Compilers and Language Processing Tools

Summer Term 2011

Prof. Dr. Arnd Poetzsch-Heffter

Software Technology Group
TU Kaiserslautern

Content of Lecture

5. Garbage Collection

Chapter Outline

Garbage collection

More and more programming languages use automatic memory management:

- Simplification of programming
- Feasible costs
- Security aspects

For language implementation, this means that

- the set of reachable objects has to be known
- objects/record variables must be suitable for garbage collection
- garbage collection must be implemented

The first two aspects concern the compiler.

Garbage collection (2)

The third aspect *mainly* concerns the runtime environment.

The runtime environment performs tasks that are the same for all user programs. It is linked to every user program.

Learning objectives:

- Overview of basic procedures for garbage collection.

Naming convention

Each source language has different memory objects that are relevant for garbage collection:

- procedural languages: record variables
- object-oriented languages: objects
- functional languages: term representations, function closures

In the following, we use the term "*object*" to refer to an entity to be garbage collected.

Root variables and reachability

An object is reachable from a set of variables V in a state S

- if it is (directly) referenced by a variable $x \in V$ in S, or
- if it is referenced by a variable of an object reachable from V in S.

The set of root variables *RV of a state S* contains the variables:

- allocated globally (e.g., global variables, static variables) or
- allocated on the stack (e.g., instances of local variables, actual parameters)

such that only the objects reachable from *RV* can be "used" in an execution following S.

An object is reachable *in a state S*, if it is reachable from the root variables of S.

Root variables and reachability (2)

Remarks:

- Unreachable objects remain unreachable.
- Garbage collection frees memory that is used by unreachable objects.

Outline:
Garbage collection by

- reference counting
- mark and sweep
- copying collection

5.1 Reference Counting

Reference counting

Garbage collection by reference counting works without support of the runtime environment. The corresponding code for memory management is attached to each user program by the compiler.

Idea:

For each object X, an object-local variable, the reference counter, stores the number of references pointing to X at maximum. The compiler generates code that

- increments/decrements the reference counter
- triggers deallocation of memory if the reference counter of an objects is set to zero.

Reference counting (2)

Translation incorporating reference counting

In principle, for each assignment zv:= z of a pointer z to the pointer variable zv, code for the following actions has to be generated:

```
zv.count  := zv.count  - 1 ;
if zv.count = 0  then addToFreeList(zv);
z.count   := z.count + 1 ;
zv        := z ;
```

where `addToFreelist(zv)` enters the object X that is referenced by zv into the list of unreachable objects.

Reference counting (3)

When X is added to the freelist or when the memory of X is reused, the reference counters of the objects pointed to by X's instance variables are decremented.

Additionally, the respective reference counters have to be decremented appropriately if root variables are deallocated (in the epilog of procedures).

Reference counting (4)

Discussion:
Reference counting is relatively easy to implement, but has two significant drawbacks:

- unreachable objects that reference each other cyclically or that are referenced to by a cycle cannot be deallocated (Possible Solution: combination of reference counting with one of the following two approaches)
- non-optimizing implementations are very inefficient, but clever optimizations increase the implementation complexity enormously.

Remark:
Reference counting is important for implementing distributed and/or persistent objects.

5.2 Mark and Sweep

Mark and sweep

Garbage collection with the mark and sweep approach works as a co-routine with the user program and is part of the runtime environment:

If there is no more memory available or there is a suitable point during program execution, garbage collection is triggered.

The compiler does not have to generate code for garbage collection. Only, memory space

- for a marking bit and
- for storing the number of instance variables of the object

have to be reserved in the object layout.

Mark and sweep (2)

Idea: 2 phases (mark and sweep)

- Mark all reachable objects by a depth-first search starting from the root variables.
- Sweep: Traverse the heap and purge all non-marked objects.

Mark and sweep algorithm

```
void mark( RootSet rs ) {
  forall v in rs {
    depthTraversal(v)
  }
}

void depthTraversal( Pointer y ) {
  if( y points to an object Y && !y.mark ) {
    y.mark := true;
    forall fields f of Y {
      if( f is of a pointer type )
        depthTraversal( y.f );
    }
  }
}
```

Mark and sweep algorithm (2)

```
void sweep(){
  pointer zv := first address of heap;
  while( zv < (last address of heap) ) {
    if( zv points to an object ) { // not in FreeList
      if( !zv.mark ){
        addToFreeList(zv);
      } else {
        zv.mark := false;
      }
    }
    zv := zv + sizeof(zv);
  }
}
```


Problem of naive implementation

For recursive depth-first traversal with maximal depth t, a stack with up to t stack frames is required, i.e., the required stack space is potentially larger than the complete heap.

Steps towards solution:

1. Work with explicitly controlled stack
2. Use pointer reversal

Depth-first traversal with explicit stack

```
Stack frontSet := emptyStack();

void depthTraversal( Pointer y ) {
  if( y points to an object && !y.mark ) {
    y.mark := true;
    push( y, frontSet );
    while( !isEmpty(frontSet) ) {
      Pointer x := top( frontSet );
      pop( frontSet );
      forall fields f of object pointed to by x {
        if( f is of a pointer type && x.f!=null && !x.f.mark ){
          x.f.mark:= true;
          push( x.f, frontSet );
        }
      }
} } }
```


Depth-first traversal with pointer reversal

The explicit stack also requires too much memory,
but it is the key to realizing pointer reversal.

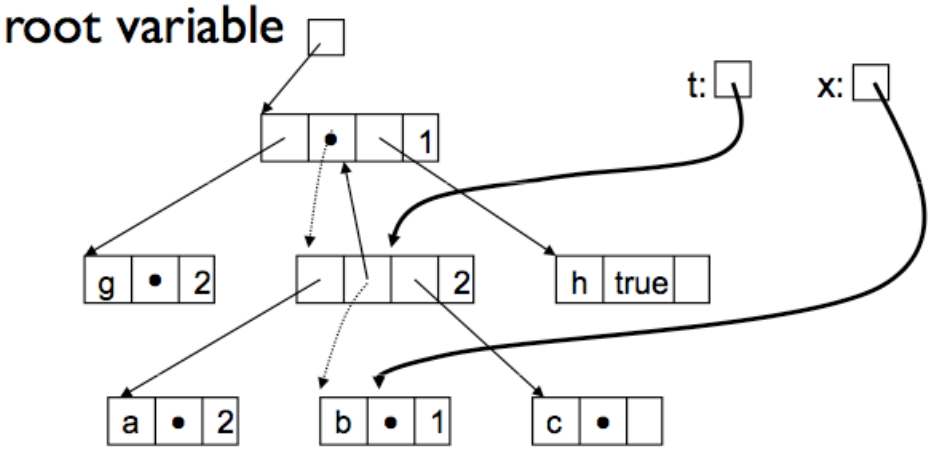
Idea: Use the visited instance variables to realize the stack:

- Each object gets an additional variable that stores the number of the currently processed instance variables, starting with 0.
- The state in a loop iteration consists of
  - a pointer x to the current object and
  - the number i of the current instance variables f of x

We distinguish three cases:

- $x.f_i$ is a pointer to the heap and not marked
- $x.f_i$ is not a pointer to the heap or already marked
- x is completely processed such that the next object referencing x is processed

Depth-first traversal with pointer reversal (2)



Depth-first traversal with pointer reversal (3)

In the following version of depth-first traversal (cf. figure on last slide),

- *x* contains the pointer to the currently processed object
- *t* contains the pointer to the predecessor of *x*
- *y* contains a pointer for intermediate values
- the instance variable index contains the index of the instance variable to be processed next
- *i* contains the current index of *x*

Depth-first traversal with pointer reversal – algorithm

```
void depthTraversal( Pointer x ){
  if( x points to an object && !x.mark ) {
    x.mark := true;
    t := null; x.index := 0;
    while( true ){
      i:= x.index;
      if( i < (number of fields of object pointed to by x) ) {
        if( f_i is of a pointer type && x.f!=null && !x.f_i.mark ){
          x.f_i.mark:= true;
          y:= x.f_i; x.f_i:= t; t:= x; x:= y; x.index := 0;
        } else { x.index := i+1; }
      } else {
        if( t==null ) return;
        y:= x; x:= t; i:= x.index;
        t:= x.f_i; x.f_i:= y;
        x.index:= i+1;
} } } }
```


Discussion of mark and sweep

- Advantages
  - easy to implement with acceptable runtime efficiency
  - good memory usage
  - addresses remain unchanged
- Disadvantages wrt. copying approaches
  - complexity proportional to size of heap
  - no support for locality and for avoiding fragmentation
  - not easy to refine/improve

5.3 Copying Collection

Copying collection

Coping garbage collection approaches are part of the runtime environment (similar to mark and sweep)

Idea:

- Split heap into two (or more) parts
- In the (active) parts, a pointer indicates the beginning of the free memory area (no list of free memory chunks needed)
- Copy all reachable objects from one part (old part, *from-space*) to another part (new part, *to-space*)
- Set root variables to the objects into the new part

The old part is purged. While copying, the object graph is compacted.

Illustration of copying garbage collection

Before garbage collection:

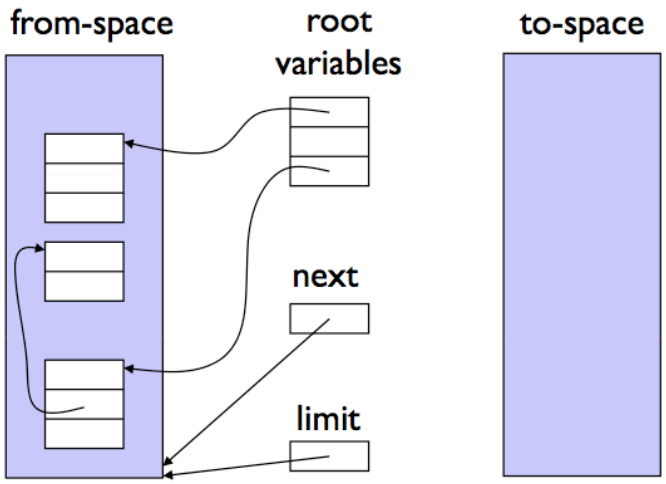
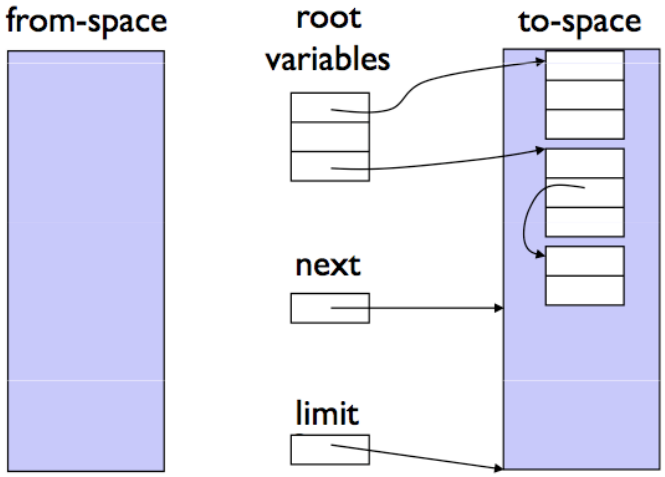


Illustration of copying garbage collection (2)

After garbage collection



Cheney's algorithm (1970)

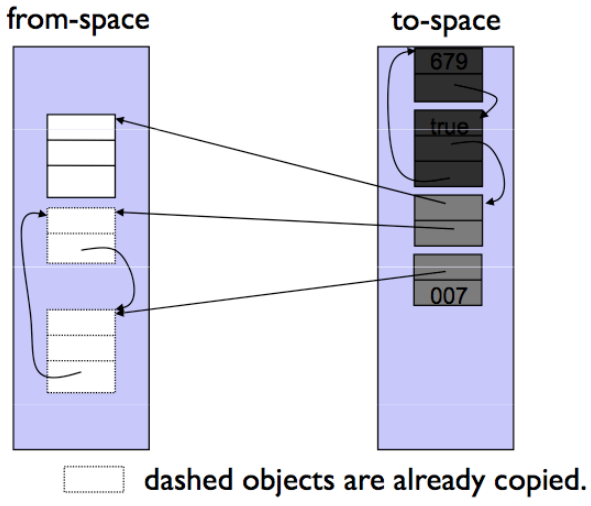
Idea: Copy all reachable objects with breadth-first traversal, i.e., first the set M1 of objects that are reachable from the root variables, then the set of objects M2 that are reachable from objects in M1, etc.

An object *x* has three different states (w,g,b):

- *white*: *x* is still in the old part (from-space)
- *gray*: *x* is copied to the new part (to-space),
  but it still references objects in the old part (from-space)
- *black*: *x* is in the new part (to-space)
  and only references objects in the new part (to-space)

Cheney's algorithm (1970) (2)

Example:



Cheney's algorithm - realization

Needed information:
For each object *x* in the from-space, it has to be checkable whether it has already been copied and, if so, where to. We assume too additional fields:

```
bool isCopied
Object forward
```

If `x.isCopied` is true, `x.forward` returns the pointer to the copy in the to-space (*forwarding pointer*).

Cheney's algorithm - realization (2)

Procedure for changing a pointer *p* pointing to the from-space to a pointer *p* pointing to the to-space, if applicable, with copying of the referenced object:

```
Pointer move( Pointer x ){
  if( x is pointer to from-space ){
    if( !x.isCopied ) {
      forall fields f of object pointed to by x {
        next.f = x.f;
      }
      x.isCopied = true;
      x.forward := next;
      next:= next + sizeof(x);
    }
    return x.forward;
  } else return x;
}
```

Cheney's algorithm - realization (3)

Copying garbage collection with breadth-first traversal:

```
void copyingGarbageCollection( RootSet rs ){
  Pointer z := beginning of new part;
  next := beginning of new part;
  forall x in rs { move(x); }
  while( z < next ) {
    forall fields f of object pointed to by z {
      z.f := move(z.f);
    }
    z := z + sizeof(z);
  }
}
```


Discussion

- Advantages wrt. mark and sweep (see above)
- Disadvantages (in the presented form):
  - "double" memory space required
  - breadth-first search breaks locality
  - no differentiation wrt. age of objects

Improving copying collection

In general, there are three ways to improve/refine the presented copying collection approach:

- Improving the traversal strategy
- Considering the age of the objects
- Incremental procedures

Improving the graph traversal strategy

Breadth-first search has the advantage that no stack is required/that pointer reversal is unnecessary.

In order to achieve locality between an object x and an object referenced by x, breadth-first traversal can be combined with a limited depth-first traversal.

Generational collection

The life time of objects is very different:

- Most objects do not get old.
- Objects of a certain age get older with high probability.

The procedure presented above copies old objects over and over again.

In a generational collection approach, the heap is separated into generations, e.g., young, middle, old.
The young generation is prefered for garbage collection.

Incremental collection

For interactive programs and for real-time requirements, longer breaks of the execution are not acceptable.
The garbage collection must be performed incrementally. Incremental approaches are in general more complex.

Literature

Recommended reading for garbage collection

- Appel: Chap 13.1 – 13.3, pp. 277 – 301