06—A Low-Level Virtual Machine

CS4215: Programming Language Implmentation

Martin Henz

February 17, 2012

Generated on Thursday 16 February, 2012, 17:34

1. A Virtual Machine for simPL (continued)
   - simPLd: Getting Serious
   - simPLe: Getting Recursive
2. Heap Memory
3. Heap Management Techniques

simPLd: Functions

$$\frac{}{x} \qquad \frac{E}{\texttt{fun}\ x_1 \cdots x_n\ \texttt{->}\ E\ \texttt{end}} \qquad \frac{E \quad E_1 \quad \cdots \quad E_n}{(E\ E_1 \cdots E_n)}$$

Outline

- Compilation of identifiers
- Execution of identifiers
- Compilation of function application
- Compilation of function definition
- Execution of function definition
- Execution of function application
- Returning from a function

Compilation of Identifiers

- add register e (environment), mapping identifiers to denotable values.
- translation:

$$\frac{}{x \hookrightarrow \texttt{LDS}\ x}$$

Execution of Identifiers

$$\frac{s(pc) = \texttt{LDS}\ x}{(os, pc, e, rs) \rightrightarrows_s (e(x).os, pc + 1, e, \texttt{rs})}$$

Compilation of Function Application

$$\frac{E \hookrightarrow s \qquad E_1 \hookrightarrow s_1 \cdots E_n \hookrightarrow s_n}{(E\ E_1 \cdots E_n) \hookrightarrow s.s_1.\ldots.s_n.\mathtt{CALL}\ n}$$

Compilation of Function Definition

$$\frac{E \hookrightarrow s}{\texttt{fun } x_1 \ldots x_n \texttt{ -> } E \texttt{ end} \hookrightarrow \texttt{LDFS } x_1 \cdots x_n.\texttt{GOTOR } |s| + 2.s.\texttt{RTN}}$$

Execution of Function Definition

$$\frac{s(pc) = \texttt{LDFS}\ x_1 \cdots x_n}{(os, pc, e, rs) \rightrightarrows_s ((pc+2, x_1 \cdots x_n, e).os, pc+1, e, rs)}$$

Such a triple $(address, formals, e)$ is called a *closure*.

Execution of Function Application

$$\frac{s(pc) = \texttt{CALL}\ n}{\begin{array}{l}(v_n.\ldots v_1.(address, x_1 \cdots x_n, e').os, pc, e, rs) \rightrightarrows_s \\ (\langle\rangle, address, e'[x_1 \leftarrow v_1] \cdots [x_n \leftarrow v_n], (pc+1, os, e).rs)\end{array}}$$

Returning from a Function

$$\frac{s(pc) = \texttt{RTN}\ n}{(v.os, pc, e, (pc', os', e').rs) \rightrightarrows_s (v.os', pc', e', rs)}$$

The Actual Machine: Example

```
(fun x -> x + 1 end  2)
```

becomes

```
[LDF 4     0
LDCI 2     1
CALL 1     2
DONE       3
LD 0       4
LDCI 1     5
PLUS       6
RTN]       7
```

Execution of Identifiers

```
case OPCODES.LD:        os.push(e.elementAt(i.INDEX));
                        pc++;
                        break;
```

Representation of Closures

```
public class Closure implements Value {
  public Environment environment;
  public int ADDRESS;
  Closure(Environment e, int a) {
    environment = e;
    ADDRESS = a;
  }
}
```

Execution of Function Definition

```
case OPCODES.LDF:   Environment env = e;
                    os.push(new Closure(env,i.ADDRESS));
                    pc++;
                    break;
```

Representing Runtime Stack Frames

```
public class StackFrame {
  public int pc;
  public Environment environment;
  public Stack operandStack;
  public StackFrame(int p, Environment e, Stack os) {
     pc = p;
     environment = e;
     operandStack = os;
  }
}
```

Execution of Application

```
case OPCODES.CALL: {int n = i.NUMBEROFARGUMENTS;
                    Closure closure
                      = os.elementAt(os.size()-n-1);
                    Environment newEnv
                      = closure.environment.extend(n);
                    int s = newEnv.size();
                    for (int j = s-1; j >= s-n; --j)
                      newEnv.setElementAt(os.pop(),j);
                    os.pop(); // function value
                    rs.push(new StackFrame(pc+1,e,os));
                    pc = closure.ADDRESS;
                    e = newEnv; os = new Stack();
                    break;}
```

Returning from a Function

```
case OPCODES.RTN:      Value returnValue = os.pop();
                       StackFrame f = rs.pop();
                       pc = f.pc;
                       e = f.environment;
                       os = f.operandStack;
                       os.push(returnValue);
                       break;
```

Translation of simPLe

$$\frac{E \hookrightarrow s}{\texttt{recfun}\ f\ x_1 \ldots x_n\ \texttt{->}\ E\ \texttt{end} \hookrightarrow \\ \texttt{LDRFS}\ f\ x_1\ \cdots x_n.\texttt{GOTOR}\ |s| + 2.s.\texttt{RTN}}$$

Execution of SVMLe

$$\frac{s(pc) = \texttt{LDRFS}\ f\ x_1 \cdots x_n}{(os, pc, e, rs) \rightrightarrows_s ((pc + 2, f, x_1 \cdots x_n, e).os, pc + 1, e, rs)}$$

Execution of SVMLe, continued

$$\frac{s(pc) = \texttt{CALL}\ n}{\begin{array}{l}(v_n.\ldots.v_1.(address, f, x_1 \cdots x_n, e').os, pc, e, rs) \rightrightarrows_s \\ (\langle\rangle, address, \\ e'[f \leftarrow (address, f, x_1 \cdots x_n, e')] \\ [x_1 \leftarrow v_1] \cdots [x_n \leftarrow v_n], \\ (pc+1, os, e).rs)\end{array}}$$

Idea of Implementation

Create a circular data structure:
Environment of recursive function value points to function itself.

Implementation of SVMLe

```
case OPCODES.LDRF:     Environment envr = e.extend(1);
                       Value fv = new
                              Closure(envr,i.ADDRESS);
                       envr.setElementAt(fv,e.size());
                       os.push(fv);
                       pc++;
                       break;
```

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 alloation

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. A Virtual Machine for simPL (continued)

2. Heap Memory
   - Memory Allocation for Programs
   - A Heap Memory Model
   - simPLvm with Heap
   - Implementing a Heap

3. 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

$$E \subseteq \{(v, f, w) | v \in V, f \in \mathbf{LS} + \mathbf{Int}, w \in V + \mathbf{PV}\}$$

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

Operations on Heaps

$$
\begin{aligned}
\mathit{newnode}((V,E)) &= (v,(V\cup\{v\},E)) \\
&\quad \text{where } v \text{ is chosen new; } v\notin V
\end{aligned}
$$

$$\mathit{update}(v,f,w,(V,E)) = (V,(v,f,w)\cup(E-\{(v,f,w')|w'\in W\})$$

Operations on Heaps (continued)

$$
\begin{aligned}
\mathit{children}(v, (V, E)) &= \{w \in W \mid (v, f, w) \in E \text{ for some } f \in L\} \\
\mathit{nodechildren}(v, (V, E)) &= \{w \in V \mid (v, f, w) \in E \text{ for some } f \in L\} \\
\mathit{labels}(v, (V, E)) &= \{f \in L \mid (v, f, w) \in E \text{ for some } w \in W\} \\
\mathit{deref}(v, f, (V, E)) &= w, \text{ where } (v, f, w) \in E
\end{aligned}
$$

Operations on Heaps (continued)

$$
\begin{array}{rcl}
\mathit{copy}(v,(V,E)) & = & (v',(V\cup\{v'\},E\cup\{(v',f_1,\mathit{deref}(v,f_1,(V,E))),\ldots,\\
& & \qquad\qquad\qquad\qquad (v',f_n,\mathit{deref}(v,f_n,(V,E)))\}))\\
& & \text{where } \{f_1,\ldots,f_n\} = \mathit{labels}(v,(V,E))
\end{array}
$$

Operations on Heaps (continued)

$$
\begin{aligned}
\mathit{newstack}(h) &= (v, h'') \\
& \text{where } (v, h') = \mathit{newnode}(h), \\
& \text{and } h'' = \mathit{update}(v, \texttt{size}, 0, h') \\
\mathit{push}(v, w, h) &= h'' \\
& \text{where } s = \mathit{deref}(v, \texttt{size}, h), \\
& h' = \mathit{update}(v, \texttt{size}, s + 1, h), \\
& \text{and } h'' = \mathit{update}(v, s, w, h') \\
\mathit{pop}(v, h) &= (w, h') \\
& \text{where } s = \mathit{deref}(v, \texttt{size}, h), \\
& h' = \mathit{update}(v, \texttt{size}, s - 1, h), \\
& \text{and } w = \mathit{deref}(v, s - 1, h')
\end{aligned}
$$

1. A Virtual Machine for simPL (continued)

2. Heap Memory
   - Memory Allocation for Programs
   - A Heap Memory Model
   - simPLvm with Heap
   - Implementing a Heap

3. Heap Management Techniques

Starting State of simPLvm with Heap

We start the machine with a state of the form $(os_0, 0, e_0, rs_0, h_0)$, where

$$
\begin{aligned}
(os_0, h') &= newstack((\emptyset, \emptyset)) \\
(e_0, h'') &= newnode(h') \\
(rs_0, h_0) &= newstack(h'')
\end{aligned}
$$

Rules of simPLvm with Heap

$$\frac{s(pc) = \texttt{LDCI}\ i}{(os, pc, e, rs, h) \rightrightarrows_s (os, pc+1, e, rs, h')} \quad h' = push(os, i, h)$$

Rules of simPLvm with Heap (continued)

$$\frac{s(pc) = \mathrm{PLUS}}{(os, pc, e, rs, h) \rightrightarrows_s (os, pc+1, e, rs, h''')}$$

where

$$\begin{array}{rcl}
(i_2, h') & = & pop(os, h) \\
(i_1, h'') & = & pop(os, h') \\
h''' & = & push(os, i_1 + i_2, h'')
\end{array}$$

Rules of simPLvm with Heap (continued)

$$\frac{s(pc) = \texttt{GOTOR}\ i}{(os, pc, e, rs, h) \rightrightarrows_s (os, pc + i, e, rs, h)}$$

$$\frac{s(pc) = \texttt{JOFR}\ i}{(os, pc, e, rs, h) \rightrightarrows_s (os, pc + 1, e, rs, h')} \quad \text{if } (t, h') = pop(os, h)$$

Rules of simPLvm with Heap (continued)

$$\frac{s(pc) = \texttt{LDS}\ x}{(os, pc, e, rs, h) \rightrightarrows_s (os, pc + 1, e, rs, h')}$$

$$h' = push(os, deref(e, x, h), h)$$

Rules of simPLvm with Heap (continued)

$$\frac{s(pc) = \texttt{LDFS}\ x_1 \cdots x_n}{(os, pc, e, rs, h) \rightrightarrows_s (os, pc+1, e, rs, h')}$$

where

$$\begin{array}{rcl}
(c, h^{(1)}) & = & newnode(h) \\
(f, h^{(2)}) & = & newnode(h^{(1)}) \\
h^{(3)} & = & update(c, \texttt{address}, pc+2, h^{(2)}) \\
h^{(4)} & = & update(c, \texttt{formals}, f, h^{(3)}) \\
h^{(5)} & = & update(c, \texttt{environment}, e, h^{(4)}) \\
h^{(6)} & = & push(os, c, h^{(5)}) \\
h^{(6+i)} & = & update(f, i, x_i, h^{(6+i-1)}), \text{ where } 1 \leq i \leq n \\
h' & = & h^{(6+n)}
\end{array}$$

Rules (continued)

$$s(pc) = \texttt{CALL}\ n \ / \ (os, pc, e, rs, h) \rightrightarrows_s (os', a, e', rs, h')$$

$$\begin{array}{rcl}
(v_{n-i+1}, h^{(i)}) & = & pop(os, h^{(i-1)}),\ \text{where } 1 \leq i \leq n \\
h^{(n+i)} & = & update(e', deref(f, i, h^{(n+i-1)}), v_i, h^{(n+i-1)}) \\
(c, h^{(2n+1)}) & = & pop(os, h^{(2n)}) \\
a/f & = & deref(c, \texttt{address}/\texttt{formals}, h^{(2n+1)}) \\
(e', h^{(2n+2)}) & = & copy(deref(c, \texttt{environment}, h^{(2n+1)}), h^{(2n+1)}) \\
(sf, h^{(2n+3)}) & = & newnode(h^{(2n+2)}) \\
h^{(2n+4)} & = & update(sf, \texttt{pc}, pc + 1, h^{(2n+3)}) \\
h^{(2n+5)} & = & update(sf, \texttt{os}, os, h^{(2n+4)}) \\
h^{(2n+6)} & = & update(sf, \texttt{e}, e, h^{(2n+5)}) \\
(os', h^{(2n+7)}) & = & newstack(h^{(2n+6)}) \\
h' & = & push(rs, sf, h^{(2n+7)})
\end{array}$$

Memory Consumption of Instructions

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

Example: `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
  $V_{live} \supseteq V_{useful}$
  $V_{dead} \subseteq V_{useless}$

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 $(os, 400, e)$

```
HEAP[120] = ....   // write book keeping information
...
HEAP[124] = ....   // write book keeping information
HEAP[125] = 100;   // save address of operand stack
HEAP[126] = 400;   // save pc
HEAP[127] = 110;   // save address of environment
```

Mutator Operations

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

Reference Counting

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

Every update operation identifies all new elements of V_{dead} and makes them available for future *newnode* operations.

Freelist for Keeping Track of Free Memory

```
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

```
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

```
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:

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

Mark-Sweep

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

Liveness

$$\frac{\exists f.(v_1, f, v_2) \in E}{v_1 \longrightarrow v_2}$$

$$\frac{}{v \longrightarrow^* v} \qquad \frac{v_1 \longrightarrow v_2}{v_1 \longrightarrow^* v_2} \qquad \frac{v_1 \longrightarrow^* v_2 \qquad v_2 \longrightarrow^* v_3}{v_1 \longrightarrow^* v_3}$$

Liveness (continued)

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

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

$\{os, e, rs\}$ are called roots.

Idea

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]);
   }
}
```

Mark-Sweep (continued)

```
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
$M = |V| = \texttt{HEAPSIZE}/\texttt{NODESIZE}$.
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.

Initialization

```
void init() {
   Tospace = HEAPBOTTOM;
   SPACESIZE = HEAPSIZE / 2;
   Topofspace = Tospace + SPACESIZE - 1;
   Fromspace = Topofspace + 1;
   Free = Tospace;
}
```

Allocating New Nodes

```
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

```
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)

```
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

$$m_{Copy} = \frac{M}{2} - R$$

$$t_{Copy} = c \cdot R$$

$$e_{Copy} = \frac{m_{Copy}}{t_{Copy}} = \frac{\frac{M}{2} - R}{cR} = \frac{1}{2cr} - \frac{1}{c}$$

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

```
var p : ^t
p := nil

new(p)

dispose(p)
```

Space Leak

```
new(p);
p := nil
```

Dangling Reference

```
a.s := p;
dispose(p)
```

Memory Management in Software Systems

Space leaks can occur even in systems with automatic memory management.

Large systems implement their own memory management (Unix, Adobe Photoshop).

Next Lecture (after recess)

- Midterm
- Tail call optimization
- The language rePL—Adding data structures to simPL
- Denotational semantics of rePL