Compiler Practical 2013

Storage Administration: Overview

Berthold Hoffmann (B. Gersdorf, T. Röfer)
hof@informatik.uni-bremen.de
Cartesium 2.48

Deutsches Forschungszentrum für Künstliche Intelligenz GmbH
Universität Bremen

Structure

1. Heap Administration
2. Automatic Garbage Collection
3. Reference Countin
4. Mark and Sweep Collectors
5. Copying Collectors
6. Mark and Compact Collectors

Heap Administration: Push-Only Stack

- Can be implemented very fast and easy
- Standard implementation in LOOP-0
- Full, some time ... ;-)
- Cannot be used if the program shall run for some time, and works with data on the heap.

Heap Aministration: Free Lists

Deutsches Forschungszentrum für Künstliche Intelligenz GmbH
Universität Bremen

- For programming languages mit explicit (manual) allocation/de-allocation:
  - Keeping one or several lists that link free blocks on the heap
    - Allocation in these lists
    - De-allocated blocks are added to one of these lists
  - Fragmentation (and counter measures)
    - Bigger blocks are no longer available
    - Allocation: *First-fit* vs. *Best-fit*
    - De-allocation: Joining adjacent blocks to bigger blocks

Automatic Garbage Collection I

- Explicit storage management is error-prone
  - Storage leaks
  - Multiple de-allocation
  - Dereferencing of „dangling pointers“
- Automatic storage management can be faster
  (Benjamin Zorn, The Measured Cost of Conservative Garbage Collection, *Software - Practice and Experience*)
  - Using times of low activity
  - Generational garbage collection

Autom. Garb. Collection II

Drawbacks

- Temporary suspension of program execution during garbage collection
- Some operations are slowed down, e.g., assignments, by reference counting
- Possibly additional initialization of variables that are otherwise useless

De-Allocation of Unused Storage

Deutsches Forschungszentrum für Künstliche Intelligenz GmbH
Universität Bremen

With simple de-allocation:

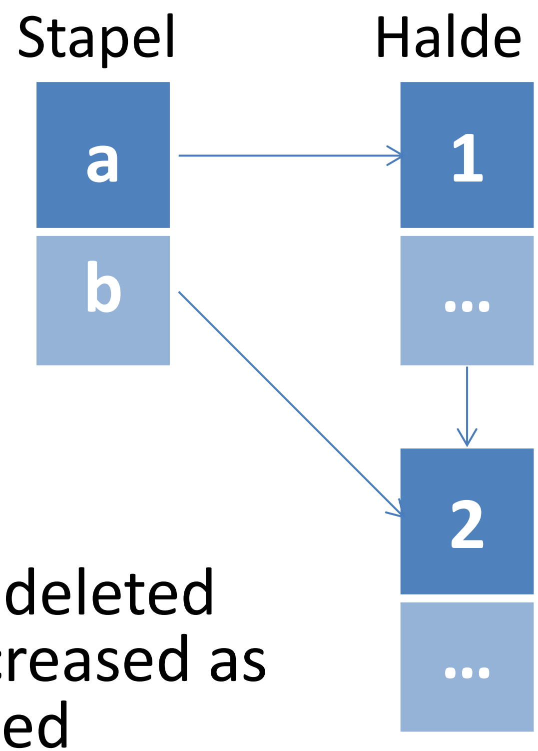
- Used storage is untouched
- All pointers / references stay valid
- Can be done in parallel with program execution
- Does not solve fragmentation

Compactification

- Used storage is pushed together
- All pointers / references have to be adjusted
  - On the stack
  - Within objects on the heap
- Cannot be done in parallel to program execution

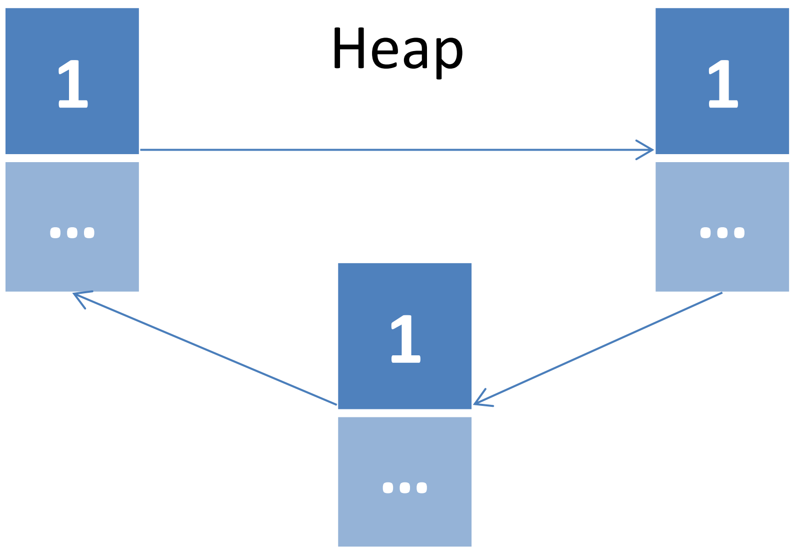
Reference Counters

- Every object on the heap gets a reference count
- The counter is set to 0 during allocation
- With every assignment to a reference
  - The counter of the newly assigned object is increased, and the counter of the object held previously is decreased
  - If this count equals 0, the object is deleted
  - For every object reference of the deleted object, the reference count is decreased as well, and the object may be deleted



Reference Counters: Cycles

- Cyclic structures cannot be de-allocated



Separate Storage Adjustment

Deutsches Forschungszentrum für Künstliche Intelligenz GmbH
Universität Bremen

- When?
  - If no heap space is available
  - If there is time
- How?
  - Determining which blocks on the heap can be reached from the stack
  - De-allocation of all not reachable blocks
  - Compactification of the heap

Mark and Sweep Collector

Deutsches Forschungszentrum für Künstliche Intelligenz GmbH
Universität Bremen

- Mark every block on the heap as *not reachable*
- Construct the *RootSet*: all blocks on the heap that are directly referenced from the stacl
- For every block *b* in the RootSet that is marked as unreachable:
  - Mark *b* as *reachable*
  - Check, recursively, every block *b'* that is still unreachable, and is referenced from block *b*
- De-allocate every block on the heap that is still marked as *unreachable*
- Cost: *O*(number of blocks on the heap)

Bakers Mark and Sweep Collector

- Every block belongs always to exactly one of the following lists
  - *free*: list of free blocks
  - *unreached*: **list of occupied blocks** (additional mark within the blocks)
  - *unscanned*: temporry list of unscanned blocks
  - *scanned*: temporary list of scanned blocks
- Cost: O(number of reachable blocks)

Bakers Mark and Sweep Collector

Deutsches Forschungszentrum für Künstliche Intelligenz GmbH
Universität Bremen

```
scanned := Ø
unscanned := blocks in RootSet
WHILE unscanned # Ø DO
    WITH b ∈ unscanned DO
        unscanned := unscanned \ b
        scanned := scanned ∪ {b}
        FOR EACH b‘ referenced from b DO
            IF b‘ ∈ unreached THEN
                unreached := unreached \ b‘
                unscanned := unscanned ∪ {b‘}
            END IF
        END FOR
    END WITH
END WHILE
free := free ∪ unreached
unreached := scanned
```

Conservative Storage Administration

- A storage block is *reachable* if the stack or a reachable heap blocks contain numbers that *might be* heap addresses
- Works without information about the structure of data on stack and heap
- Compactification is not possible
- May miss to de-allocate some blocks

Non-Cons. Storage Administration

Deutsches Forschungszentrum für Künstliche Intelligenz GmbH
Universität Bremen

- Structure information is used to identify references on the stack and within heap blocks
- Only these are checked for reachability
- Language must be *type-safe*
  - References always point to compatible objects

Cheney's Mark and Sweep Collector

- Blocks are copied from one heap to a second one, and back
- Drawback: Halves the storage capacity
- Cost *O*(size of reachable blocks
  + number of heap)

```
FOR EACH b ϵ Heap DO
	b.newAddr := NULL
END FOR
```

```
free := start address of target heap
FOR EACH b ϵ RootSetDO
	b := lookupNewAddr(b)
END FOR
b := start address of target heap
WHILE b < free DO
	FOR EACH Reference b.r aus b DO
		b.r := lookupNewAddr(b.r)
	END FOR
	b := b + b.size
END WHILE
```

```
METHOD lookupNewAddr(b) IS
	IF b.newAddr = NULL THEN
		b.newAddr := free
		free := free + b.size
		Copyb to b.newAddr
	END IF
	RETURN b.newAddr
END METHOD
```

Mark and Compact Collector

Deutsches Forschungszentrum für Künstliche Intelligenz GmbH
Universität Bremen

- Determine reachable blocks (as with *Mark and Sweep*)
- Allocate new addresses
- Correct references
- Copy blocks
- Cost: *O*(size of reachable blocks + number of heap blocks)

```
FOR EACH b ϵ RootSetDO
    b := b.newAddr;
END FOR
```

```
FOR EACH b ϵ Heap (ascending) DO
    IF b.reached THEN
        FOR EACH reference b.r aus b
        DO b.r := b.r.newAddr
        END FOR
    END IF
END FOR
```

```
free := start address of heap
FOR EACH b ϵ Heap (ascending) DO
    IF b.reached THEN
        b.newAddr := free
        free := free + b.size
    END IF
END FOR
```

```
FOR EACH b ϵ Heap (ascending) DO
    IF b.reached THEN
        Copyb to b.newAddr
    END IF
END FOR
```