Concepts of Programming Languages
Lecture Notes: Week 6—Tail Recursion

Stefan Mitsch

School of Computing, DePaul University
smitsch@depaul.edu

Tail Recursion

We well discuss tail recursion in terms of the details of the implementation of function calls. Remember that function calls store local data on the *stack* in the form of activation records (or stack frames). When we call a function, a new activation record is pushed onto the stack, on return we pop the topmost activation record and return to the caller, whose activation record becomes then active (new topmost on the stack). Since function calls and returns cannot be arbitrarily nested (a callee is called after its caller, and returns before it), a stack is the most natural data structure for storing activation records. Since memory in a computer is finite, however, there are limits to the size of the call stack, which can cause problems with deep recursion.

Let's consider a recursive definition of a count-down in C below:

```
1  int count_down (int x) {
2    if (x == 0) {
3      return 0;
4    } else {
5      return 1 + count_down (x - 1);
6    }
7  }
8
9  int main (int argc, char **argv) {
10   long num = strtol (argv[1], NULL, 10);
11   count_down (num);
12   return 0;
13 }
```

For some arguments this code will succeed, for others it produces a segmentation fault when the computer runs out of stack size. Note that, in C, this behavior depends on the operating system and on the execution context: we won't necessarily find a limit, but will see behavior that some calls fail and then some with larger numbers succeed. The tendency is that the higher the argument the more often we will see the program fail. In Linux, we can modify some of those limits with shell arguments. In interpreted languages, such as Scheme, the same behavior occurs, but just like in C, it depends on the runtime environment. In Java, you may also have encountered a runtime exception of type StackOverflowError; this usually happens when we make a programming mistake in a recursive algorithm, but it can also happen in perfectly sensible code (for some definition of sensible) that just happens to exhaust the stack limits (e.g., code that has lots of backtracking options).

Intuitively, in the recursive definition of countDown in Scala below, each (1+...) represents a new activation record, because we must store the intermediate 1+ that still need to be summed up.

```
def countDown (x:Int) : Int = if x == 0 then 0 else 1 +
    countDown (x - 1)

// countDown (4)
// --> 1 + countDown (3)
// --> 1 + (1 + countDown (2))
// --> 1 + (1 + (1 + countDown (1)))
// --> 1 + (1 + (1 + (1 + countDown (0))))
// --> 1 + (1 + (1 + (1 + 0)))
```

Let's take a more detailed look under the hood to find out why we have a call stack.

Tail-recursive calls have a special structure that has all recursive calls in tail position, which means that all recursive calls happen as the final step on the call stack. In assembly language, that corresponds to jumps at the very end of each recursive call, rather than a jump and then another operation after. This opens up the possibility for an optimization called *tail call optimization* that performs the work in a loop without creating new activation records on the stack. Such an optimization is vital when algorithms have potentially exponential work to perform, which can exhaust stack limits quickly. The optimization includes mutually recursive calls (f calls g, which calls f).

Tail-Recursive Implementations

The C code below computes the sum over a list of integers in a tail-recursive method.

```
typedef struct node node;
struct node { int item; node *next; };

int sum_aux (node *data, int result) {
  if (!data) {
    return result;
  } else {
    return sum_aux (data->next, result + data->item);
  }
}

int sum (node *data) {
  return sum_aux (data, 0);
}
```

When inspecting the assembly language of that code, we can clearly see the loop behavior:

```
sum_aux:
  testq   %rdi, %rdi
  movl    %esi, %eax
  je      .L7
.L9:
  addl    (%rdi), %eax
```

```
7     movq    8(%rdi), %rdi
8     testq   %rdi, %rdi
9     jne     .L9
10  .L7:
11    rep
12    ret
```

Lines 5–9 advance dereference the pointer, then advance it, then test for being null and jump back to Line 5 if not null. If null, the program continues into `.L7` and returns the result.

Let us now inspect some behavior of non-tail-recursive functions and to practice rewriting of functions that are not tail-recursive into functions that are.

```
1  def longList(n: Int) =
2    if n==0 then List(1)
3    else
4      val sublist = longList(n-1)
5      sublist ::: sublist
6  end longList
```

First, let's make sense of the code: the function `longList` takes an argument n to produce a result list of length 2^n. It does so by doubling the length of the list on every n-decrement step using `:::`.

Exercise 1 (Tail-recursive or not?). Why is this function not tail-recursive?

Think for yourself before you read on. The recursive call to `longList` is not the very last step in the else-branch of `longList`.

Exercise 2 (Space complexity?). What is the space complexity of the function?

Think for yourself before you read on. Exponential in n

Exercise 3 (Runtime complexity?). How many recursive calls are made?

Think for yourself before you read on. Linear in n

Exercise 4 (Bad or ugly?). Is it necessary to store the created sublist in a variable or could we just directly make recursive calls as part of the last statement?

Think for yourself before you read on. If we're not reusing the result we end up with the Fibonacci disaster of exponential in n calls to `longList` instead of linear in n.

In order to make it tail-recursive, we follow an approach resembling forward vs. backward processing of folds, as seen in Lecture "Folds".

First, a non-tail-recursive function computing the sum of the elements in a list.

```
1  def sum(xs: List[Int]) = xs match
2    case Nil       => 0
3    case x :: tail => x + sum(tail)
4  end sum
```

Exercise 5 (Make it tail-recursive). How do we need to change the code above to implement `sum` in a tail-recursive way?

Think for yourself before you read on. To make it tail-recursive, we need to first compute a partial result from the current element and the previous partial result. For this, we need to introduce an additional argument to pass in partial results.

```
1 def sum(xs: List[Int], result: Int = 0) = xs match
2   case Nil       => result
3   case x :: tail => sum(tail, result+x)
4 end sum
```