HLP Worksheet 5 - Exploring Garbage Collection

Contents

Introduction
Demonstration Code
Simulating List Data Usage
List Data Structure
Stack Data
Function Calls
Memory Allocation
Writing Simulation Code
Garbage collection
Tick Assessment

Introduction

This worksheet explores how garbage collection is implemented by instrumenting an F# program with additional code that simulates its usage of heap memory. The simulation can monitor memory usage and even test different garbage collectors.

It is important to distinguish between F# memory, which is garbage collected by the F# run-time system and allows F# programs to use memory, and the *simulated* memory usage. The simulation makes a few simplifications in the interests of simplicity.

- The simulated memory data will all be lists of floats. This allows a variety of list processing functions to be implemented and is enough to test garbage collection. Extending the simulation to more complex data structures is possible.
- The simulated functions all use a (simulated) stack on which local data is stored. It turns out that rewriting pure functional code to use such a stack is straightforward.
- The data in use by the program at any stage - and hence marked or copied by the simulated garbage collector - is assumed to be exactly that which is accessible from the simulated stack.
- For this to work, as we will see, the simulation needs to make use of mutable data. Both local data items, and the stack itself, will be mutable and changed throughout program execution.

This application illustrates powerfully how a hybrid functional language like F# can be used to write "difficult" imperative algorithms on mutable data like garbage collection. Mutation is used where it is required, and pure functional code used elsewhere. High order functions allow abstractions that greatly simplify writing code.

Demonstration Code

Code is written in answers and the complete working code (excepting those parts used for assessment) is provided here.

Simulating List Data Usage

List Data Structure

The simulation cannot use F# lists because we need to keep track of how each list node is allocated and freed as the program executed. The type **`IList`** defined below associates with each list node an integer **`id`** that represents the identity of the memory used by the node. For example id might be the index of an array each element of which is an IList node. The complete array would then represent the available heap memory. In the simulation this array is not required. IList nodes are actually implemented in F# (and the required memory is garbage collected by the F# gc). The **`id`** tags on each list node allow code to *simulate* allocation and deallocation of heap memory.

The lists we use can potentially have any element type, but for simplicity the demo will use only lists of floats. Obviously, lists of any data type that requires memory allocation (e.g. lists of float list) would complicate the garbage collector.

```
1:  type 'a IList = INode of  Id: int * Hd: 'a * Tl : 'a IList | INil
2:
3:  /// internal data constructor which takes garbage collector as a parameter
4:  /// iCons will be derived from this after gc is defined
5:  let iCons' gcAlloc h t = INode( Id = gcAlloc t, Hd=h, Tl=t)
6:
7:  let iHd x =
8:      match x with
9:      | INode(i,h,t) -> Some h
10:     | _ -> None
11:
12: let iTl x =
13:     match x with
14:     | INode(i,h,t) -> Some t
15:     | _ -> None
16:
17:
18: let iId = function
19:     | INode(i, h, t) -> i
20:     | INil -> -1
21:
22: let (|IMatch|_|) = function
23:     | INode (i, h, t) -> Some (h,t)
24:     | _ -> None
25:
26:
27: type Data = float IList // Type of all data used in simulations
```

The IList functions are **`iCons'`**, **`iHd`**, **`iTl`** corresponding to **`list`** functions **`List.Cons`**, **`List.head`**, **`List.tail`**. **`iCons'`** has an extra parameter **`gcAlloc`** used to interface garbage collection code. We will look at this later, for now you can suppose that **`gcAlloc`** ignores its parameter and returns a unique id corresponding to a new block of free memory whenever it is called.

`INil` is the **`IList`** data constructor corresponding to **`List.empty`** or **`[]`**.

The function **`IMatch`** is a partial active pattern equivalent to the operator **`::`** on lists. Built-in lists in F# use the same **`::`** operator for **`list.cons`** and pattern matching. This convenient syntax is not possible for **`IList`** since user-defined active patterns cannot be operators.

The definition of `IMatch` means that `| IMatch (h,t) ->` is the `IList` equivalent of list `| h :: t ->`. As with normal lists, we will write nearly all code using `IMatch` and therefore not need `iHd` or `iTl`.

Stack Data

In order to use `IList` we will need to define a garbage collector. The simulation will rely on a *stack* data structure from which all alive data can be reached. We will follow the (real) implementation of compiled languages by implementing a separate *stack frame* for each active function call. The stack frame contains the function parameters and any other local variables needed to implement the function body. The stack is an (F#) list of stack frames. Stack is a global mutable variable, allowing the gc function to access stack data. The gc function can be called from gcAlloc, therefore all stack data must be in a consistent state, and all previously allocated data cells whenever `iCons` is called.

Each stack frame contains data that simulates mutable local variables in a function call. Each variable has a string name and a mutable `Data` value. The number and names of variables can be fixed dynamically whenever a new stack frame is created (on function entry).

Q1. What is a suitable definition for type `StackFrame` and global variable `GlobalStack`?

Answer

Function Calls

Consider a two-parameter function: `f: a: Data -> b: Data -> 'T`. In order to simulate this we need to create a stack frame with simulated variables a and b on entry to `f`, adding this to the global stack, and removing it on exit from `f`. We will write a *wrapper* function which adds the necessary stack machinery to an implementation of `f` that uses stack for its parameters and local data `fUsingStack`.

```
1:  ManageCallTwoParameters1: (p1:string) -> (p2:string) -> (fUsingStack: StackFrame -> 'T) -> (Data -> Data -> 'T)
```

This high order function will return a function `f` that can be called as normal with two `Data` parameters:

```
1:  f a b
```

and invisibly instruments its function call as follows:

1. A new stack frame is created with slots for p1 (holding `a`) and p2 (holding `b`)
2. `fUsingStack` is called with the created stack frame as parameter.
3. On exit of `fUsingStack` its result is returned, and the created stack frame is removed from the stack.

Q2. Write a function `ManageCallTwoParameters1` as defined above, looking at the answer if you need inspiration.

Answer

Q3. We need two additions for this function to be generally useful.

- The 'wrapped' function may need additional local variables in its stack frame for temporary data. That can be managed with an extra parameter `locals` as below specifying the names of optional extra locals in the stack frame.

```
1: let ManageCallTwoParameters1 p1Name p2Name (locals: string list) (functionUsingStack: StackFrame -> (Data -> Data -> 'a) -> 'a) =
```

- `f` will typically be recursive. The recursive call inside `fUsingStack` must be to the *wrapped* function `f` so that each recursive call has a new stack frame. This is best implemented by feeding the wrapped function in as an extra parameter to `functionUsingStack`, which is therefore written as a non-recursive function parametrised by the recursively called function.

```
1:  functionUsingStack: (sf: StackFrame) -> (recCall: Data -> Data -> 'a) -> Data
```

Answer

> The use of higher order functions as written here is highly abstract. You may need to see typical use examples before you fully understand how it works. That is expected, and we will get to that later in the worksheet, at which point you can refer back to this code and understand it more completely. If you like you can skip ahead to the definition of `append` implemented as with `appendUsingStack` for one example.

Memory Allocation

The garbage collector (GC) `gcAlloc` is called whenever a new `IList` node is needed, by `iCons'`. The rule for garbage collection is that any time the GC is called all currently in use (or *alive*) heap ids must be reachable from data on the stack. If there is no available free memory the GC will perform a garbace collection by marking all heap ids which are reachable, and recycling all heap ids not reachable as free memory. In order to test GC correctness whenever `gcAlloc` is called it will check its parameter (which is the tail of an `iCons` operation and therefore should be alive memory) to see whether all of its ids are alive. If any such id is not alive we know there is a mistake in the code. This checking is very time intensive and cannot be left on for production code. However it is very useful for initial debugging since otherwise GC errors will result in used memory being reused and therefore overwritten, difficult to detect.

In this simulation no memory is overwritten. Two `INode`s with the same `id` can in fact have different values, because the memory is only simulated and distinct F# memory is used for every `INode`. In a real implementation that would not be true.

Q4. Check the code in the answer and see if you understand how it works. It does not do garbage collection relying on a new integer `id` for every allocated cell. However it is enough to check whether this system is correct, in the sense that all alive ids can be reached from the stack.

Answer

Writing Simulation Code

We can now put all the machinery so far together and look at how to write an **`append`** function which instruments its use of memory and can be used to test GC strategies. We will use the simple recursive append implementation:

```
1:  let rec append a b =
2:      match a with
3:      | [] -> b
4:      | h :: a' -> h :: append a' b
```

This is the desired function **`append`**. To implement it we must write **`appendUsingStack`** and express **`append`** with this and **`ManageCallTwoParameters`**.

```
 1: /// create an append function using StackFrames
 2: let appendUsingStack1 (frame:StackFrame) append =
 3:     let a,b = frame.["a"], frame.["b"]
 4:     match !a with
 5:     | IMatch(h,t) -> iCons h (append t !b)
 6:     | _ -> !b // this is always the INil case, but compiler does not know this!
 7:
 8: /// This append function can be used normal, but manages a mutable stack transparently of the user
 9: /// Note that the parameter names must match those needed by appendUsingStack or else there will be a run-time error!
10: let append = ManageCallTwoParameters "a" "b" [] appendUsingStack1
```

We also need to create some IList lists to test out append function. This creation must be done in a *safe* way so that the GC rule is obeyed as the lists are being created. For this we write a special function. Notice how all of the nodes in the partially constructed list are reachable from **`tmpVar`** which is on the stack.

```
 1: /// turn a float list into a Data item (float IList)
 2: /// the partly constructed IList is held in tmpVar on stack
 3: /// ensuring all Data cells are alive at all times
 4: let makeDataList (lst: float list) =
 5:     let oldStack = GlobalStack
 6:     let tmpVar = ref INil
 7:     GlobalStack <- Map.ofList [ "x", tmpVar ] :: oldStack
 8:     List.iter (fun f -> tmpVar := iCons f !tmpVar) (List.rev lst)
 9:     GlobalStack <- oldStack
10:     !tmpVar
```

Then we can test **`append`**.

```
1:  append (makeDataList [0.0 ; 1.0 ; 2.0]) (makeDataList [0.5 ; 1.5 ; 2.5])
```

Q5. Run the complete code so far which runs **`append`** as generated from **`appendUsingStack1`**. The GC fails, with an alive id not reachable from the stack. You should be able to work out where this happens and what is the mistake in **`appendUsingStack1`**. Rewrite this as **`appendUsingStack2`** and check that the complete code now works.

Answer

Q6. Write an instrumented function **`sortTwoNodeList: Data->Data`** that accepts a list of (guaranteed) two elements and returns a sorted list. Thus the function will either return its input, or a list made of the input with elements swapped. Use the helper function **`ManageCallOneParameter`** and note that you will need a second local variable (separate from the input) to implement the swap safely.

You will need to write **`sortedtwoNodeListUsingStack`**:

```
1:  let sortTwoNodeListUsingStack (frame: StackFrame) sortTwoNodeList = ...
```

Answer

Note that in code using higher order functions the types can nearly always be inferred. However it is recommended, for all except the simplest code, that you annotate functions with the types of their parameters as here. That will mean type errors are easier to debug because localised to the function that is actually wrong!

Garbage collection

The function **`gcAlloc`** can be rewritten to simulate different GC algorithms. The *simple code* performs a crude mark operation to calculate alive memory ids every time a list node is allocated, but never calculates free memory. New lists nodes have ever-increasing ids.

Given a fixed set of allowed ids, for example the range **`[0..N-1]`**, the free memory can be computed as the difference of this set and the set of alive ids. More efficiently we could perform a *mark and scan* garbage collection as follows:

1. Define a mutable boolean array **`alloc: int -> bool`** indexed by node id to record whether a node is allocated.
2. Define mark() and gc() functions as below

```
// C-like pseudocode
void mark(INode x)
{
    if (x == INil) return; // no marking needed for atomic data
    if (! alloc[x.Id]) {
        alloc[x.Id] = true;
        mark(x.Tl); // mark the rest of the list
    } else {
        return; // if node is marked it's list will already have been traversed so
                // no need to do this again
    }
}

void gc()
{
    // mark phase
    for (i=0; i < N; i++) alloc[i] = false;
    for <all variables v on stack> mark(v);
    // scan phase
    freeList := INil; // needed for first scan
    for (i=0; i < N; i++) {
        if (! alloc[i]) freelist = INode(Id = i ; Hd = 0.0 ; Tl = freeList);
    }
}
```

The list of ids to allocate is now contained in the free INode cells, all in freeList. Each allocation is a pop operation from this list:

```
int allocate()
{
    int id
    if (freeList == INil) {
        gc() // when no more free cells perform gc to collectnew free cells
    }
    if (freeList == INil) { /* error, memory full */ }
    // pop next id from free list
    id = freeList.Id;
    freeList = freeList.Tl
    return id;
}
```

Q7. Comment on the efficiency of this method when compared with the **`checkconsistency`** mark algorithm that operates without an **`alloc`** array. For data used in an **`append a b`** computation with the **`a`** list of length A and **`b`** list of length B estimate how much longer the simple code will take.

Answer

Q8. Implement this better GC method in **`gcAlloc`**.

(Q8 Answer not supplied)

Complete code

Tick Assessment

Rewrite these three mergesort functions like **`append`** to simulate memory allocation. How many list nodes (not counting the original list) are used to mergesort a 10 element list: *If the whole list is required* if only the head element is required, but the mergeSort code is still evaluated eagerly

Optional (no extra marks expected, but email me for feedback and a small number of possible challenge marks if you have a decent answer). Add to IList a *continuation* | **`ICont of continuation: Unit -> IList`**. Add to **`iHd`**, **`iTl`** so they will evaluate continuations. Rewrite **`mergeSort`** using continuations so that applications of **`iTl`**, **`iHd`** are kept as continuations and evaluate their parameters lazily - only when it is known this is required. Write a function **`firstElement: Data -> float`** that returns the head only of a mergeSorted list, discarding the rest of the list. What are the number of IList nodes that **`mergeSort`** - implemented in this lazy fashion - requires. Is the maximum alive number the same as the total number used? Does your answer change for different input lists? Can you explain your answers theoretically?