CIS 425: Principles of Programming Languages
Lecture 12: memory management

Zahra Alsuliman

8 May 2019

1 Review of last lecture

What do we need to implement block structured languages? When we enter a block, we allocate all the space related to that block and when we exit we deallocate this space. This means that to implement blocks we follow a stack-based discipline (Last In First Out). We still follow the stack-based discipline in extending the language with first-order functions and functions passed as parameters. Here is a list of what we discussed:

1. First-order Functions (i.e. functions cannot be passed as arguments to functions and cannot returned by a function): To implement static scope we used a static link/access link, to return to the caller we used a dynamic/ control link. We also needed space for parameters, the return result address RRA and the return address (RA).

2. Functions in input: use the notion of a **closure** with is a pair containing a reference to the code and a reference to static environment.

Once we extend the language to allow to return functions, we **abandon the stack-based discipline**. All the activation records are allocated in the heap, and they are not deallocated when we exit a function.

Today's lecture:

1. Tail recursion (see Section 2)

2. Static vs dynamic scope (see Section 3)

3. Revisit the implementation of higher-order functions. Remind you that higher-order means that functions do not have any restrictions. We say that functions have the same status as an integer. We can pass integers in input or return an integer without any issue. So why functions should come with restrictions? (see Section 4)

4. Parameter Passing Techniques . We did not cover this point.

2 Tail recursion

We notice that in the languages we are studying, we do not have loops. We do everything using recursion, but it is an expensive method. Therefore, we use tail recursion to obtain the same efficiency of loops. A recursive function is tail recursive when the recursive call is the last thing executed by the function. In tail recursion we can reuse an activation frame/record for a recursive call to the function. This reduces the amount of space used by a recursive function.

Consider the factorial function below:

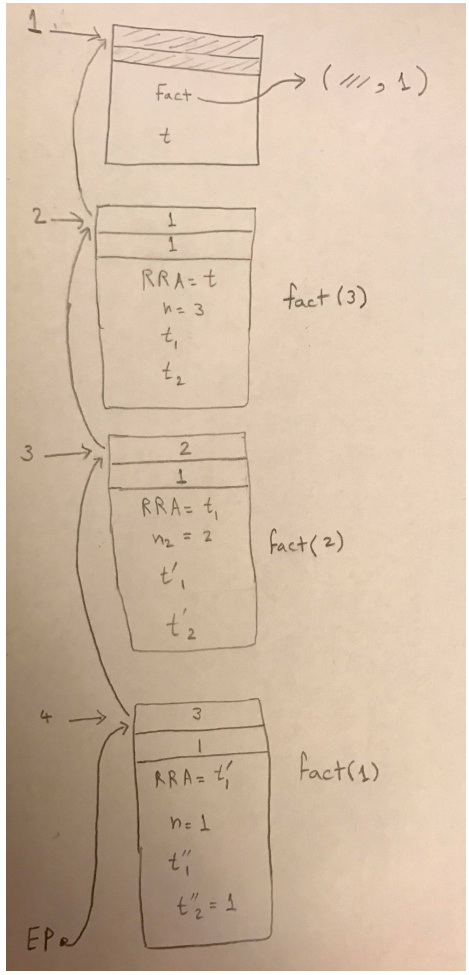
```
fun fact n = if n = 1 then 1 else n * fact(n-1);

val t = fact 3;
```

Let t1 be a temporary variable which stands for `fact(n-1)` and let t2 be a temporary variable which stands for `n*fact(n-1)`. Draw the activation frame as follows:

Step 1: Create the activation record numbered 1. Allocate space for the function fact and the variable t. We use a closure for fact and we put 1 in the second pair because it's the first block that contains the definition.

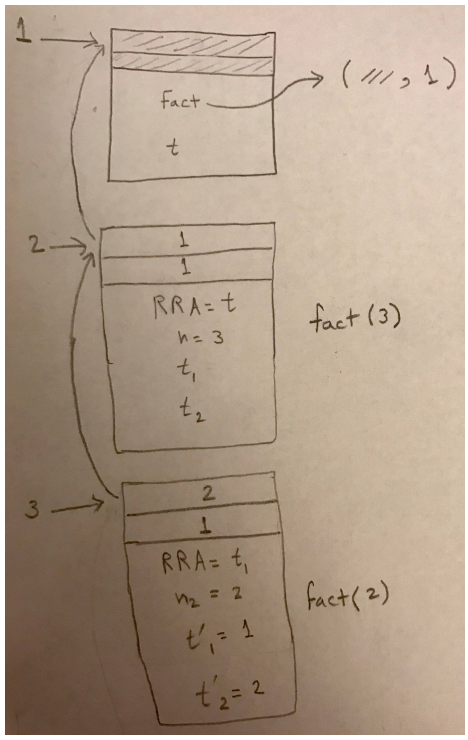
Step 2: Create the activation record numbered 2. For the control link: use activation frame 1 because it's the calling frame. Static link points to the activation frame number1 because it is where fact is defined. Allocate space for the RRA, the variable n = 3. Allocate space for t1 and t2 but we don't initialize them because we don't know their values.

Step 3: Create activation record numbered 3. Control link points to frame 2. Access link points to the activation frame 1 where fact is defined. We draw activation records for 2, 3, and 4 and allocate spaces of the content of the blocks. In each creation of an activation record, we update the environment pointer. This is what we get when we reach fact 1:

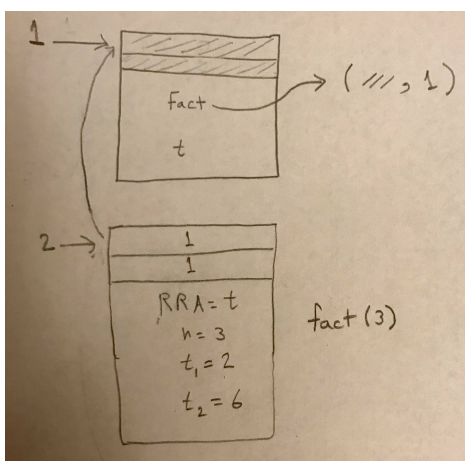


We have saved the result of the recursion, which is 1, in t2" and we are ready to pop the frames as follows:

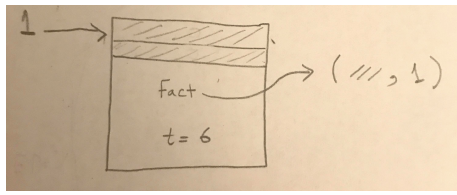
We deallocate activation frame number 4 and store the results in t1', which is 1, and in t2' we store the result of n *t1' , which is 2 :



We deallocate activation frame number 3 and store the results in t1 (which is 2) , and in t2 we store the result of n * t1:



We deallocate activation frame number 2 and store the results, which is 6, in the variable t in the first activation record:

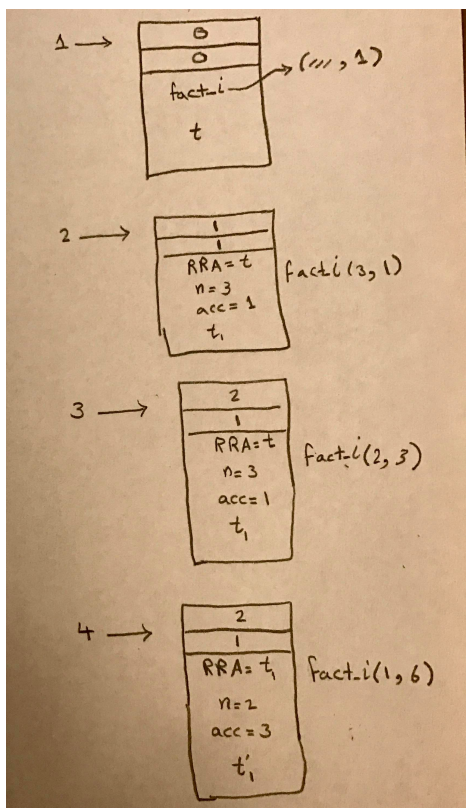


The point here is that when we come back from each recursive call, I have some work to do.

Let's now write the factorial function using tail recursion:

```
fun fact-i(n, acc,) = if n = 1 then  acc  else  fact-i(n-1, n*acc);

val t = fact-i(3, 1);
```

Let's draw the activation frames:

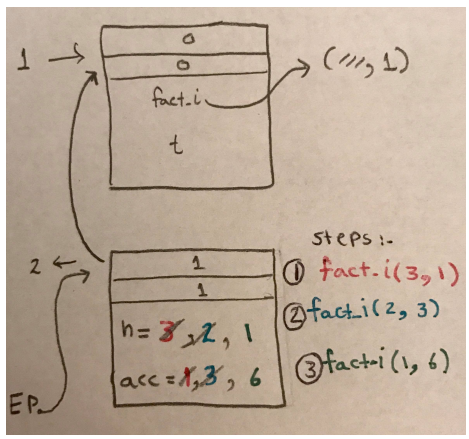


Notice that when we deallocate an activation frame, we do not have to do any work other than returning to the caller again. This suggest that we can do a better job in the implementation. We can execute this function by reusing the

same activation frame and updating it. Steps to do this:

Step 1:
Create activation frame 1 as previously.

Step 2:
Create activation frame 2 as follows and update it with each recursive call as follows:



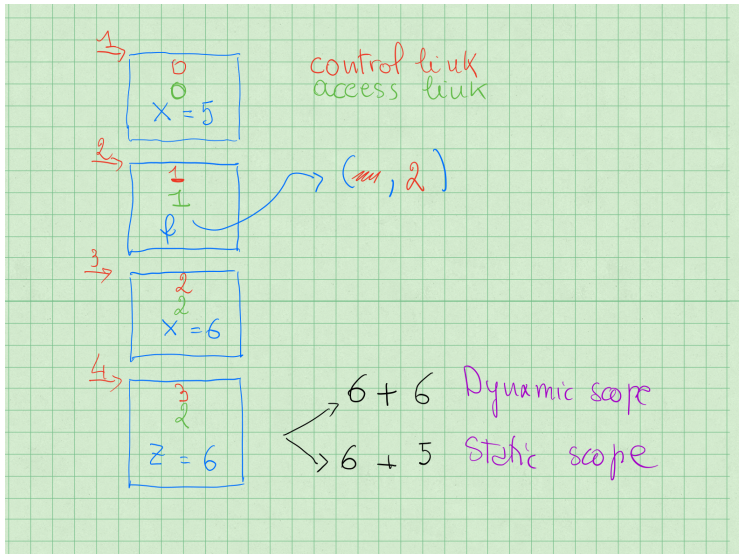
The takeaway: In the non-tail recursion factorial, the definition of factorial allocates an activation record for each invocation of the call because we need to access n so we cannot deallocate n. In tail recursion, the definition is equivalent to a loop. It executes as fast as a loop because we do not need to allocate a separate frame for each invocation of the call. So, we can execute the factorial using just one frame.

3 Difference between static and dynamic scope

Consider this code:

```
let val x = 5
in
  let
    fun f z =  z + x
     in
       let
         val x = 6
        in f x
       end
     end
  end
```

Drawing of the activation record:



What is the result of this program?

Result under static scope:

To know the definition of a variable we look in the closest enclosing scope of the program. We don't need to execute the program to link variable occurrences to their definitions. The advantage of static scope is that for each variable occurrence, we can link it to its definition without executing the program.

In the current example, when we execute x + z, we get the value of x by following the static link. In frame 4, the static link is referring to frame 2, so we go to frame 2. Then, the static link in frame 2 is referring to frame 1, so we go to frame 1. From frame 1, we get the value of x which is 6. We do the computation (z + x) that is (6 + 6).

Result Under dynamic scope:

To know the definition of a variable, we look to the identifier associated with the most recent activation record. You get the last variable you allocated. So, after we reach frame 4, to compute z + x, we need to follow the dynamic link to access the right x. The dynamic link of frame 4 is referring to frame 3. So, we go to frame 3 and we find the value of x which is 6. Hence the result of computing (z + x) is 6 + 6.

Note that if we only have blocks, then the static link and control link will be the same because the block calling the next frame is the same as the block enclosing the next frame.

For reasoning, static scope is much better because the compiler can analyze the program.

Renaming:

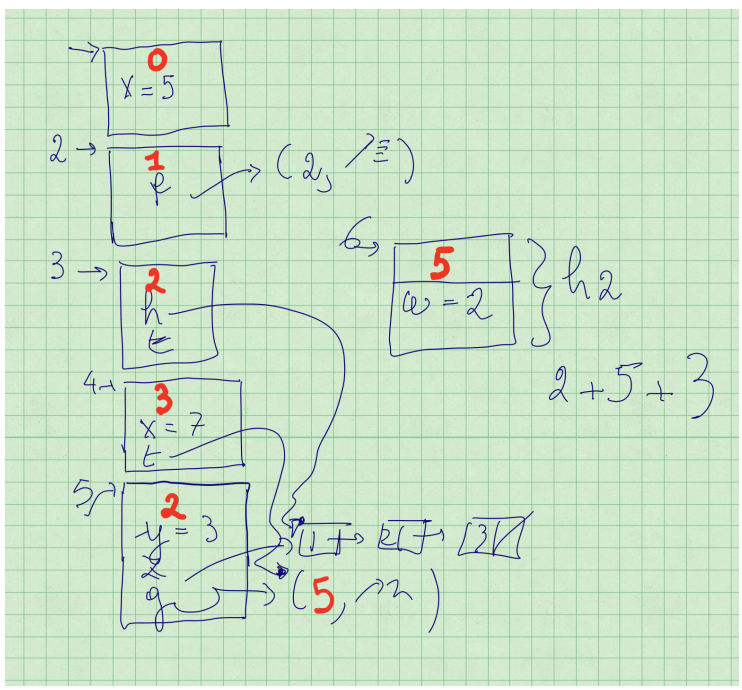
You can freely rename variables with static scope. However, you cannot do the same thing with dynamic scope because when you rename, you change the behavior of the program.

4 Functions in output

Example 1 Consider the code:

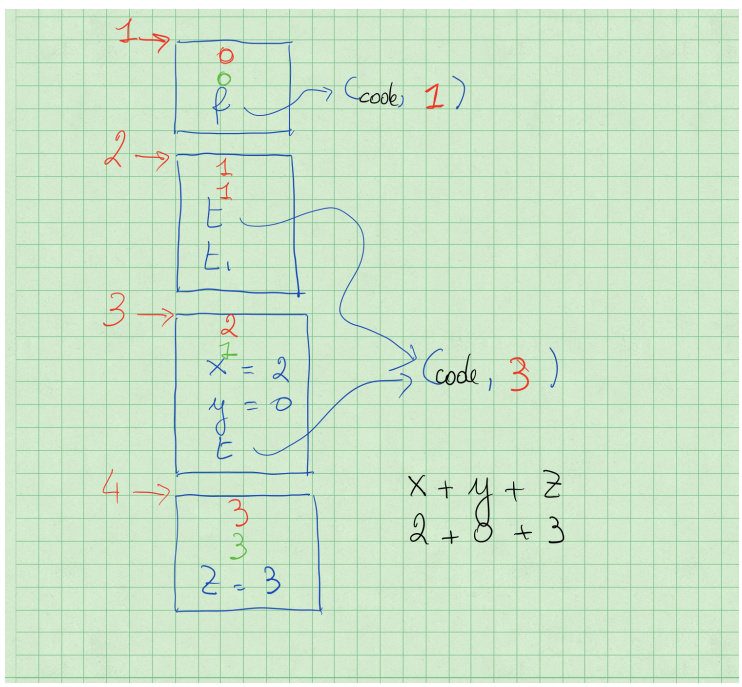
```
let
val x = 5
fun f y = let val z = [1, 2, 3]
                    fun g w = w + x + y
              in g
              end
val h = let val x = 7
                 val t = f 3
            in t
            end
t = h 2
```

We have (we are not showing the dynamic link):



Note that when we call function h the activation record 5 has not been deallocated. This means that once functions are allowed to be return as a result, the stack discipline has to be abandoned.

Example 2 Consider the following code:

```
let
fun f x = let val y = 0
              in fn z => x + y + z
              end
in  let
       val t = f 2
    in   t 3
   end
end
```

We have: