Maintaining the Stack for Recursion

Nitin Verma
mathsanew.com

October 26, 2020

Suppose there is a recursive method *recur*(). We want to create a variant of this which does not call itself and maintains its own stack to simulate the recursive calls. Thus recursion will still take place, but via our own stack instead of the one used for the program execution.

In this article, we will figure out how this can be done and apply it to a simple example. Further details may need to be worked out based on the given situation. We will refer to the new variant method as *recur_stack*(), the stack it will maintain as *vc_stack* ("virtual-call stack"), and the recursive calls which will get simulated now on this stack as "virtual-calls". The stack used for general function calls during program execution will be referred as "program-stack".

We can bring some ideas from the program-stack. So let us briefly look at how it works. It consists of stack-frames where each frame corresponds to an individual call. The order of frames on the stack is according to the order of calls which have been made till the current executing function, which corresponds to the top of the stack. Each frame is used to record the state of the individual call, which generally includes the local variables, input arguments, and instruction location to resume execution when the called function returns.

We will try to find out what kind of facilities are needed for simulating the recursive calls on our own stack.

Memory for Virtual-Calls

Generally, when a method like *recur*() keeps calling itself, it will also need to remember the memory state for each call on the call-stack. This state includes the local variables and input arguments. Each call on the call-stack also needs to remember where it should resume its execution when the called function returns. When we simulate the recursion on our stack, we too will need to associate some memory state for each virtual-call. We will refer to this memory as the *vc_frame* ("virtual call stack frame") of that virtual-call. Similar to the program-stack's frames, these frames will be pushed on the *vc_stack* to simulate a recursive virtual-call, and popped from it when the virtual-call returns. So, at any point, the top of *vc_stack* contains the *vc_frame* for the current executing virtual-call, the frames below it correspond to the earlier virtual-calls in their calling order. The bottom-most frame corresponds to the initial call of *recur_stack*() itself.

Loop for Repetition

Whenever a recursive method like *recur*() calls itself, it results in executing the complete method or a part of it again. And this repetition can take place any number of times. But the new method *recur_stack*() can contain no such recursive function calls. So we will need to arrange for repeating the logic of method *recur*() or a part of it, while continuing to execute within *recur_stack*(). A while-loop can be used for this. We will keep one iteration for every time the method *recur*() is called afresh by its caller (here a virtual-call). Also, when a virtual-call returns, we need to resume execution of the earlier virtual-call which had made that call. We will use a new iteration to resume executing the call (there is nothing like "returning" to an earlier iteration).

Iterations and Frames

Here is how the *vc_stack* and iterations will work together. At the beginning of *recur_stack*(), we set up the *vc_stack* with only one frame which specifies the state to use for the very first virtual-call. After this, we start the iterations. At the beginning of each iteration, we will read the topmost frame of *vc_stack*, which becomes the *current* frame for that iteration. Then we

execute some logic in the iteration according to the state specified in the current frame.

- When a recursive virtual-call has to be made, we will make any necessary updates in the current frame, setup the frame for the new call with its input arguments, push it on the *vc_stack*, and continue with a new iteration. In the new iteration, the new call would execute afresh.
- When the current virtual-call has to return, we will pop its frame from top of the *vc_stack*, and continue with a new iteration. In the new iteration, we resume execution of its "caller" using the caller's own frame (which is top of the stack now).

Resume-Points for Resuming

Whenever any general function call returns, the execution resumes in the caller just after that point of call. It happens with the help of program-counter saved on the program-stack during each call. But for resuming our virtual-calls, our implementation needs to figure out at which code location should the execution resume. We will refer to such locations as "resume-point". If there is only one place in the method *recur*() where recursive call is made, there is only one possible resume-point, and so during every time we resume, we can start execution at that resume-point.

But if there are more than one such recursive calls and hence multiple resume-points, it may not be trivial to figure out where to resume the execution. For this, we can number these resume-points uniquely, and keep a numeric variable *res_pt* ("resume point") in each *vc_frame*. Whenever a recursive virtual-call is being made, this variable is set in the current call's frame indicating where to resume this call later. Optionally, a special value 0 of *res_pt* can be used for a frame which is just being pushed to start a new virtual call, and so it will indicate a fresh call (not resuming).

Switch-Case for Jump

At the beginning of each iteration, we will read the current frame's *res_pt* and then will need to execute the logic according to that. For this, we can

organize the complete logic into multiple code-blocks, each for one resume-point. As said earlier, for a fresh call (not resuming) also, we can use a special resume-point of 0 and have its own code-block. Based on the *res_pt* value read, we will jump to that particular code-block and execute it. For convenience, a switch-case can be used, with *res_pt* as the switch value, and each code-block within its own case block for that resume-point.

As the method's logic will now get distributed across code-blocks, there can be a code-block B which is not only linked to a resume-point, but is also to be executed after another code-block A in continuity. To handle this, code-block A will simply modify the *res_pt* of the current frame to point it to B and, without a push/pop on the stack, continue to the new iteration. B will get executed in this new iteration. This technique will be used in the example below, where a recursive call is made only under an `if` block, but its resuming code-block is needed otherwise also.

Example

Below is an example implementation using the described approach, written in C language. For the recursive method *quick_sort*(), we create a variant *quick_sort_stack*(). The *vc_stack* is a fixed-size array, but other ways to store the stack can be used. Code for bound-checking the *vc_stack* array has not been shown.

```
typedef struct vc_frame
{
  char res_pt;
  int start;
  int end;
} vc_frame;

#define SWAP(arr, i, j) {int t; t = arr[i]; arr[i] = arr[j]; arr[j] = t;}

void quick_sort(int start, int end, int *arr)
{
  int pivot_loc, pivot;
  int i, j;

  pivot_loc = (start+end)/2;
  pivot = arr[pivot_loc];

  SWAP(arr, start, pivot_loc);

  i = start+1;
  j = end;

  while(1)
  {
    while(i <= end && arr[i] <= pivot)
      i++;

    while(j >= start+1 && arr[j] > pivot)
      j--;

    if(i < j)
      SWAP(arr, i, j)
    else
      break;
  }

  SWAP(arr, start, i-1);

  /* Recursion: sort the left partition */
  if(start < i-2)
    quick_sort(start, i-2, arr);

  /* Recursion: sort the right partition */
  if(i < end)
    quick_sort(i, end, arr);
}
```

```
void quick_sort_stack(int start, int end, int *arr)
{
  vc_frame vc_stack[10000];
  int top;

  /* Setup the frame for initial virtual-call. */
  vc_stack[0].res_pt = 0;
  vc_stack[0].start = start;
  vc_stack[0].end = end;
  top = 0;

  while(top >= 0)
  {
    /* Read the topmost frame from vc_stack.*/
    vc_frame *curr = &vc_stack[top];
    int s = curr->start;
    int e = curr->end;

    switch(curr->res_pt)
    {
      case 0:
      {
        /* This code portion is simply copied from quick_sort(): START */
        /* The local variables required only in this code-block... */
        int pivot_loc, pivot;
        int i, j;

        pivot_loc = (s+e)/2;
        pivot = arr[pivot_loc];

        SWAP(arr, s, pivot_loc);

        i = s+1;
        j = e;

        while(1)
        {
          while(i <= e && arr[i] <= pivot)
            i++;

          while(j >= s+1 && arr[j] > pivot)
            j--;

          if(i < j)
            SWAP(arr, i, j)
          else
            break;
        }
```

```
        SWAP(arr, s, i-1);
        /* This code portion is simply copied from quick_sort(): END */

        if(s < i-2)
        {
          /* Recursion: sort the left partition. Push a new frame on
             vc_stack. */
          vc_stack[top+1].start = s;
          vc_stack[top+1].end = i-2;
          vc_stack[top+1].res_pt = 0;
          top++;
        }

        /* Use curr's start/end to store the range of right partition,
           to use upon resuming this call. */
        curr->start = i;
        curr->end = e;
        curr->res_pt = 1;
        break;
      }

      case 1:
      {
        /* The earlier value of 'i' was saved in curr->start. */
        int i = s;

        if(i < e)
        {
          /* Recursion: sort the right partition. Push a new frame on
             vc_stack. */
          vc_stack[top+1].start = i;
          vc_stack[top+1].end = e;
          vc_stack[top+1].res_pt = 0;
          top++;
        }

        curr->res_pt = 2;
        break;
      }

      case 2:
      {
        /* Pop the current frame from vc_stack. */
        top--;
        break;
      }
    }
  }
}
```