Pointers and the Heap

Farzaneh Derakhshan
based on material by John C. Reynolds

CS 536: Science of Programming, Fall 2023
Lecture 19

In the next two lectures, we will expand our programming language and reasoning tools to include pointer structures. Pointer structures allow us to represent lists, trees, etc., and write programs that use addresses in memory to access data.

1 Store vs Heap

Up until now, we have only been dealing with programs that use states (σ) to map variables to values. However, to work with pointer structures, we need to add another component to the states called the heap. The heap (h) is in the form of functions from location addresses to values. In this new setting, the state is a pair of a store σ, and a heap h.

The store σ, similar to its originial definition, maps variables to their values. For example $\sigma_1 \triangleq \{x = 52, y = 3\}$, maps variable x to value 52, and variable y to value 3. We can also represent σ as a function from variables *Var* to values, i.e. $\sigma : \mathit{Var} \to \mathit{Values}$. In the example above, we have $\sigma_1(x) = 52$ and $\sigma_1(y) = 3$. We can think of the store (sometimes also called the stack) as a temporary data storage, for example the contents of the registers.

The heap, on the other hand, describes the contents of an addressable memory. We model it as a function that maps the address of each location in the memory to a value, i.e., i.e. $h : \mathit{Addresses} \to \mathit{Values}$. These location addresses are represented by natural numbers. We can perform various operations on the heap, such as reading from, writing to, extending, or shrinking it.

Example 1. The store $\sigma_1 \triangleq \{x = 52, y = 3\}$ maps x to 52 and y to 3. The heap $h_1 \triangleq \{52 \mapsto 3\}$ maps the location with address 52 to value 3, i.e., the content of the memory location with address 52 is 3.

To summarize,

state	=	(*store* × *heap*)	*e.g.*	(σ_1, h_1)
store	=	*Var* → *Values*	*e.g.*	$\sigma_1 = \{x = 52, y = 3\}$
heap	=	*Addresses* → *Values*	*e.g.*	$h_1 = \{52 \mapsto 3\}$.

2 The Extended Language

The syntax and semantics of expressions are the same as before. In particular, expressions do not depend upon the heap and never cause any side effects on the heap. But we need to expand the syntax and semantics of our statements to work with the heap. We add four new constructs to our language as below.

Statements	e	::=	$\cdots$	
		|	$x :=\!!e$	Lookup
		|	$!e_1 := e_2$	Mutation
		|	$x := \mathbf{cons}(e_1, \cdots, e_n)$	Allocation
		|	$\mathbf{dispose}(e)$	Deallocation

The notation $!e$ denotes the contents of the storage at address e. For example, in $h_1 = \{52 \mapsto 3\}$, we represent the contents of address 52 as $!\overline{52}$ which is equal to 3. Now, Let's take a closer look at each of these constructs.

2.1 Lookup ($x :=!e$):

This statement first evaluates e with respect to the store to get an address location and then assigns the content of the address in variable x.
Example 2. If we start with the store $\sigma_1 \triangleq \{x = 52, y = 3\}$ and heap $h_1 \triangleq \{52 \mapsto 3\}$, running $z :=!x$ assigns the contents of the heap location at the address 52 to a variable z in the store. The resulting store will be $\sigma_2 \triangleq \sigma_1[z \mapsto 3] = \{x = 52, y = 3, z = 3\}$ and the heap remains unchanged. We can write this using the small-step notation as

$$\langle z :=!x, (\sigma_1, h_1)\rangle \rightarrow \langle \mathsf{skip}, (\sigma_2, h_1)\rangle.$$

We will provide the rules for the small step semantics of the extended semantics more formally later in this lecture notes (Section 5), but for now observe that the configuration consists of a statement and a pair of a store and a heap.

2.2 Mutation ($!e_1 := e_2$):

The statement evaluates e_1 and e_2 using the store to get an address location and a value, respectively. Then it assigns the value to the location.
Example 3. Consider the store $\sigma_2 \triangleq \{x = 52, y = 3, z = 3\}$ and heap $h_1 \triangleq \{52 \mapsto 3\}$, running $!x := y + 1$ assigns value 4 to the location with address 52. The store doesn't change after running this statement, i.e., we still have $\sigma_2 \triangleq \{x = 52, y = 3, z = 3\}$, but the heap changes to $h_2 \triangleq \{52 \mapsto 4\}$. We can write this using the small-step notation as

$$\langle !x := y + 1, (\sigma_2, h_1)\rangle \rightarrow \langle \mathsf{skip}, (\sigma_2, h_2)\rangle.$$

2.3 Allocation ($x := \mathbf{cons}(e_1, \cdots, e_n)$)

This statement is used to allocate new location(s) in the heap. It allocates n fresh consecutive locations in the heap with the values e_1 to e_n and assigns the address of the first location to the store variable x. A location is fresh if it not already in use. We assume that there are an infinite number of addresses available, so fresh locations are always available.
Example 4. Consider the store $\sigma_2 \triangleq \{x = 52, y = 3, z = 3\}$ and heap $h_2 \triangleq \{52 \mapsto 4\}$. Let's say that we want to allocate one location in the heap with value 1 and keep its address in a store variable w. We write $w := \mathbf{cons}(\overline{1})$. After running this statement, we get a larger heap $h_3 \triangleq \{52 \mapsto 4, 9584 \mapsto 1\}$ and a new store $\sigma_3 \triangleq \{x = 52, y = 3, z = 3, w = 9584\}$. It is important to notice that the choice of the address for the newly allocated cell is unpredictable. The system can choose any address as long as it is not in use already. In this example, we assume that the system gives us the address 9584. Again, we write this using the small-step notation as

$$\langle w := \mathbf{cons}(\overline{1}), (\sigma_2, h_2)\rangle \rightarrow \langle \mathsf{skip}, (\sigma_3, h_3)\rangle.$$

Example 5. We just observed that we cannot predict the address of the newly allocated memory, but we can use the **cons** construct to allocate multiple addresses that are next to each other. For example, having the store $\sigma_3 \triangleq \{x = 52, y = 3, z = 3, w = 9584\}$ and heap $h_3 \triangleq \{52 \mapsto 4, 9584 \mapsto 1\}$, we want to allocate two new memory cells in the heap that are next to each other such that the first one has value 2 and the second one has value 3. We write $u := \mathbf{cons}(\overline{2}, \overline{3})$. After running this statement, we get a larger heap $h_4 \triangleq \{52 \mapsto 4, 9584 \mapsto 1, 1001 \mapsto 2, 1002 \mapsto 3\}$ and a new store $\sigma_4 \triangleq \{x = 52, y = 3, z = 3, w = 9584, u = 1001\}$. The two cells are consecutive (1001 and 1002), and we only record the address of the first one in the store variable u. We write this using the small-step notation as

$$\langle w := \mathbf{cons}(\overline{2}, \overline{3}), (\sigma_3, h_3)\rangle \rightarrow \langle \mathsf{skip}, (\sigma_4, h_4)\rangle.$$

Exercise 1. Having σ_4 and h_4, what is the value of $!(u + 1)$? What about $(!u) + 1$?

2.4 Deallocation (dispose(e))

We use this statement to deallocate a location in the heap. We first evaluate the expression e using the store to find a location and then deallocate the location from the heap.

Example 6. Consider the store $\sigma_4 \triangleq \{x = 52, y = 3, z = 3, w = 9584, u = 1001\}$ and the heap $h_4 \triangleq \{52 \mapsto 4, 9584 \mapsto 1, 1001 \mapsto 2, 1002 \mapsto 3\}$, we want to deallocate the location which its address is stored in the store variable x. We write $\mathbf{dispose}(x)$. After running this statement, we get a smaller heap $h_5 \triangleq \{9584 \mapsto 1, 1001 \mapsto 2, 1002 \mapsto 3\}$ and the same store $\sigma_4 \triangleq \{x = 52, y = 3, z = 3, w = 9584, u = 1001\}$. We write this using the small-step notation as

$$\langle \mathbf{dispose}(x), (\sigma_4, h_4)\rangle \to \langle \mathsf{skip}, (\sigma_4, h_5)\rangle.$$

3 Run-time Errors

As you may have noticed, working with pointers is very delicate and error-prone. In this section, we will discuss some scenarios that may lead to runtime errors when using the statements above in a program. For example, looking up, mutation, or deallocating a dangling address will cause an error.

Example 7. Consider the store $\sigma_4 \triangleq \{x = 52, y = 3, z = 3, w = 9584, u = 1001\}$ and the heap $h_5 \triangleq \{9584 \mapsto 1, 1001 \mapsto 2, 1002 \mapsto 3\}$. In this case, $!x$ is a dangling address, i.e., in Example 6 we deallocated the address stored in x, and thus it is not available in the heap anymore. Statements $v :=!x$ and $!x := e$ and $\mathbf{dispose}(x)$ all result in runtime error, i.e.,

$$\begin{array}{lcl} \langle v :=!x, (\sigma_4, h_5)\rangle & \to & \langle \mathsf{skip}, \bot\rangle \\ \langle !x = e, (\sigma_4, h_5)\rangle & \to & \langle \mathsf{skip}, \bot\rangle \\ \langle \mathbf{dispose}(x), (\sigma_4, h_5)\rangle & \to & \langle \mathsf{skip}, \bot\rangle. \end{array}$$

Similarly, using pointer arithmetic may result in run-time error. For example $v :=!(u + \overline{2})$ looks up the address $1001 + 2 = 1003$ which does not exist in the heap h_5 and results in a runtime error.

$$\langle v :=!(u + \overline{2}), (\sigma_4, h_5)\rangle \quad \to \quad \langle \mathsf{skip}, \bot\rangle$$

Moreover, our programming language has a explicit memory managenent, i.e., there is no garbage collection to free up heap locations that have been allocated but are no longer needed by the program.

4 Example - Linked List

A linked list is a linear data structure where each element is stored as a separate object. Unlike an array, linked list elements are not stored continuously in memory. Instead, each node is stored in two consecutive memory locations. In the first location the data entry is stored, and the second location stores a reference to the next node in the sequence. A reference, usually called the head, points to the node with the first data entry. The last node in the list has the reference *nil* instead of the next node to signify the end of the list. See Figure 1.

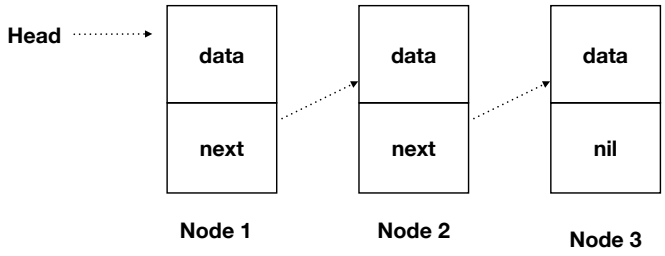

Figure 1: Linked list structure

Figure 2 shows an example of a linked list with three nodes with addresses of each memory location being assigned. Note that each node has two consecutive addresses. The first entry contains the data of the node and the second entry of the node keeps the address of the data entry of the next node.

Finally, Figure 3 shows a linked list that stores the array $[10, 5, 3]$.

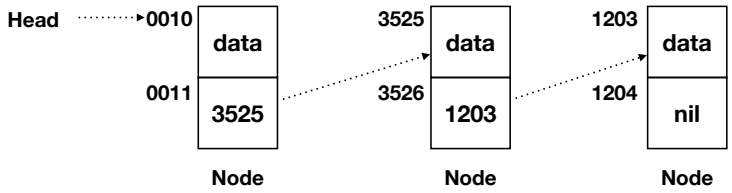

Figure 2: Linked list with addresses

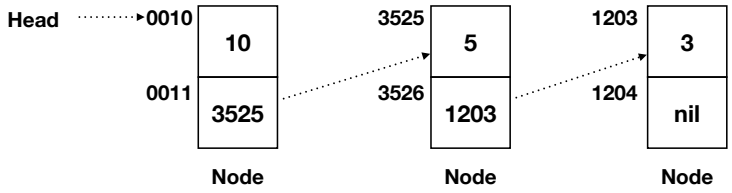

Figure 3: Linked list corresponding to the array [10,5,3]

Next, we write some programs to build and manipulate the linked lists in our language.

Example 8. The program S_1 builds the linked list in Figure 3, starting from an empty heap (we use **emp** for an empty heap) and any stack σ.

$$\begin{array}{rl} S_1 := & x_0 := \mathbf{cons}(\overline{3}, nil); \\ & x_1 := \mathbf{cons}(\overline{5}, x_0); \\ & head := \mathbf{cons}(\overline{10}, x_1) \end{array}$$

The first line allocates the last node in Figure 3 and assigns the address of its first location in the variable x_0. The second line allocates the second node in Figure 3 and assigns the address of its first location in the variable x_1. Similarly, the third line allocates the first node and assigns itsfirst location address in a variable called *head*.

Running the program using the small step semantics, we get:

$$\begin{array}{ll} \langle S_1, (\sigma, \mathbf{emp})\rangle & \rightarrow^2 \\ \langle x_1 := \mathbf{cons}(\overline{5}, x_0); head := \mathbf{cons}(\overline{10}, x_1), (\sigma[x_0 \mapsto 1203], \{1203 \mapsto 3, 1204 \mapsto nil\})\rangle & \rightarrow^2 \\ \langle head := \mathbf{cons}(\overline{10}, x_1), (\sigma[x_0 \mapsto 1203][x_1 \mapsto 3525], \{1203 \mapsto 3, 1204 \mapsto nil, 3525 \mapsto 5, 3526 \mapsto 1203\})\rangle & \rightarrow \\ \langle \mathsf{skip}, (\sigma[x_0 \mapsto 1203][x_1 \mapsto 3525][head \mapsto 0010], \{1203 \mapsto 3, 1204 \mapsto nil, 3525 \mapsto 5, 3526 \mapsto 1203, & \\ \qquad 0010 \mapsto 10, 0011 \mapsto 3525\})\rangle & \end{array}$$

The final result is the store $\sigma' := \sigma[x_0 \mapsto 1203][x_1 \mapsto 3525][head \mapsto 0010]$ and the heap $h' := \{1203 \mapsto 3, 1204 \mapsto nil, 3525 \mapsto 5, 3526 \mapsto 1203, 0010 \mapsto 10, 0011 \mapsto 3525\}$

Example 9. Next, we write a program that increments the value of the 2nd node of the list by one.

$$\begin{array}{rl} S_2 := & x :=!(head + \overline{1}); \\ & y :=!x; \\ & !x := y + \overline{1} \end{array}$$

Example 10. Next, we write a program that deallocates the first node (but first, it stores the address of the second node in x so we can make that the new head).

$$\begin{array}{rl} S_2 := & x :=!(head + \overline{1}); \\ & \mathbf{dispose}(head); \\ & \mathbf{dispose}(head + \overline{1}); \\ & head := x \end{array}$$

Example 11. Finally, we write a program that appends two lists L_1 and L_2 into a single list L , when the first one (L_1) has exactly three nodes and its head is stored in store variable $head_1$, the head of the

second one (L_2) is stored in store variable $head_2$. After merging the two lists, we assign the head of L in the variable $head$.

$$\begin{array}{rl} S_2 := & x_0 :=!(head_1 + \overline{1}); \\ & x_1 :=!(x_0 + \overline{1}); \\ & !(x_1 + \overline{1}) := head_2; \\ & head = head_1 \end{array}$$

This code only works if we know that the two lists L_1 and L_2 are disjoint, i.e., they don't have any node in common. We will see why separation is important and how we can enforce it in the next lecture.

5 Small Step Semantics

In Section 2, we described the small step semantics of each new construct informally. Here, we provide the more formal rules.

$$\frac{\ell = \sigma(e) \qquad \ell \in dom(h) \qquad v = h(\ell) \qquad \sigma' = \sigma[x \mapsto v]}{\langle x :=!e, (\sigma, h)\rangle \to \langle \mathsf{skip}, (\sigma', h)\rangle} \qquad \frac{\ell = \sigma(e_1) \qquad v = \sigma(e_2) \qquad \ell \in dom(h)}{\langle !e_1 := e_2, (\sigma, h)\rangle \to \langle \mathsf{skip}, (\sigma, h[\ell \mapsto v])\rangle}$$

$$\frac{\ell_1, \ell_1 + 1 \cdots, \ell_1 + n - 1 \notin dom(h) \qquad \forall i.\, v_i = \sigma(e_i) \qquad h' = h[\ell_i \mapsto v_i]_{1 \le i \le n} \qquad \sigma' = \sigma[x \mapsto \ell_1]}{\langle x = \mathbf{cons}(e_1, \cdots, e_n), (\sigma, h)\rangle \to \langle \mathsf{skip}, (\sigma, h')\rangle}$$

$$\frac{\ell = \sigma(e) \qquad h = h', \ell \mapsto v}{\langle \mathbf{dispose}(e), (\sigma, h)\rangle \to \langle \mathsf{skip}, (\sigma, h')\rangle}$$

Since the new configuration also contains the heap, we update the previous set of rules (minimally) to include the heap in the configurations. (The only interesting rule is the first sequencing rule which allows the first statement to update both the store and the heap.)

$$\frac{}{\langle x := e, (\sigma, h)\rangle \to \langle \mathsf{skip}, (\sigma[x \mapsto \sigma(e)], h)\rangle} \qquad \frac{0 \le \sigma(e_1) < |\sigma(a)|}{\langle a[e_1] := e_2, (\sigma, h)\rangle \to \langle \mathsf{skip}, (\sigma[a[\sigma(e_1)] \mapsto \sigma(e_2)], h)\rangle}$$

$$\frac{\langle S_1, (\sigma, h)\rangle \to \langle S_1', (\sigma', h')\rangle}{\langle S_1; S_2, (\sigma, h)\rangle \to \langle S_1'; S_2, (\sigma', h')\rangle} \qquad \frac{}{\langle \mathsf{skip}; S, (\sigma, h)\rangle \to \langle S, (\sigma, h)\rangle}$$

$$\frac{\sigma(e) = T}{\langle \mathsf{if}\ e\ \mathsf{then}\ S_1\ \mathsf{else}\ S_2\ \mathsf{fi}, (\sigma, h)\rangle \to \langle S_1, (\sigma, h)\rangle} \qquad \frac{\sigma(e) = F}{\langle \mathsf{if}\ e\ \mathsf{then}\ S_1\ \mathsf{else}\ S_2\ \mathsf{fi}, (\sigma, h)\rangle \to \langle S_2, (\sigma, h)\rangle}$$

$$\frac{}{\langle \mathsf{while}\ e\ \mathsf{do}\ S\ \mathsf{od}, (\sigma, h)\rangle \to \langle \mathsf{if}\ e\ \mathsf{then}\ S; \mathsf{while}\ e\ \mathsf{do}\ S\ \mathsf{od}\ \mathsf{else}\ \mathsf{skip}\ \mathsf{fi}, (\sigma, h)\rangle}$$

Finally, we provide the rules handling runtime error as described in Section 3.

$$\frac{\sigma(e) = \perp}{\langle x :=!e, (\sigma, h)\rangle \to \langle \mathsf{skip}, \perp\rangle} \qquad \frac{\ell = \sigma(e) \qquad \ell \notin dom(h)}{\langle x :=!e, (\sigma, h)\rangle \to \langle \mathsf{skip}, \perp\rangle} \qquad \frac{\sigma(e_1) = \perp \ \ or \ \ \sigma(e_2) = \perp}{\langle !e_1 := e_2, (\sigma, h)\rangle \to \langle \mathsf{skip}, \perp\rangle}$$

$$\frac{\ell = \sigma(e_1) \qquad \ell \notin dom(h)}{\langle !e_1 := e_2, (\sigma, h)\rangle \to \langle \mathsf{skip}, \perp\rangle} \qquad \frac{\exists i.\, \sigma(e_i) = \perp}{\langle x = \mathbf{cons}(e_1, \cdots, e_n), (\sigma, h)\rangle \to \langle \mathsf{skip}, \perp\rangle}$$

$$\frac{\sigma(e) = \perp}{\langle \mathbf{dispose}(e), (\sigma, h)\rangle \to \langle \mathsf{skip}, \perp\rangle} \qquad \frac{\ell = \sigma(e) \qquad \ell \notin dom(h)}{\langle \mathbf{dispose}(e), (\sigma, h)\rangle \to \langle \mathsf{skip}, \perp\rangle}$$