CS4215—Programming Language Implementation

Martin Henz and Chin Wei Ngan

Sunday 8th January, 2017

Chapter 9

Memory Management

9.1 Motivation

Programs running on computers consume two kinds of resources: time and memory.

In the previous chapter, we have managed to introduce a semantic framework that lets us take a realistic account of the runtime of dPL programs. We achieved this by compiling dPL to the virtual machine language sVML. The runtime of the virtual machine is measured as the number of machine instructions that get executed. Even the most complex instruction, `CALL`, can be implemented in a few clock cycles on modern hardware, which indicates that our approach in principle allows to adequately account for actual runtime.

However, the second resource, memory, is not represented so far. In our rules specifying sVM's transition function $\rightrightarrows_s$, we freely construct tuples, such as runtime stack frames, and mappings, such as environments, without regard of the actual realization of these data structures. This makes it hard to account for the memory usage of sVML code. Once a data structure is created, will it have to be stored until the end of the program execution? Will a given program run out of memory? Can we design a virtual machine that makes effective use of the available memory?

Before we address these questions, we discuss general memory management techniques in programming systems in Section 9.2. Section 9.3 introduces a model for heap memory. Using this model, we reformulate the virtual machine of the previous chapter in Section 9.4. Section **??** describes a realization of a heap using actual computer memory. Sections 10.6, 10.7, and 10.8 describe three common memory management techniques, namely reference counting, mark-sweep garbage collection and copying garbage collection. Finally, Section 10.9 gives an overview of the history of memory management and further reading.

9.2 Memory Allocation for Programs

The simplest way to allocate memory used in programs is to assign a fixed memory location for every identifier occurring in the program. Such static allocation was the initial memory allocation technique for the first high-level languages such as FORTRAN. Static allocation limits the set of program constructs of a language in the following ways [**?**]:

- 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.

The advantages of static allocation are speed and safety (since programs cannot run out of space at run-time).

The next memory allocation technique is stack allocation. In the previous chapter, we have seen the usage of a runtime stack to keep track of the operand stack, program counter and environment of functions. With stack allocation, functions can be recursive, because different invocations can operate on different instances of identifiers. The size of local variables such as arrays may depend on a parameter passed to the function, because the stack can grow as much as required. If all memory, including the operands, the environment and data structures such as closures get allocated directly on the runtime stack, the following disadvantages remain [**?**]:

- It is difficult to manipulate data structures since they are tied to the function invocations that created them. In particular, recursive data structures such as list and trees cannot be handled in a natural way.
- Only objects whose size is known at compile time can be returned as the result of a function, because the caller needs to reserve stack space for the return value.

In dPL, stack allocation alone is not sufficient, since functions can be returned as the result of another function. Closures can have different sizes, depending on their environment. Since the compiler cannot know, which function will be returned at runtime, it cannot reserve an adequate amount of memory for the return value to be placed in.

The last memory allocation technique, heap allocation, overcomes all these difficulties. With heap allocation, data structures may be allocated and deallocated in any order. This means that complex pointer structures will evolve at runtime. The down-side of heap allocation is therefore that efficient management of these structures becomes a non-trivial task. Sections 10.6, 10.7, and 10.8 describe three common memory management techniques. Before we can describe them in detail, we introduce a formal heap memory model in the next section.

9.3 A Heap Memory Model

Memory is used to store structured information, such as stack frames, environments, closures, and primitive data such as integers and booleans. We find it convenient to represent this information in form of a mathematical object called *heap*.

We distinguish nodes in the heap, representing entities like stack frames, operand stacks, environments, and closures, from edges, which represent references between these entities. For example, an operand stack can have a reference to a closure. To distinguish the different references from each other, for example the entries of an operand stack, we label the edges with symbols or numbers. Furthermore, we need to represent primitive values such as integers and booleans in the heap, and therefore, we allow edges to point to such values.

More formally, this leads to the following variant of edge-labeled graphs. For a given set of primitive values **PV**, and a set of label symbols **LS**, a heap is a pair (V, E), where V is a set of vertices, and

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

In a given edge (v, f, w), the node v is called the source, f is called the label, and w is called the target. The set of targets $V + \mathbf{PV}$ is denoted by W in the following, and the set of labels $\mathbf{LS} + \mathbf{Int}$ is denoted by L. We require that the ternary relation formed by E is a function in its first two components. That means for any $v \in V$ and $f \in L$ there is at most one $w \in W$ such that $(v, f, w) \in E$. For a given node v the set of targets $\{w | (v, f, w) \in E \text{ for some } f \in L\}$ is called the set of children of v, and the set $\{f | (v, f, w) \in E \text{ for some } w \in W\}$ is called the set of labels of v.

The following operations on heaps will prove useful. The first operation *newnode* constructs a heap where a new node is added.

$$\begin{array}{rcl} \mathit{newnode}((V, E)) & = & (v, (V \cup \{v\}, E)) \\ & & \text{where } v \text{ is chosen new, which means } v \notin V \end{array}$$

Note that this operation returns a pair consisting of a new node and a new heap.

The next operation *update* constructs a heap, where the target of a node and label is changed.

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

Note that this operation preserves the property that E is a function in its first two components.

The operations *newnode* and *update* are two of the operations that are allowed to change the heap. From the point of view of memory management, the running program is a heap changer that executes a sequence of such operations on the heap. These operations are therefore called *mutator* operations (from *mutare*, Latin, "to change").

The following operations return all children, the children that are nodes, all labels and a specific child of a given node, respectively.

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

Note that *deref* is a partial function.

The next mutator operation, *copy*, creates a new node which has the same children under the same labels as a given node.

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

In order to represent stacks in the heap, it is convenient to introduce the following stack-specific mutator operations, which are defined in terms of the mutator operations.

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

9.4 A Memory-aware Virtual Machine for dPL

Now we can reformulate our rules defining the virtual machine for dPL to use a heap h, where $\mathbf{PV} = \mathbf{Int} + \mathbf{Bool} + \mathbf{Id}$, and where $\mathbf{LS} = \mathbf{Id}$. The registers $\mathit{os}, \mathit{pc}, e, \mathit{rs}$ are now simply nodes in h. The heap itself becomes part of the state of the machine.

We start the machine with a state of the form $(\mathit{os}_0, 0, e_0, \mathit{rs}_0, h_0)$, where

$$\begin{array}{rcl}
(\mathit{os}_0, h') & = & \mathit{newstack}((\emptyset, \emptyset)) \\
(e_0, h'') & = & \mathit{newnode}(h') \\
(\mathit{rs}_0, h_0) & = & \mathit{newstack}(h'')
\end{array}$$

The following rules describe the new machine. Observe, where the mutator (here the virtual machine instructions) makes use of the mutator operations *newnode* and *update*, and its stack variants *newstack*, *push*, and *pop*.

We modify the rule for `LDCI` (`LDCB` is similar) as follows.

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

We give the rules for `PLUS` and `UMINUS`; the instructions for the remaining primitive operators are similar.

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

where

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

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

where

$$\begin{array}{rcl} (i, h') & = & pop(os, h) \\ h'' & = & push(os, 0 - i, h') \end{array}$$

The jump instructions are straightforward.

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

$$\frac{s(pc) = \texttt{JOFR}\ i}{(os, pc, e, rs, h) \rightrightarrows_s (os, pc+1, e, rs, h')} \quad \text{if } (true, h') = pop(os, h)$$

$$\frac{s(pc) = \texttt{JOFR}\ i}{(os, pc, e, rs, h) \rightrightarrows_s (os, pc+i, e, rs, h')} \quad \text{if } (false, h') = pop(os, h)$$

The `LDS` instruction follows the edge from node e with the label x given in the instruction.

$$\frac{s(pc) = \texttt{LDS}\ x}{(os, pc, e, rs, h) \rightrightarrows_s (os, pc+1, e, rs, h')} \quad \text{where } h' = \mathit{push}(os, \mathit{deref}(e, x, h), h)$$

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

where

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

As usual, the instruction `CALL` is the most complicated case.

$$\frac{s(pc) = \texttt{CALL}\ n}{(os, pc, e, rs, h^{(0)}) \rightrightarrows_s (os', a, e', rs, h')}$$

where

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

The last rule that we mention is for the `RTN` instruction. The remaining rules (`LDRFS`, its corresponding `CALL` rule, and `TAILCALL`) are left to the reader.

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

where

$$\begin{array}{rcl}
(sf, h^{(1)}) & = & pop(rs, h) \\
os' & = & deref(sf, \texttt{os}, h^{(1)}) \\
pc' & = & deref(sf, \texttt{pc}, h^{(1)}) \\
e' & = & deref(sf, \texttt{e}, h^{(1)}) \\
(v, h^{(2)}) & = & pop(os, h^{(1)}) \\
h' & = & push(os', v, h^{(2)})
\end{array}$$

With this machine in place, we can exactly account for the memory usage of each instruction. For example, the instruction `LDFS x y z` creates two nodes, one for the closure c, and one for the formals f, and seven edges, three to connect c to its children, one to connect the operand stack to c, and three to connect `x`, `y` and `z` to f.

The space consumption of a program can be measured as the number of nodes and edges that are created by it. Two sets of questions arise from this setup.

Firstly, how realistic is this graph view of a heap? How do we implement the heap graph in a computer memory? Is there a relationship between the number of nodes and edges and the amount of computer memory needed to represent them? This question is answered in the next section.

The second set of questions arises from the fact that instructions keep creating new nodes in the heap and never take them out again. The heap therefore becomes bigger and bigger during the computation, and eventually, we are running out of memory. Is this behavior inevitable? Once a node is created, will it have to be stored until the end of the program execution? The answer is clearly negative. For example, once a runtime stack frame has been popped from the runtime stack, it will never be used again.

At any point of time, the set of nodes V can be split into two sets, $V = V_{useful} \cup V_{useless}$. The set V_{useful} contains those nodes that will be used in the future. These nodes will appear in registers of the machine and will be operated on by instructions. The nodes in $V_{useless}$ on the other hand will never be used again. If we had an algorithm A that would split V into V_{useful} and $V_{useless}$, we could run A after each instruction (or once in a while) to find $V_{useless}$. We could then reuse the nodes in $V_{useless}$ and edges to and from $V_{useless}$ for future heap operations.

Unfortunately, the question whether a given node will be used in the future is in general undecidable. We cannot hope for an algorithm A. The best we can do is to approximate V_{useful} and $V_{useless}$ with an algorithm that splits V into V_{live} and V_{dead}, where $V_{live} \supseteq V_{useful}$ and $V_{dead} \subseteq V_{useless}$. All memory management techniques are based on such a distinction. They may differ in the choice of liveness criteria used. We shall see examples in Sections 10.6, 10.7, and 10.8.

9.5 Reference Counting

Memory management techniques identify elements of $V_{useless}$ so that the memory they are occupying can be reused. A sufficient condition for a node v to be in $V_{useless}$ is that there is no edge whose target is v. Thus, the idea of our first memory management technique is for every node to keep track of the number of edges that end in it. If that number drops to 0, the node has become unreachable, and therefore must be in $V_{useless}$. The memory that it occupies can be reused.

Thus, reference counting defines V_{dead} as follows.

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

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

Initially, all cells are placed in a `Freelist` data structure, indicating that all memory is available to the virtual machine. This data structure is realized directly in the heap by reusing a specific field in the node with offset `NEXT`. This way, a chain of nodes is created, each pointing to the next one using the `NEXT` field. Similarly, we keep track of the reference count of each node using a field with offset `RC`. The following code suggests the initialization of the heap.

```
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;
```

`NEXT` and `RC` are global constants, whereas `Freelist` is a global variable that will change during runtime. We can keep the `NEXT` pointer and the reference count `RC` in the same field, because at any point in time, only one of the two will be carrying useful information.

Note that for simplicity, we assume that all nodes have the same size, given by the constant `NODESIZE`. In actual virtual machines, nodes come in different sizes which may complicate memory management significantly.

Allocating a new node In order to allocate a new node, we need to look for a free cell in the freelist. The operation `newnode()` returns a reference to the newly created node to the virtual machine. Therefore, the reference count of the resulting node is 1.

```
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 Once the reference count of a cell drops to 0, the cell becomes garbage. Then, we can decrement the reference count of all of its children by 1 (one less "meaningful" reference), which may turn them into garbage as well. The operation `update()` therefore, carries out a `delete()` operation, which calls itself recursively, if `RC` has dropped to 0.

```
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;
}
```

Jones and Lins [**?**] list the following advantages of reference counting.

Incrementality Reference counting reclaims memory on the fly. Its execution is interleaved with the host program, and therefore, the real-time behavior of the host program is not much affected.

Locality A node whose reference count becomes zero can be reclaimed without access to other cells. Therefore, cache or page faults for memory management are minimized.

Immediate reuse Often, nodes in programs are short-lived and only a single reference to a node gets created. In this case, reference counting can quickly reuse the memory, leading to practically constant space usage of the host program.

The disadvantages of reference counting are:

Runtime overhead The updating of reference counts for every pointer manipulation incurs a significant runtime overhead on modern processor architectures.

Cyclic data structures Reference counting is unable to reclaim cyclic data structures that become inaccessible from the outside, since none of the elements of a cycle get a reference count of 0.

These two considerations led most runtime system implementors to abandon reference counting in favor of variants of one of the following two algorithms.

9.6 Mark-Sweep Garbage Collection

The next memory management algorithm is called a garbage collector. When `newnode()` runs out of memory, a garbage collector computes a set V_{dead}—with respect to a given liveness criterion—and reclaims the memory its elements were occupying.

The `update()` operation is not affected by the garbage collectors presented here, and thus looks as follows.

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

The garbage collector in this section is called `mark_sweep`; its `newnode()` operation is defined as follows.

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

Liveness From the operations on the heap, it is clear that a node can be reached in the future, only if it is connected through edges to some register of the machine. Another way of distinguishing a set $V_{live} \supseteq V_{useful}$ is therefore to check for such a connection. If we define the relation $\longrightarrow$ by the rule

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

we can define connectedness as the reflexive transitive closure $\longrightarrow^*$ of $\longrightarrow$.

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

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

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

The nodes $\{os, e, rs\}$ are called the roots of the heap, denoted by `Roots` in the following programs.

Algorithm The idea of mark-sweep garbage collection is to visit all nodes in V_{live} and mark them as such. This mark phase starts at the root nodes of the heap and recursively visits all nodes connected to them. Then, a sweep phase visits every node in the heap and makes every unmarked node available to future memory allocations. The algorithm uses a field `MARKBIT` for marking nodes as `MARKED` and `UNMARKED`.

```
static int MARKBIT  = 1;
static int MARKED   = 1;
static int UNMARKED = 0;
```

Free memory in mark-sweep is managed exactly like in reference counting, as a linked list of free nodes, whose first element is located at position `Freelist` in `HEAP`. Thus, `mark_sweep()` is implemented as follows.

```
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 The performance of a garbage collector can be expressed as the amount of memory reclaimed per unit of time. Thus for mark-sweep, we have

$$e_{MS} = \frac{m_{MS}}{t_{MS}}$$

where m_{MS} is the amount of reclaimed memory and t_{MS} is the time taken to reclaim it. When mark-sweep gets invoked, the size of the heap is $M = |V| =$ HEAPSIZE/NODESIZE. Recall that R denotes the number of live nodes. The quotient $r = R/M$ is called the residency of the program. The memory reclaimed by mark-sweep is

$$m_{MS} = M - R$$

The time taken by mark-sweep is the time taken by the mark procedure, which scans the live memory, and the time taken by sweep, which scans the entire heap. Thus

$$t_{MS} = a \cdot R + b \cdot M$$

for some constants a and b. The overall efficiency can therefore be expressed as

$$e_{MS} = \frac{m_{MS}}{t_{MS}} = \frac{M - R}{aR + bM} = \frac{1 - r}{ar + b}$$

As residency approaches 0, the efficiency is given by $1/b$, thus is dominated by the time spent by sweep at each node. The efficency approaches 0 as residency approaches 1.

9.7 Copying Garbage Collection

Copying garbage collection employs the same notion of live nodes as mark-sweep. This garbage collector only uses half of the available memory for allocating nodes. Once this half is filled up, the garbage collector copies the live memory contained in the first half, the so-called from-space, to the other half, the so-called to-space. Now, the roles of the halves is reversed and the other half is filled up and so on. To achieve this, we initialize the heap as follows.

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

The address `Free` points to the first location in to-space. The operation `newnode()` allocates memory at the position that `Free` points to. Similar to mark-sweep, `newnode()` performs garbage collection—here using `flip()`—when no memory is available. If there is still no memory after garbage collection, the program aborts with the message "memory exhausted".

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

Algorithm The following implementation of copying garbage collection is due to C.J. Cheney [?]. It first switches the roles of to-space and from-space. The pointer `Free` keeps track of the next free position in to-space. Then the roots are copied to to-space. Now, the pointer `scan` visits each already copied node and copies its children to to-space. The copying process terminates as soon as `scan` catches up with `Free`, at which point all children of all nodes have been copied to to-space. Allocation can resume in the part of the new to-space that remains free.

```
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;
   }
}
```

The `copy()` function uses a field in every node to keep track of the forwarding address.

```
static int FORWARDINGADDRESS = 1;
```

Which fields are used is not relevant as long as it is not written to by `update()`, and initialized to an integer that is not a heap address, for example `-1`. If a node in from-space has already been moved to to-space, the field `FORWARDINGADDRESS` of the original node in from-space indicates the address of its copy in to-space.

The procedure `copy()` for copying from-space nodes `v` checks whether `v` has already been copied. In that case, it returns `v`'s forwarding address. If not, then the node is moved to to-space. To avoid repeated copying, the original node in from-space is equipped with the address of its copy in to-space.

```
int copy(int v) {
   if (already_copied(v))
      return HEAP[v+FORWARDINGADDRESS];
   else {
      int addr = Free;
      move(v,Free);
      Free = Free + NODESIZE;
      HEAP[v+FORWARDINGADDRESS] = addr;
      return addr;
   }
}
```

It is straightforward to decide if a given node has been copied already. It suffices to check whether its forwarding address is in to-space.

```
boolean already_copied(int v) {
   return HEAP[v+FORWARDINGADDRESS] >= Tospace
       && HEAP[v+FORWARDINGADDRESS] <= Topofspace;
}
```

Performance The amount of memory recovered by copying garbage collection is

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

since the other half of M is reserved for copying. Since copying garbage collection only visits live nodes, it takes

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

where c is a constant. The resulting efficiency is

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

Thus the efficiency of copying garbage collection approaches infinity as residency approaches 0, whereas it approaches 0 as residency approaches 1/2, which is the limit where it reports "exhausted memory".

From comparing the efficiency of mark-sweep with copying garbage collection, we conclude that mark-sweep is more efficient than copying as residency approaches 1/2. Copying on the other hand is more efficient than mark-sweep, when residency is very low. Which algorithm is better therefore depends on the runtime characteristics of the application.

9.8 Historical Background and Further Reading

Historical notes Automatic memory management was pioneered by first implementations of the language LISP. All techniques described here were developed in the context of LISP.

Reference counting was first described by Gelernter, Hansen and Gerberich [?]. The standard algorithm presented here is due to Collins [?] and has been used as memory management technique in early versions of Smalltalk, and Modula-2+. Mark-sweep garbage collection is due to McCarthy [?], and the algorithm presented here is based on his description. The first copying collector was Minsky's garbage collector for Lisp 1.5 [?]. The copying collector in Section 9.7, Cheney's algorithm [?], is the simplest and most elegant variant.

With Java, automatic memory management has entered the mainstream of information technology. Most implementations of the Java Virtual Machine use variants of mark-sweep garbage collection.

Explicit heap allocation The first usage of heap allocation was to accommodate user-defined data structures in Algol and Pascal. These languages provide for pointer variables of a given type. For example,

```
    var p :  ↑ t
```

declares the variable `p` to be a pointer that can point to data structures of type `t`. Pointers can be changed by assignment such as `p := nil`. The operation `new(p)` makes `p` point to newly allocated memory capable of storing data of type `t`. Instead of delegating the management of heap memory to the runtime system, these languages provide an operation `dispose(p)` for explicitly deallocating memory such that the memory manager can reuse it in future `new` operations.

The need for explicit deallocation in such languages often considerably increases the complexity and therefore the cost of software development. Common errors resulting from incorrect deallocation are space leaks and dangling references. An example for a space leak is the program

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

The object created by `new` has become inaccessible. Since it has not been deallocated, the memory manager cannot reuse its space. The space has "leaked away". An example for a dangling reference is

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

Now the data structure `a` has a reference to a structure that may be reused by the memory manager. Eventually, the memory manager will allocate a different data structure to the location where `a.s` points to, which will give rise to a program error.

Memory management in software systems Memory management is of course not limited to programming systems. Programmers need to carefully manage memory regardless whether the runtime system they use employs automatic memory management or not.

Space leaks can occur even in systems with automatic memory management. A space leak occurs in such a system, if the program accumulates larger and larger data structures that are accessible from the root nodes, but that will actually never be used.

In response, large software systems sometimes implement their internal "automatic" memory management. For example, reference counting is used as a software pattern in systems as diverse as the Unix operating system and Adobe Photoshop.

Further reading Automatic memory management is a prolific area of research within programming language implementation research. The book *Garbage Collection* by Richard Jones and Rafael Lins [**?**] gives an excellent introduction to the area and has an extensive bibliography.