SWE 212 Analysis of Programming Languages

Week 3 Summary of 'Recursive Programming' by E. W. Dijkstra

Samyak Jhaveri

In this paper, Dijkstra introduces the concept of recursion implemented using runtime stack

The paper starts with mentioning the problems with subroutines having their own private fixed working spaces. All the subroutines together would take up too much storage space than they would need if they operated simultaneously. Also, it is not possible to call a subroutine when one of its previous activations is still in use and being computed. Such problems would not allow recursion to be used with procedures since if a procedure calls itself many times, soon there would not be enough memory space for that procedure. A stack can be used to store a sequence of information units that increase or decrease at one end only. This could be used in programming by using an administrative quantity called "stack pointer" that would increase in value adding information units and decrease in value upon removing an information value. For instance, when a function is to be evaluated in a given subroutine, the subroutine should be arranged in a way such that it operates in the first free places of the stack.

When using a stack with a function present in a subroutine, the stack is filled by each intermediate result obtained from the previous operation. As the function's variables get evaluated, with every operation the stack becomes more vacant as the operands and operators are used.

For instance, if `C` happened to be an expression to be evaluated, instead of a numerical value, at the end of the evaluation of that expression, `C` would be the same as if it were a value stored in memory. This seems intuitive to the modern programmer but was a radical thought when this paper was first published.

All the variables to be used in this function are stored freely in the stack.These variables can be of mainly three types

1. **Parameters:** Parameters are pieces of information that are presented to a subroutine. To use parameters with a subroutine, the first free place in the stack is left empty for the subroutine to place its "function value" in it. The consequent places in the stack are for the parameters. It is a convenient idea to have the same storage size allocated to all the parameters i.e. they acquire the same number of places in the stack.
2. **Local Variables:** Local variables are the ones that are limited to the scope of the subroutine and will not be relevant before or after that instance/activation of the subroutine. They can work with the parameters' values to perform the right operations , but unlike parameters they do not occupy a fixed number of places in the stack. The amount of storage they occupy depends on the parameters they work with.
3. **Most "anonymous" intermediate results:** During its execution, a subroutine might call one or more subroutines as expressions get evaluated and stored into an intermediate result. This intermediate result is anonymous and is placed in the stack the same way as that of the main program, but now the control must begin with the first free place following the last local variable.

As the main program uses stack exclusively for storing intermediate numerical information, only the topmost place inthe stack was the relevant interest at any point in the subroutine. This meant that there was no need for random access memory to store the most anonymous intermediate results of a subroutine of the stack. However, for interest in local variables, some kind of deeper access needed to be provided, which meant the employment of random access memory to store their addresses. The point in the stack from which the local variable is available for a subroutine is handed over to the subroutine when it is called. The static reference of the local variables can only occur as a fixed position w.r.t the reference point in the stick, where the value of this reference point is dynamically derived from the stack pointer at the moment of the call, which means that it may vary from call to call.

With a final flourish, Dijkstra goes on to show that ‘link’ information must be preserved in the stack when calling subroutines, so that regardless of complexity we can pick up exactly where we left off (in the ALU)- this is to include a return address. This ‘link’ should consist of the data regarding the state of the arithmetic unit as it goes through a series of changes, to be able to reconstruct it later. Furthermore, a ‘return address’ should also be part of the ‘link’.

In the case of a subroutine A, the parameters(including thelink) and the local variables of A all have a fixed position with respect to each other but their position as a whole may vary, which is indicated by the parameter pointer. Since the current value of the parameter pointer of A has to be available at all times during the execution of subroutine A in the arithmetic unit, it is stored with the rest of the link data. Let subroutine A call another subroutine B with their parameter pointer values being indicated by PPA and PPB. Upon being called, PPA’s value is recorded as part of the link for this call. Now, as the link’s place is deduced from the value of the stack pointer, which now assumes the value of PPB. When the control returns from B to A, the return address is found at the PPB’s place in the stack as well as in PPA. Finally, the ‘return’ operation decreases the stack pointer’s value and can be deduced from the old value of the PPB.

This is the core concept behind recursion as this same process can be used by subroutine A to call itself in the middle of its execution in the arithmetic unit. Since hte subroutine only has to appear in memory once, and since it can have multiple incarnations, the innermost activation causes the same piece of text to work in a higher part of the stack. This is what allows for recursion to happen!