06—A Low-Level Virtual Machine

CS4215: Programming Language Implementation

Martin Henz

February 17, 2012

Generated on Thursday 16 February, 2012, 17:50
1 A Virtual Machine for simPL (continued)
   • simPLd: Getting Serious
   • simPLE: Getting Recursive

2 Heap Memory

3 Heap Management Techniques
simPLd: Functions

\[
\begin{array}{c}
\text{fun } x_1 \cdots x_n \rightarrow E \text{ end} \\
\end{array}
\]
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: 

  \[ x \mapsto \text{LDS } x \]
Execution of Identifiers

\[ s(pc) = LDS \ x \]

\[ \text{(os, pc, e, rs)} \implies_s (e(x).os, pc + 1, e, rs) \]
Compilation of Function Application

\[ E \rightarrow s \quad E_1 \rightarrow s_1 \cdots E_n \rightarrow s_n \]

\[ (E \; E_1 \cdots E_n) \rightarrow s.s_1 \cdots s_n.CALL \; n \]
Compilation of Function Definition

\[ E \leftrightarrow s \]

\[ \text{fun } x_1 \ldots x_n \rightarrow E \text{ end } \leftrightarrow \text{LDFS } x_1 \ldots x_n.\text{GOTOR } |s| + 2.s.\text{RTN} \]
Execution of Function Definition

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

\[(os, pc, e, rs) \xrightarrow{s} ((pc + 2, x_1 \cdots x_n, e).os, pc + 1, e, rs)\]

Such a triple \(\text{(address, formals, e)}\) is called a \textit{closure}. 

Execution of Function Application

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

\[
(v_n, \ldots, v_1, (address, x_1, \ldots, x_n, e').os, pc, e, rs) \Rightarrow_s

(\langle \rangle, address, e'[x_1 \leftarrow v_1] \ldots [x_n \leftarrow v_n], (pc + 1, os, e).rs)
\]
Returning from a Function

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

\[
(v.os, pc, e, (pc', os', e').rs) \Rightarrow_s (v.os', pc', e', rs)
\]
The Actual Machine: Example

(fun x -> x + 1 end 2)

becomes

\[
\begin{array}{c}
\text{[LDF 4 0}} \\
\text{LDCI 2 1}} \\
\text{CALL 1 2}} \\
\text{DONE 3}} \\
\text{LD 0 4}} \\
\text{LDCI 1 5}} \\
\text{PLUS 6}} \\
\text{RTN] 7}}
\end{array}
\]
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

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);  
brake;
A Virtual Machine for simPL (continued)

- simPLd: Getting Serious
- simPLe: Getting Recursive

2 Heap Memory

3 Heap Management Techniques
Translation of simPLe

\[ E \leftarrow s \]

\[
\text{recfun } f \ x_1 \ldots x_n \rightarrow E \text{ end } \leftarrow \\
\text{LDRFS } f \ x_1 \ldots x_n.\text{GOTOR } |s| + 2.s.\text{RTN}
\]
Execution of SVMLe

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

\[
(os, pc, e, rs) \Rightarrow_s ((pc + 2, f, x_1 \cdots x_n, e).os, pc + 1, e, rs)
\]
Execution of SVMLe, continued

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

\( (v_n \ldots v_1.(address, f, x_1 \cdots x_n, e').os, pc, e, rs) \Rightarrow_s \)

\( (\langle \rangle, address, e'[f \leftarrow (address, f, x_1 \cdots x_n, e')]) \]

\[ [x_1 \leftarrow v_1] \cdots [x_n \leftarrow v_n], \]

\( (pc + 1, os, e).rs) \)
Idea of Implementation

Create a circular data structure:
Environment of recursive function value points to function itself.
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;
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
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
A Virtual Machine for simPL (continued)

Heap Memory

Heap Management Techniques

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

Heap Memory

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

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

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

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)

\[ children(v, (V, E)) = \{ w \in W \mid (v, f, w) \in E \text{ for some } f \in L \} \]
\[ nodechildren(v, (V, E)) = \{ w \in V \mid (v, f, w) \in E \text{ for some } f \in L \} \]
\[ labels(v, (V, E)) = \{ f \in L \mid (v, f, w) \in E \text{ for some } w \in W \} \]
\[ deref(v, f, (V, E)) = w, \text{ where } (v, f, w) \in E \]
Operations on Heaps (continued)

\[
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 \( \{ f_1, \ldots, f_n \} = \text{labels}(v, (V, E)) \)
A Virtual Machine for simPL (continued)

Heap Memory

Heap Management Techniques

Memory Allocation for Programs

A Heap Memory Model

simPLvm with Heap

Implementing a Heap

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 \(w = \text{deref}(v, s - 1, h')\)
A Virtual Machine for simPL (continued)

Heap Memory

Memory Allocation for Programs
A Heap Memory Model
simPLvm with Heap
Implementing a Heap

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

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

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

\[
\begin{array}{c}
\text{(os, pc, e, rs, h)} \Rightarrow_s (\text{os, pc + 1, e, rs, h'}) \\
\text{h' = push(os, i, h)}
\end{array}
\]
Rules of simPLvm with Heap (continued)

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

\[
\begin{align*}
(os, pc, e, rs, h) \Rightarrow_s (os, pc + 1, e, rs, h''')
\end{align*}
\]

where

\[
\begin{align*}
(i_2, h') &= \text{pop}(os, h) \\
(i_1, h'') &= \text{pop}(os, h') \\
\quad h''' &= \text{push}(os, i_1 + i_2, h'')
\end{align*}
\]
Rules of simPLvm with Heap (continued)

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

\[
\begin{align*}
(os, pc, e, rs, h) &\Rightarrow_s (os, pc + i, e, rs, h) \\
(s(pc) = \text{JOFR } i &\quad \text{ if } (t, h') = \text{pop}(os, h) \\
(os, pc, e, rs, h) &\Rightarrow_s (os, pc + 1, e, rs, h')
\end{align*}
\]
Rules of simPLvm with Heap (continued)

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

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

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

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

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

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)

\[ s(pc) = \text{CALL } n / (os, pc, e, rs, h) \Rightarrow_s (os', a, e', rs, h') \]

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

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

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

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

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

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

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

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

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

\[ (os', h^{(2n+7)}) = \text{newstack}(h^{(2n+6)}) \]

\[ h' = \text{push}(rs, sf, h^{(2n+7)}) \]
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?
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. 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
Implementing the Heap

```java
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 \((os, 400, e)\)

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

```c
int newnode(int size);
void update(int v, int f, int w);
```
A Virtual Machine for simPL (continued)

Heap Memory

Heap Management Techniques

- Reference Counting
- Mark-Sweep Garbage Collection
- Copying Garbage Collection
- Assessment
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.
Freelist for Keeping Track of Free Memory

```c
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;
}
```
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

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

\[ v_1 \rightarrow v_2 \]

\[ v \rightarrow^* v \]

\[ v_1 \rightarrow^* v_2 \]

\[ v_1 \rightarrow^* v_3 \]

\[ v_1 \rightarrow^* v_2 \]

\[ v_2 \rightarrow^* v_3 \]
Liveness (continued)

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

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

$\{os, e, rs\}$ 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

```c
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)

```c
void sweep() {
    int v = HEAPBOTTOM;
    while (v < HEAPTOP) {
        if (HEAP[v+MARKBIT] == UNMARKED) free(v);
        else HEAP[v+MARKBIT] = UNMARKED;
        v = v + NODESIZE;
    }
}
```
A Virtual Machine for simPL (continued)
Heap Memory
Heap Management Techniques

Reference Counting
Mark-Sweep Garbage Collection
Copying Garbage Collection
Assessment

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.
Initialization

```c
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

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

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

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

\[
\begin{align*}
\text{var } & \quad p : \bowtie t \\
p & \quad := \text{nil} \\
\text{new}(p) \\
\text{dispose}(p)
\end{align*}
\]
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