Lab Week 7

CS4215: Programming Language Implementation

Martin Henz

February 28, 2012

Generated on Tuesday 28 February, 2012, 09:49
1 Heap Memory
   - Memory Allocation for Programs
   - A Heap Memory Model
   - simPLvm with Heap
   - Implementing a Heap

2 Heap Management Techniques
Resources of computing

- Time: accounted for by simPLvm, number of executed instructions
- Memory: not well represented yet, instructions freely construct “things”
Questions

- Does a given data structure have to be stored forever?
- Will a given program run out of memory?
- Can we design a virtual machine that makes effective use of the available memory?
Memory Allocation for Programs

- Static allocation
- Stack allocation
- Heap allocation
Static Allocation

- Assign fixed memory location for every identifier

Limitations

- The size of each data structure must be known at compile-time. For example, arrays whose size depends on function parameters are not possible.
- Recursive functions are not possible, because each recursive call needs its own copy of parameters and local variables.
- Data structures such as closures cannot be created dynamically.
Stack Allocation

- Keep track of information on function invocations on runtime stack
- Recursion possible
- Size of locals can depend on arguments
- Remaining shortcomings:
  - Difficult to manipulate recursive data structures
  - Only objects with known compile time size can be returned from functions
Heap Allocation

- Data structures may be allocated and deallocated in any order
- Complex pointer structures will evolve at runtime
- Management of allocated memory becomes an issue
1 Heap Memory
   - Memory Allocation for Programs
   - A Heap Memory Model
   - simPLvm with Heap
   - Implementing a Heap

2 Heap Management Techniques
A Heap Memory Model

- Nodes: stack frames, operand stacks, environments etc
- Edges: references between nodes
- Labels on edges
- Nodes can point to primitive values
Formal Model of Heap

A heap is a pair \((V, E)\), where

\[ E \subseteq \{(v, f, w) | v \in V, f \in \text{LS + Int}, w \in V + \text{PV}\} \]

Edge \((v, f, w)\) has source \(v\), label \(f\) and target \(w\).
Edges are functional in the first two arguments.
Operations on Heaps

\[
\text{newnode}((V, E)) = (v, (V \cup \{v\}, E))
\]
where \( v \) is chosen new; \( v \not\in V \)

\[
\text{update}(v, f, w, (V, E)) = (V, (v, f, w) \cup (E - \{(v, f, w')|w' \in W}\})
\]
Operations on Heaps (continued)

\[
\text{children}(v, (V, E)) = \{ w \in W \mid (v, f, w) \in E \text{ for some } f \in L \}
\]

\[
\text{nodechildren}(v, (V, E)) = \{ w \in V \mid (v, f, w) \in E \text{ for some } f \in L \}
\]

\[
\text{labels}(v, (V, E)) = \{ f \in L \mid (v, f, w) \in E \text{ for some } w \in W \}
\]

\[
\text{deref}(v, f, (V, E)) = w, \text{ where } (v, f, w) \in E
\]
Operations on Heaps (continued)

\[
\text{copy}(v, (V, E)) = (v', (V \cup \{v'\}, E \cup \{(v', f_1, \text{deref}(v, f_1, (V, E))), \ldots, (v', f_n, \text{deref}(v, f_n, (V, E)))\})
\]

where \(\{f_1, \ldots, f_n\} = \text{labels}(v, (V, E))\)
Operations on Heaps (continued)

\[
\text{newstack}(h) = (v, h'')
\]

where \((v, h') = \text{newnode}(h)\),
and \(h'' = \text{update}(v, \text{size}, 0, h')\)

\[
\text{push}(v, w, h) = h''
\]

where \(s = \text{deref}(v, \text{size}, h)\),
\(h' = \text{update}(v, \text{size}, s + 1, h)\),
and \(h'' = \text{update}(v, s, w, h')\)

\[
\text{pop}(v, h) = (w, h')
\]

where \(s = \text{deref}(v, \text{size}, h)\),
\(h' = \text{update}(v, \text{size}, s - 1, h)\),
and \(w = \text{deref}(v, s - 1, h')\)
1 Heap Memory
   • Memory Allocation for Programs
   • A Heap Memory Model
   • simPLvm with Heap
   • Implementing a Heap

2 Heap Management Techniques
We start the machine with a state of the form \((os_0, 0, e_0, rs_0, h_0)\), where

\[
\begin{align*}
(os_0, h') &= \text{newstack}\((\emptyset, \emptyset)\) \\
(e_0, h'') &= \text{newnode}(h') \\
(rs_0, h_0) &= \text{newstack}(h'')
\end{align*}
\]
Rules of simPLvm with Heap

\[ s(pc) = \text{LDCI } i \]

\[ (os, pc, e, rs, h) \Rightarrow_s (os, pc + 1, e, rs, h') \]

\[ h' = \text{push}(os, i, h) \]
Rules of simPLvm with Heap (continued)

\[ s(pc) = \text{PLUS} \]

\[
(os, pc, e, rs, h) \xrightarrow{s} (os, pc + 1, e, rs, h''')
\]

where

\[
(i_2, h') = \text{pop}(os, h)
\]

\[
(i_1, h'') = \text{pop}(os, h')
\]

\[
h''' = \text{push}(os, i_1 + i_2, h'')
\]
Rules of simPLvm with Heap (continued)

\[ s(pc) = \text{GOTOR } i \]

\[ (os, pc, e, rs, h) \xrightarrow{s} (os, pc + i, e, rs, h) \]

\[ s(pc) = \text{JOFR } i \]

\[ (os, pc, e, rs, h) \xrightarrow{s} (os, pc + 1, e, rs, h') \]

if \((t, h') = \text{pop}(os, h)\)
Rules of simPLvm with Heap (continued)

\[ s(pc) = \text{LDS} \times \]

\[
(\text{os}, pc, e, rs, h) \Rightarrow_s (\text{os}, pc + 1, e, rs, h')
\]

\[ h' = \text{push}(\text{os}, \text{deref}(e, x, h), h) \]
Rules of simPLvm with Heap (continued)

\[ s(pc) = \text{LDFS } x_1 \cdots x_n \]

\[
(os, pc, e, rs, h) \xrightarrow{s} (os, pc + 1, e, rs, h')
\]

where

\[
(c, h^{(1)}) = \text{newnode}(h)
\]
\[
(f, h^{(2)}) = \text{newnode}(h^{(1)})
\]
\[
h^{(3)} = \text{update}(c, \text{address}, pc + 2, h^{(2)})
\]
\[
h^{(4)} = \text{update}(c, \text{formals}, f, h^{(3)})
\]
\[
h^{(5)} = \text{update}(c, \text{environment}, e, h^{(4)})
\]
\[
h^{(6)} = \text{push}(os, c, h^{(5)})
\]
\[
h^{(6+i)} = \text{update}(f, i, x_i, h^{(6+i-1)}), \text{ where } 1 \leq i \leq n
\]
\[
h' = h^{(6+n)}
\]
Rules (continued)

\[ s(pc) = \text{CALL } n \]

\[
(os, pc, e, rs, h) \xrightarrow{s} (os', a, e', rs, h')
\]

\[
(v_{n-i+1}, h^{(i)}) = \text{pop}(os, h^{(i-1)}), \text{ where } 1 \leq i \leq n
\]

\[
h^{(n+i)} = \text{update}(e', \text{deref}(f, i, h^{(n+i-1)}), v_i, h^{(n+i-1)}),
\]

\[
\text{where } 1 \leq i \leq n
\]

\[
(c, h^{(2n+1)}) = \text{pop}(os, h^{(2n)})
\]

\[
a = \text{deref}(c, \text{address}, h^{(2n+1)})
\]

\[
f = \text{deref}(c, \text{formals}, h^{(2n+1)})
\]

\[
(e', h^{(2n+2)}) = \text{copy}(\text{deref}(c, \text{environment}, h^{(2n+1)}), h^{(2n+1)})
\]

\[
(sf, h^{(2n+3)}) = \text{newnode}(h^{(2n+2)})
\]

\[\ldots\]
Rules (continued)

\[ s(pc) = \text{CALL } n \]

\[
(os, pc, e, rs, h) \xrightarrow{s} (os', a, e', rs, h')
\]

\[
(v_{n-i+1}, h^{(i)}) = \text{pop}(os, h^{(i-1)}), \text{ where } 1 \leq i \leq n
\]

\[
h^{(n+i)} = \text{update}(e', \text{deref}(f, i, h^{(n+i-1)}), v_i, h^{(n+i-1)}),
\]

\[ h^{(n+i)} = \text{update}(e', \text{deref}(f, i, h^{(n+i-1)}), v_i, h^{(n+i-1)}) \]

\[
(e', h^{(2n+2)}) = \text{copy}(\text{deref}(c, \text{environment}, h^{(2n+1)}), h^{(2n+1)})
\]

\[
(sf, h^{(2n+3)}) = \text{newnode}(h^{(2n+2)})
\]

\[
h^{(2n+4)} = \text{update}(sf, pc, pc + 1, h^{(2n+3)})
\]

\[
h^{(2n+5)} = \text{update}(sf, os, os, h^{(2n+4)})
\]

\[
h^{(2n+6)} = \text{update}(sf, e, e, h^{(2n+5)})
\]
Memory Consumption of Instructions

We count the number of nodes and edges created by each instruction.

Example: \texttt{LDFS x y z} creates

- 2 nodes: \( c, f \),
- 7 edges: 3 leaving \( c \), 1 leaving \( os \), and 3 leaving \( f \).
Questions

- How realistic is this graph view of a heap?
- Once a node is created, will it have to be stored until the end of the program execution?
Memory Management

- \( V = V_{useful} \cup V_{useless} \)
- Is there an algorithm to compute \( V_{useful} \) and \( V_{useless} \)?
  - No, undecidable! :-|
- Idea: Approximate \( V_{useful} \) and \( V_{useless} \) by
  - \( V_{live} \supseteq V_{useful} \)
  - \( V_{dead} \subseteq V_{useless} \)
1 Heap Memory
   • Memory Allocation for Programs
   • A Heap Memory Model
   • simPLvm with Heap
   • Implementing a Heap

2 Heap Management Techniques
Implementing the Heap

static int HEAPBOTTOM = 0;    // smallest heap address
static int HEAPSIZE = 1000000; // size of heap is fixed
static int[] HEAP = new int[HEAPSIZE];
Example of Node

Stack frame \((os, 400, e)\)

\[
\begin{align*}
\text{HEAP}[120] & = \ldots \quad \text{// write book keeping information} \\
& \quad \ldots \\
\text{HEAP}[124] & = \ldots \quad \text{// write book keeping information} \\
\text{HEAP}[125] & = 100; \quad \text{// save address of operand stack} \\
\text{HEAP}[126] & = 400; \quad \text{// save pc} \\
\text{HEAP}[127] & = 110; \quad \text{// save address of environment}
\end{align*}
\]
Mutator Operations

```c
int newnode(int size);
void update(int v, int f, int w);
```
Heap Memory

Heap Management Techniques

- Reference Counting
- Mark-Sweep Garbage Collection
- Copying Garbage Collection
- Background
Reference Counting

\[ V_{\text{dead}} = \{ w \in V \mid \text{there is no } f \in L, v \in V, \text{ s.t. } (v, f, w) \in E \} \]

Every update operation identifies all new elements of \( V_{\text{dead}} \) and makes them available for future \textit{newnode} operations.
static int NEXT = 1; // field 1 keeps the next pointer
static int RC = 1;    // field 1 keeps the reference count
static int Freelist = HEAPBOTTOM;
int current = HEAPBOTTOM;
while (current+NODESIZE < heapsize) {
    heap[current+NEXT] = current+NODESIZE;
    current = current + NODESIZE;
}
heap[current+NEXT] = NIL;
Allocating a New Node

```c
int allocate() {
    int newnode = freelist;
    freelist = heap[freelist+next];
    return newnode;
}

int newnode() {
    if (freelist == NIL) abort("Memory exhausted");
    int newnode = allocate();
    heap[newnode+RC] = 1;
    return newnode;
}
```
Updating an Edge

```c
void free(int n) {
    heap[n+NEXT] = freelist;
    freelist = n;
}

void delete(int n) {
    heap[n+RC] = heap[n+RC] - 1;
    if (heap[n+RC] == 0) {
        for c in children(n) do delete(heap[n+c]);
        free(n);
    }
}

void update(int v,int f,int w) {
    delete(heap[v+f]);
    heap[w+RC] = heap[w+RC] + 1;
    heap[v+f] = w;
}
```
Adavantages of Reference Counting

- Incrementality
- Locality
- Immediate reuse
Disadvantages of Reference Counting

- Runtime overhead
- Cannot reclaim cyclic data structures
Garbage Collectors

When `newnode()` runs out of memory, a garbage collector computes a set $V_{dead}$ and reclaims the memory its elements were occupying.

`update()` not affected by GC:

```c
void update(int v, int f, int w) {
    heap[v+f] = w;
}
```
Mark-Sweep

```c
int newnode() {
    if (freelist == NIL) mark_sweep();
    int newnode = allocate();
    return newnode;
}
```
Liveness

\[ \exists f. (v_1, f, v_2) \in E \]

\[ v_1 \rightarrow v_2 \]

\[ v \rightarrow^* v \]

\[ v_1 \rightarrow^* v_2 \]

\[ v_1 \rightarrow^* v_3 \]
Liveness (continued)

The set $V_{\text{live}}$ of a machine in state $(os, pc, e, rs, (V, E))$ is now defined as follows:

$$V_{\text{live}} = \{ v \in V | r \rightarrow^* v, \text{ where } r \in \{os, e, rs\} \}$$

{$os, e, rs$} are called roots.
Visit all nodes in $V_{live}$ and MARK them.
Visit every node in the heap and free every UNMARKED node.
Mark-Sweep

void mark_sweep() {
    for r in Roots
        mark(r);
    sweep();
    if (Freelist == NIL) abort("memory exhausted");
}

void mark(v) {
    if (HEAP[v+MARKBIT] == UNMARKED) {
        HEAP[v+MARKBIT] = MARKED;
        for (int c = FIRSTCHILD, c <= LASTCHILD, c++) {
            mark(HEAP[v+c]);
        }
    }
}
void sweep() {
    int v = HEAPBOTTOM;
    while (v < HEAPTOP) {
        if (HEAP[v+MARKBIT] == UNMARKED) free(v);
        else HEAP[v+MARKBIT] = UNMARKED;
        v = v + NODESIZE;
    }
}
Performance

\[ e_{MS} = \frac{m_{MS}}{t_{MS}} \]

\( m_{MS} \): amount of reclaimed memory  
\( t_{MS} \): time taken  
\( M = |V| = \text{HEAPSIZE}/\text{NODESIZE} \).  
\( R \): the number of live nodes.  
\( r = R/M \): residency
Performance (continued)

\[ m_{MS} = M - R \]
\[ t_{MS} = a \cdot R + b \cdot M \]
\[ e_{MS} = \frac{m_{MS}}{t_{MS}} = \frac{M - R}{aR + bM} = \frac{1 - r}{ar + b} \]
Idea

- Use only half of the available memory for allocating nodes
- Once this half is filled up, copy the live memory contained in the first half to the second half
- Reverse the roles of the halves and continue.
void init() {
    Tospace = HEAPBOTTOM;
    SPACESIZE = HEAPSIZE / 2;
    Topofspace = Tospace + SPACESIZE - 1;
    Fromspace = Topofspace + 1;
    Free = Tospace;
}
### Allocating New Nodes

```c
int newnode() {
    if (Free + NODESIZE > Topofspace)
        flip();
    if (Free + NODESIZE > Topofspace)
        abort("memory exhausted");
    int newnode = Free;
    Free = Free + NODESIZE;
    return newnode;
}
```
Cheney’s Algorithm

```c
void flip() {
    int temp = Fromspace;
    Fromspace = Tospace; Tospace = temp;
    Topofspace = Tospace + SPACESIZE - 1;
    int scan = Tospace; Free = Tospace;
    for r in Roots
        r = copy(r);
    while (scan < Free) {
        for (int c = FIRSTCHILD, c <= LASTCHILD, c++) {
            HEAP[scan+c] = copy(HEAP[scan+c]);
            scan = scan + NODESIZE;
        }
    }
}
```
Cheney’s Algorithm (continued)

```c
int copy(v) {
    if (moved_already(v))
        return HEAP[v+FORWARDINGADDRESS];
    else {
        int addr = free
        move(v,free);
        free = free + NODESIZE;
        HEAP[v+FORWARDINGADDRESS] = addr;
        return addr;
    }
}
```
**Performance**

\[
m_{\text{Copy}} = \frac{M}{2} - R
\]

\[
t_{\text{Copy}} = c \cdot R
\]

\[
e_{\text{Copy}} = \frac{m_{\text{Copy}}}{t_{\text{Copy}}} = \frac{\frac{M}{2} - R}{cR} = \frac{1}{2cr} - \frac{1}{c}
\]
Historical Background

- Pioneered by LISP implementations
- Reference counting: Gelernter et al 1960, Collins 1960, used in Smalltalk, and Modula-2+
- Mark-sweep: McCarthy 1960, widely used, e.g. JVM
- Minsky 1963, Cheney 1970, widely used in functional and logic programming
Explicit Heap Deallocation

```haskell
var p : ^t
p := nil
new(p)
dispose(p)
```
Space Leak

new(p);
p := nil
Dangling Reference

a.s := p;
Dispose(p)
Space leaks can occur even in systems with automatic memory management.
Large systems implement their own memory management (Unix, Adobe Photoshop).
The language rePL—Adding data structures to simPL
- Denotational semantics
- Pass-by-name and pass-by-need