each object that a local variable refers to (multiple decrement if shared)
- Moreover: **transitive loss of reachability**
 - when reference count of object becomes zero:
decrement reference count of each object pointed to (and add object storage to free list)

Example 18.3

(on the board)

Reference-Counting Garbage Collection

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 (reference removal may produce further garbage)
- but: recursive modification can be deferred

Disadvantages

- **Incompleteness:** cannot collect unreachable cyclic data structures (cf. Example 18.3)
- **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**

Mark-and-Sweep Garbage Collection

Mark-and-Sweep Garbage Collectors I

Working principle

- **Mutator** runs and makes allocation requests
- **Collector** runs periodically (typically when space exhausted/at critical threshold)
 - computes set of reachable objects (by attaching reachability flags)
 - reclaims storage for objects in complement set

Mark-and-Sweep Garbage Collection

Mark-and-Sweep Garbage Collectors II

Algorithm 18.4 (Mark-and-sweep garbage collection)

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

Procedure: 1. (Marking phase *)*

for each o in Heap, let $r_o := \text{true}$ iff o referenced by Root (initialise r flags *)*

2. let $W := \{o \mid r_o = \text{true}\}$ (initialise working set *)*

3. while $o \in W \neq \emptyset$ do

i. let $W := W \setminus \{o\}$

ii. 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 $r_o = \text{false}$, add o to Free

Output: modified free list

Example 18.5

(on the board)

Mark-and-Sweep Garbage Collection

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

- May introduce long pauses into mutator execution (sweeping inspects complete heap)

Solution: 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

Mark-and-Sweep Garbage Collection

Interface Between Compiler and Garbage Collector

Compiler interface

Compiler interacts with garbage collector by

- generating code for **memory allocation**
- generating description of **root locations** (for each garbage-collecting cycle)
- generating description of **memory layout** of objects
- for (some variants of) incremental collection:
generating instructions to implement **read/write barriers**