Heaps and Garbage Collection

David May: April 8, 2014

Introduction

Many programming languages include the ability to create data structures formed from *tuples*. Each tuple contains a set of values, and each value may itself be a tuple. As execution proceeds, more and more tuples are formed and eventually no space remains. However, usually when the space runs out, many of the tuples are no longer needed; the space they occupy can be reclaimed and execution can continue.

The space used for the tuples is known as the *heap* and the process of performing the space reclamation is known as *garbage collection*.

Memory layout

A typical memory layout for a language using globals, locals and heap is:

high memory	heap
	globals
	stack (descending)
	program
low memory	large constants

Tuples

A tuple consists of a number of word locations. One of these is a special *control word*, containing:

- the length of the tuple
- a flag that is used by the garbage collector to mark tuples that are to be retained
- an address that is allocated by the garbage collector; this will become the new address of the tuple at the end of the garbage collection process

A *heap pointer* is used to point to the next available location in the heap; this is used to allocate space when a tuple is created.

Identifying locations containing heap addresses

If the programming language is typed, it may be possible to identify which local and global variables (and which variables within data structures) hold addresses. These addresses can then be checked to determine if they lie within the heap.

If not, it is possible to encode addresses and numbers so that they can be easily distinguished. In a byte-addressed memory, the least significant bit of a word address will always be zero. The tuples in the heap can all be aligned to start at word addresses, so the address of any tuple will have a least significant bit of 0. By setting the least significant bit of all numbers to 1, they can be distinguished from addresses of tuples.

This means that one fewer bit is available for representing numbers and a number x will be represented by a value X, where $X = (x << 1)+1$. This requires some minor changes to the compilation of some operators. For example:

expression	compilation
$a + b$	$A + (B - 1)$
$a - b$	$A - (B - 1)$
$a \wedge b$	$A \wedge B$
$a \vee b$	$A \vee B$
$\neg a$	$\neg(A - 1)$
$a << b$	$((A - 1) << (B >> 1)) + 1$

This ensures that the result of the operators always has a least significant bit of 1.

Garbage Collection

When a program tries to create a new tuple and the heap pointer has reached the top of memory, the garbage collector is entered. It operates as follows:

1. It examines all of the words in memory starting from the stack pointer and ending at the top of the global region. For every word that holds an address of a tuple in the heap, it sets the mark flag in the tuple and in each of the tuples in the graph of tuples connected to it.

2. It resets the heap pointer to the lowest address in the heap and then examines all of the tuples in the heap starting at the lowest address. For each marked tuple, it allocates a new location in the heap using the heap pointer and increases the heap pointer by the size of the tuple. It also writes the address of the new location into the control word of the tuple.

3. It examines all of the words in memory starting from the stack pointer and

ending at the top of the heap. For every word that holds an address of a tuple in the heap, it replaces the value of the word with the address held in the tuple control word.

4. It resets the heap pointer to the lowest address in the heap and then examines all of the tuples in the heap starting at the lowest address. For each marked tuple, it copies the tuple to a new location in the heap using the heap pointer and increases the heap pointer by the size of the tuple. As it copies the tuple, it clears the mark flag.

5. Finally, it creates the new tuple using the heap pointer and returns control to the program.

Marking

Three methods can be used during the marking step (1) above:

1. Recursion: The graph can be traversed recursively, following heap addresses in each tuple (unless they address an already-marked tuple). This has the disadvantage that space must be reserved for the stack used to implement the recusrsion.

2. Repetitive marking: The heap can be repeatedly scanned, identifying heap addresses within marked tuples and marking the tuples they address; this continues until a scan fails to mark any more tuples. This has the disadvantage that it is potentially slow.

3. Pointer reversal: This traverses the graph in the same way as recursion, but avoids the need for a stack. It maintains a pointer c to the current tuple and a pointer p to the previous tuple. When using a location n in the current tuple to advance to a new tuple, it performs $c, p, n \leftarrow n, c, p$. This has the effect of storing the address of the previous tuple in the current tuple before advancing the current and previous pointers. It is now possible to return to the previous tuple using this stored address, and to replace it with its original value by performing $c, p, n \leftarrow p, n, c$. This can normally be implemented by a few instructions using processor registers to hold c and p.