1. A Virtual Machine for simPL (continued)
   - simPLd: Getting Serious
   - simPLe: Getting Recursive

2. Heap Memory

3. Heap Management Techniques
simPLd: Functions

\[ \text{fun } x_1 \cdots x_n \rightarrow E \text{ end} \]

\[ (E \quad E_1 \cdots E_n) \]
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: 
  \[ x \mapsto \text{LDS} \ x \]
Execution of Identifiers

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

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

\[
E \mapsto s \quad E_1 \mapsto s_1 \ldots E_n \mapsto s_n
\]

\[
(E \ E_1 \ldots E_n) \mapsto s.s_1 \ldots s_n.\text{CALL } n
\]
Compilation of Function Definition

\[ E \rightarrow s \]

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

\[
\begin{align*}
  s(pc) &= \text{LDFS } x_1 \cdots x_n \\
  \hline
  (os, pc, e, rs) &\Rightarrow_s ((pc + 2, x_1 \cdots x_n, e).os, pc + 1, e, rs)
\end{align*}
\]

Such a triple \((address, forms, e)\) is called a \textit{closure}.
Execution of Function Application

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

\[
(v_n \ldots v_1.(\text{address}, x_1 \cdots x_n, e').os, pc, e, rs) \Rightarrow_s s

(\langle \rangle, \text{address}, e'[x_1 \leftarrow v_1] \cdots [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

\[
\begin{align*}
[&\text{LDF } 4 & 0 \\
&\text{LDCI } 2 & 1 \\
&\text{CALL } 1 & 2 \\
&\text{DONE} & 3 \\
&\text{LD } 0 & 4 \\
&\text{LDCI } 1 & 5 \\
&\text{PLUS} & 6 \\
&\text{RTN}] & 7
\end{align*}
\]
Execution of Identifiers

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


case OPCODES.RTN:
    Value returnValue = os.pop();
    StackFrame f = rs.pop();
    pc = f.pc;
    e = f.environment;
    os = f.operandStack;
    os.push(returnValue);
    break;
1. A Virtual Machine for simPL (continued)
   - simPLd: Getting Serious
   - simPLE: Getting Recursive

2. Heap Memory

3. Heap Management Techniques
Translation of simPLe

\[ E \leadsto s \]

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

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

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

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

\[
(v_n \ldots v_1.(address, f, x_1 \ldots x_n, e').os, pc, e, rs) \rightarrow_s \]
\[
(\langle \rangle, address, \]
\[
e'[f \leftarrow (address, f, x_1 \ldots x_n, e')] \]
\[
[x_1 \leftarrow v_1] \cdots [x_n \leftarrow v_n], \]
\[
(pc + 1, os, e).rs) \]
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);
pcrement();
break;
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
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

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

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

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; \( v \notin V \)

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

\[
\begin{align*}
\text{children}(v, (V, E)) &= \{ w \in W \mid (v, f, w) \in E \text{ for some } f \in L \} \\
\text{nodechildren}(v, (V, E)) &= \{ w \in V \mid (v, f, w) \in E \text{ for some } f \in L \} \\
\text{labels}(v, (V, E)) &= \{ f \in L \mid (v, f, w) \in E \text{ for some } w \in W \} \\
\text{deref}(v, f, (V, E)) &= w, \text{ where } (v, f, w) \in E
\end{align*}
\]
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 \(w = \text{deref}(v, s - 1, h')\)
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
We start the machine with a state of the form $(os_0, 0, e_0, rs_0, h_0)$, where

$$(os_0, h') = \text{newstack}((\emptyset, \emptyset))$$

$$(e_0, h'') = \text{newnode}(h')$$

$$(rs_0, h_0) = \text{newstack}(h'')$$
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)

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

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

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)

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

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

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

\[ s(pc) = LDFS \ x_1 \cdots x_n \]
\[ \Rightarrow_s (os, pc + 1, e, rs, h') \]

where

\[ (c, h^{(1)}) = \text{newnode}(h) \]
\[ (f, h^{(2)}) = \text{newnode}(h^{(1)}) \]
\[ h^{(3)} = \text{update}(c, \text{address}, pc + 2, h^{(2)}) \]
\[ h^{(4)} = \text{update}(c, \text{formals}, f, h^{(3)}) \]
\[ h^{(5)} = \text{update}(c, \text{environment}, e, h^{(4)}) \]
\[ h^{(6)} = \text{push}(os, c, h^{(5)}) \]
\[ h^{(6+i)} = \text{update}(f, i, x_i, h^{(6+i-1)}), \text{ where } 1 \leq i \leq n \]
\[ h' = h^{(6+n)} \]
s(pc) = CALL n \ /
(\text{os, pc, e, rs, h}) \Rightarrow_s (\text{os}', a, e', rs, h')
(v_{n-i+1}, h^{(i)}) = \text{pop(os, h}^{(i-1)})\), where 1 \leq i \leq n
h^{(n+i)} = \text{update(e', 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, address/formals, h}^{(2n+1)})
(e', h^{(2n+2)}) = \text{copy(deref(c, environment, h}^{(2n+1)}, h^{(2n+1)})
(sf, h^{(2n+3)}) = \text{newnode(h}^{(2n+2)})
\text{update(sf, pc, pc + 1, h}^{(2n+3)})
\text{update(sf, os, os, h}^{(2n+4)})
\text{update(sf, e, e, h}^{(2n+5)})
(\text{os}', h^{(2n+7)}) = \text{newstack(h}^{(2n+6)})
\text{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: \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
  \( V_{live} \supseteq V_{useful} \)
  \( V_{dead} \subseteq V_{useless} \)
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
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 ($os, 400, e$)

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);
```
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;
}
Advantages 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 $V_{\text{dead}}$ 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;
}
```
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:

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

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

Visit all nodes in $V_{\text{live}}$ 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;
    }
}
```
Performance

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

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

\[
m_{\text{MS}} = M - R
\]

\[
t_{\text{MS}} = a \cdot R + b \cdot M
\]

\[
e_{\text{MS}} = \frac{m_{\text{MS}}}{t_{\text{MS}}} = \frac{M - R}{aR + bM} = \frac{1 - r}{ar + b}
\]
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;
}
```
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;
    }
}
```
Heap Management Techniques

Reference Counting
Mark-Sweep Garbage Collection
Copying Garbage Collection
Assessment

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

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

```c
new(p);
p := nil
```
Dangling Reference

```latex
a.s := p;
\textit{dispose}(p)
```
Space leaks can occur even in systems with automatic memory management. Large systems implement their own memory management (Unix, Adobe Photoshop).
A Virtual Machine for simPL (continued)
Heap Memory
Heap Management Techniques

Next Lecture (after recess)

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