Transforming Recursion to Iteration in Programming

Athanasios Tsadiras
Technological Educational Institute of Thessaloniki, Greece

INTRODUCTION

The main advantage of a Recursive Algorithm (an algorithm defined in terms of itself) is that it can be easily described and easily implemented in a programming language (van Breughel, 1997). On the other hand, the efficiency of such an algorithm is relatively low because for every recursive call not yet terminated, a number of data should be maintained in a stack, causing time delays and requiring higher memory space (Rohl, 1984). Solving the same problem iteratively instead of recursively can improve time and space efficiency. For example, to solve a problem that involves N recursive procedure calls, it will require stack space linear to N. On the contrary, using iteration, the program will need a constant amount of space, independent of the number of iterations. There are **programming languages**, such as **Prolog**, that do not possess built-in iterative structures and so recursion should be used instead. Nevertheless, there are ways to write recursive programs that have similar behaviour with that of the corresponding iterative programs.

BACKGROUND

The transformation of **recursion** to **iteration** is not a new problem and it has been studied by various scientists, for example, De Moor and Sittampalam (2001) and Clinger (1998). This competition between recursion and iteration is interesting because they represent two different schools of thought in Computer Science. Various programming languages use **recursion** extensively, especially high-level programming languages that cope with Artificial Intelligence problems (Luger, 2002) such as Prolog (Bratko, 2000; Ramachandran, 1986), LISP (Lamkins, 2004; Seibel, 2005) or Scheme (Dybvig, 2004; Watson, 1996). The Prolog language will be used to exhibit the transformation of recursion to iteration (Clocksin & Mellish, 2003; Shoham, 1994). **Prolog** is a **Logic Programming Language** (Bramer, 2005) that uses heavily recursion. This happens because Prolog lacks iterative structures such as "for," "while-do" or "repeat-until" (Holmes, 2001; Langfield, 2003). In Prolog, a clause can be iterative even if it contains a recursive call. That is, a Prolog clause is iterative if it has zero or more calls to Prolog system predicates before the recursive call (Sterling & Shapiro, 1986). Furthermore, a Prolog procedure is iterative if it contains only unit clauses (facts) and iterative clauses. Having these in mind, we will try to transform **recursion** to **iteration** in Prolog, but that can apply to all other languages that use recursion.

TRANSFORMING RECURSION TO ITERATION

The fact is that there is no easy or general way to transform a recursive algorithm to an iterative algorithm. What a programmer can do to increase the efficiency of a recursive algorithm is to implement the recursive algorithm in an iterative manner. This can be done by using special variables called accumulators that will be used to keep intermediate results and facilitate acceleration.

To demonstrate the technique, the calculation of the sum of all integers from 1 to N will be used. The recursive relation that can be used to calculate the sum S of all positive integers from 1 to N is S(N)=S(N-1)+N. A Prolog predicate sum(N,S) that evaluates sum S of all integer numbers from 1 to N, is the following:

```
sum(1,1).
sum(N,S):-N1 is N-1,
          sum(N1,S1),
          S is S1+N.
```

This is a recursive implementation because the second clause involves the call of the same predicate and after it, there is a Prolog system call (S is S1+N). To illustrate the computational effort of the above implementation, the trace of the call to find the sum of all integers from 1 to 4 is shown below (Figure 1).

In Figure 1, we see that after a series of recursive calls the call sum(1,S3) stops the recursion and initiates a series of value returns, until the initial call sum(4,S) is reached. The common process of recursion is now apparent where first the problem is getting smaller and smaller, until it meets the limit case. After that, values of intermediate call are returned until the initial call is answered. If we call the same predicate for an integer that is above a certain limit, the computer will

Figure 1. The trace of the recursive implementation

```
?-sum(4,S).
  → sum(3,S1)
    → sum(2,S2)
      → sum(1,S3)
      ← sum(1,1)
    ← sum(2,3)
  ←sum(3,6)
 sum(4,10)
S=10
```

Figure 2. Trace of the iterative implementation, involving two additional parameters, one accumulator and one counter

```
?-sum_iterative1(4,S).
 → sum1(1,4,1,S)
   → sum1(2,4,3,S)
     → sum1(3,4,6,S)
       → sum1(4,4,10,S)
S=10
```

reach stack overflow because it cannot afford to provide memory space for all the recursive calls.

We will try to implement the same algorithm in an iterative manner using predicate sum_iterative1(N,S). This will require the introduction of two new parameters. The first one will play the role of the **counter** of the iteration and the second will play the role of the **accumulator**. That is, the predicate sum_iterative1(N,S) will call an augmented predicate sum1(I,N,T,S) that has the two additional parameters. Counter "I" represents the I-th iteration and accumulator "T" the temporal sum until iteration I. Both the counter and the accumulator should be initiated with value 1. For this reason, the call of sum_iterative1(N,S) lead to the call of predicate sum1(1,N,1,S) that has the first and the third argument equal to 1. The complete code is given below.

```
sum_iterative1(N,S):-
sum1(1,N,1,S).

sum1(I,N,T,S):-
      I<N,
      I1 is I+1,
      T1 is T+I1,
      sum1(I1,N,T1,S).
sum1(N,N,S,S).
```

Predicate sum1 is iterative because, as it is mentioned in the "Background" section, after the pseudo-"recursive" call at sum1(I1,N,T1,S), there is no other call. This kind of **iteration** is also called **Tail Recursion**. The introduction of the 2 parameters leads to the evaluation of the partial result of the sum at each step of the iteration and cause the termination of the iteration when counter I becomes equal to N. It is apparent that the recursion is transformed into an iteration of N steps. The iterative nature of the above implementation is illustrated in Figure 2, which shows the trace of the call to find the sum of all integers from 1 to 4.

Comparing this trace with that of Figure 1, we see that now the solution is found as soon as the call reaches the limit case at sum1(4,4,10,S). This means that the second stage of returning values and making calculations that is present in the trace of Figure 1 is now avoided, shortening the whole process.

The same problem can be solved iteratively even without having the counter I described above. In this case only one additional parameter, that of the accumulator, will be needed. The accumulator will store the partial result up to the current step of iteration. The accumulator should be initiated with value 0; this is why the second argument of the call of predicate sum2(N,0,S)is zero. The code is given below:

```
sum_iterative2(N,S):-
sum2(N,0,S).

sum2(N,T,S):-
      N>0,
      T1 is T+N,
      N1 is N-1,
      sum2(N1,T1,S).
sum2(0,S,S).
```

In this second implementation variable N also plays the role of the counter, because at every iteration step, its value decreases by one, terminating when its value becomes zero. Once again, the recursion is transformed into an iteration of N steps. This can be shown with the following trace of the call sum_iterative2(4,S).

The trace of Figure 3 is similar of that of Figure 2, that is, the second stage of returning values found in Figure 1, is omitted, and additionally, needs less memory space than the implementation traced at Figure 2, because only one additional parameter is added instead of two.