Tail Recursive Programming by Applying Generalization

Manuel Rubio-Sánchez
Departamento de Lenguajes Sistemas Informáticos I
Universidad Rey Juan Carlos, c/ Tulipán s/n, 28933 Móstoles, Madrid, Spain
manuel.rubio@urjc.es

ABSTRACT
The design of many tail recursive algorithms can involve thinking about the status of variables and parameters, and how these change with execution flow. In other words, tail recursion is closely related to iteration and imperative programming. However, it is possible to derive tail recursive functions by exclusively using concepts inherent in recursion, such as declarative programming, induction, or problem decomposition. This paper proposes a simple methodology for designing tail recursion functions by using a declarative approach and the concept of function generalization. We have carried out an evaluation of the technique with second and third-year computer science students. Results suggest that this new point of view improves students’ ability to design tail recursive programs, helps them understand the distinction between the imperative and declarative paradigms, and may reinforce their programming skills in general. Furthermore, students found the methodology easy to learn and apply, simpler than more sophisticated formal methods, and described it as fast and methodic or mechanical, as it involves a sequence of well-defined steps.

Categories and Subject Descriptors
D.1.1 [Programming techniques]: Applicative (Functional) Programming—Recursion; F.3.3 [Logics and meanings of programs]: Studies of Program Constructs—Program and recursion schemes; K.3.2 [Computers and Education]: Computer and Information Science Education—Computer Science Education

General Terms
Algorithms

Keywords

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

Copyright 2010 ACM 978-1-60558-820-9/10/06 ...$10.00.

int f_tail(int x, int y){
    if(x==0)
        return y;
    else
        return f_tail(x-1,x*y);
}
int fact_tail(int x){
    return f_tail(x,1);
}

Figure 1: Comparison of tail recursive (a), and iterative (b) factorial functions.

1. INTRODUCTION
Recursion is a key concept in computer science and mathematics, and plays an important role in the acquisition of competencies regarding problem decomposition, induction, or functional abstraction [8]. It is is a powerful problem-solving tool, which constitutes an attractive alternative to iteration, especially when problems can be solved using a divide and conquer approach. However, CS students generally find recursion hard to master. Numerous authors have tried to identify the factors (conceptual models, cognitive learning styles, functional abstraction reasoning) responsible for these difficulties [10, 8], while others have proposed methods and strategies designed to overcome these problems [5, 3, 9, 2, 6].

Traditionally, several types of recursive algorithms are studied in CS courses, which can be categorized according to the number and type of the recursive function calls within a procedure. A common classification distinguishes the following types: linear, tail (which is a special case of linear recursion), multiple (or exponential), nested, and mutual recursion [7]. Among these, tail recursion is particularly different from the rest, since many tail recursive algorithms may be designed by thinking about the status of variables and parameters, and how these change with execution flow. In other words, tail recursive programming may rely more on iteration than on the key concepts inherent in recursion (problem decomposition, induction, declarative
Table 1: Results of a pretest on iteration (a), linear non-tail recursion (b), and tail recursion (c).

<table>
<thead>
<tr>
<th></th>
<th>CS2</th>
<th>CS3</th>
</tr>
</thead>
<tbody>
<tr>
<td>(a)</td>
<td>0 1 24</td>
<td>0 1 28</td>
</tr>
<tr>
<td>(b)</td>
<td>0 1 24</td>
<td>1 1 26</td>
</tr>
<tr>
<td>(c)</td>
<td>6+10 2 7</td>
<td>9+5 0 14</td>
</tr>
</tbody>
</table>

Table 1: Results of a pretest on iteration (a), linear non-tail recursion (b), and tail recursion (c).

Recently, in experiments carried out with 25 second-year and 28 (different) third-year CS students, we observed that they had difficulties designing tail recursive functions, despite having studied recursion in previous courses. In a first preliminary test, students were given 25 minutes (we did not observe difficulties due to lack of time) to write: (a) an iterative factorial, (b) the sum of the first $n$ positive integers with linear non-tail recursion, and (c) the power function using tail recursion (see Sec. 2.2.1). Their solutions were graded as correct ($\checkmark$), partially correct (+) when the design seemed to be appropriate but a minor error was detected, and incorrect (X). Table 1 shows the results of the experiment. Clearly, and despite the fact that the power function involves one more parameter, students’ performance dropped for the tail recursive function. The table also shows an important detail: 10 out of the 16 CS2 students and 5 out of the 14 CS3 students wrote non-tail recursive power functions, which indicates that a considerable percentage of students did not fully understand the difference between tail and non-tail recursion, especially in the CS2 group.

This paper describes a methodology for deriving tail recursive functions that is based exclusively on declarative programming, etc.). Figure 1 shows an example where the relationship between tail recursion and iteration is apparent. The recursion tree for the tail recursive factorial is identical to the table of variable values that are obtained at the beginning of each loop cycle for an iterative version. Note that in both cases $y$ acts as an accumulator variable.

In Sec. 4, we analyze the results. Finally, a discussion is reported in Sec. 5. This section presents several examples of the usage of the proposed methodology. The covered functions are typically studied in CS1 programming courses and research papers on teaching recursion [4].

2.2.1 **Power**

1. The first step consists of defining the function we wish to implement:

   $g(b, e) = b^e$

   where $b \in \mathbb{R}$, and $e \in \mathbb{N}$. We could have chosen another notation such as $\prod_{i=1}^{e} b$, but the first one is simpler.

2. We then add a parameter to create the general function:

   $f(b, e, y) = y \cdot b^e$

3. The base case for $g$ is $g(b, 0) = 1$. For $f$ it is $f(b, 0, y) = y$.
4. Therefore, we can decrement the exponent $(e-1)$ in order to reach the base case.
5. In this step we form the following equation:

   $f(b, e, y) = f(b, e-1, z)$
with unknown \( z \).  

6. Solving for \( z \) yields \( z = y \cdot b \). Steps 5 and 6 can be illustrated as: 

\[
\begin{align*}
\text{f}(b, e, y) & \quad \rightarrow \quad y \cdot b^e \\
\text{f}(b, e - 1, z) & \quad \rightarrow \quad z \cdot b^{e-1} \quad \Rightarrow \quad z = y \cdot b
\end{align*}
\]

7. The tail recursive algorithm is therefore: 

\[
\text{f}(b, e, y) = \begin{cases} 
    y & \text{if } e = 0 \\
    \text{f}(b, e - 1, y \cdot b) & \text{if } e > 0
\end{cases}
\]

8. Finally, \( g(b, e) = f(b, e, 1) \).

2.2.2 Does An Integer Contain An Odd Digit? 

Consider a positive integer expressed in base 10, i.e., \( x = (d_m \cdots d_1 d_0)_{10} = \sum_{i=0}^{m} d_i \cdot 10^i \), where \( d_m \neq 0 \), and let \( \text{div} \) and \( \mod \) represent integer division and modulo operations. The next (and more complicated) example is based on the Boolean function that returns a true value if a positive integer contains an odd digit. 

1. The first step consists of obtaining a logical formula for the function: 

\[
g(x) = \bigvee_{i=0}^{m} \text{odd}(d_i) = \text{odd}(d_m) \lor \text{odd}(d_{m-1}) \lor \cdots \lor \text{odd}(d_0)
\]

where \( \text{odd}(d) \) is a Boolean function that evaluates to true if digit \( d \) is odd. 

2. The general function can be \( f(x, y) = y \lor \bigvee_{i=0}^{m} \text{odd}(d_i) \). 

3. The base case is \( g(x) = \text{odd}(x) \) when \( x \) has only one digit. Therefore, the base case for \( f \) can be \( f(x, y) = y \lor \text{odd}(x) \) if \( x < 10 \), or simply \( f(x, y) = y \) if \( x = 0 \). 

4. The natural simplification is \( x \text{ div } 10 \), which reduces the number of digits of the first parameter. 

5. We then form the equation \( f(x, y, z) = f(x \text{ div } 10, z) \): 

\[
\begin{align*}
f(x, y) & \quad \rightarrow \quad y \lor \bigvee_{i=0}^{m} \text{odd}(d_i) \\
f(x \text{ div } 10, z) & \quad \rightarrow \quad z \lor \bigvee_{i=0}^{m-1} \text{odd}(d_i) \quad \Rightarrow \quad z = ?
\end{align*}
\]

6. It might appear difficult to solve for \( z \). However, noticing that \( m - 1 = m' \) and \( d_{i+1} = d'_i \), we have: 

\[
\begin{align*}
f(x, y) & \quad \rightarrow \quad y \lor \bigvee_{i=0}^{m} \text{odd}(d_i) \\
f(x \text{ div } 10, z) & \quad \rightarrow \quad z \lor \bigvee_{i=1}^{m} \text{odd}(d_i) \quad \Rightarrow \quad z = y \lor \text{odd}(d_0)
\end{align*}
\]

where \( z \) can now be solved easily. 

7. Putting everything together yields: 

\[
f(x, y) = \begin{cases} 
    y & \text{if } x = 0 \\
    f(x \text{ div } 10, y \lor \text{odd}(x \mod 10)) & \text{if } x > 0
\end{cases}
\]

8. Finally, \( g(x) = f(x, \text{FALSE}) \).

2.2.3 Factorial Revisited 

Besides being able to express a function using appropriate notation, it is important to select an adequate generalization in order to obtain a clear and efficient algorithm. However, it is not crucial for obtaining a correct program. For example, returning to the factorial function, several generalizations lead to correct algorithms (e.g., \( x! / y, y + x!, (x!)^n \), etc.). Let the general function be \( f(x, y) = y + x! \). By considering \( x - 1 \) as a simplification we have: 

\[
\begin{align*}
f(x, y) & \quad \rightarrow \quad y + x! \\
f(x - 1, z) & \quad \rightarrow \quad z + (x - 1)!
\end{align*}
\]

where 

\[
\begin{align*}
z & = y + x! - (x - 1)! = y + (x - 1) \cdot (x - 1)! = \\
& = y + (x - 1) \cdot f(x - 1, 0)
\end{align*}
\]

which leads to a correct algorithm based on nested recursion (it is not tail recursive since it has a binary recursion tree): 

\[
f(x, y) = \begin{cases} 
    1 + y & \text{if } x = 0 \\
    f(x - 1, y + (x - 1) \cdot f(x - 1, 0)) & \text{if } x > 0
\end{cases}
\]

In general, adding a variable when the formula involves a product, or multiplying by the new parameter when the formula includes a sum, leads to nested recursion. For the factorial, the generalization \((x!)^n\) also leads to nested recursion. Finally, the type of the extra parameters should also be taken into account. If the general function is chosen to be \( f(x, y) = x! / y \), then \( f(x, y) = f(x - 1, y / x) \), where the type of second parameter must be real instead of integer.

3. DESCRIPTION OF THE EVALUATION 

In order to carry out an evaluation the methodology was explained to the two groups of students mentioned in Sec. 1, right after finishing the first preliminary test. A first example of the derivation of the factorial function, as well as a full description of the methodology, was introduced as described in sections 2 and 2.1, respectively. Subsequently, the usage of the methodology was explained with the following functions: (1) sum of the first \( n \) positive integers; (2) multiplication by repeated addition; (3) does an integer contain an odd digit? (Sec. 2.2.2); and, (4) sum of the elements in a vector of real numbers. The eight steps involved in the methodology were examined carefully in each example. Nevertheless, the pace was quick, and the material was covered in only one hour (in both groups). Lecture notes describing the methodology and the four previous examples were handed out at the beginning of the class.

At the end of the lecture students were given the following set of exercises to do as homework: (1) binomial coefficient; (2) \( m \cdot (m + 1) \cdot \cdots \cdot (n - 1) \cdot n \), with \( m \leq n \); (3) are all the integers in a vector odd?; and, (4) invert the digits of a positive integer (for example, given 32451, the function returns 15423). Students were given two weeks to hand in the exercises.

Since the methodology was not part of any syllabus, participation in the experiment was therefore voluntary. However, students were rewarded with up to 0.5 extra points on each course (the courses’ grades are between 0 and 10), depending on their performance on the exercises, and especially on a final exam. 18 out of the 25 CS2 students and 23 out of the 28 CS3 students handed in the exercises and participated in the final test. Additionally, 6 CS3 students who had not attended the lecture on the methodology, but were given a copy of the lecture notes, also completed the exercises and took the final test.

The final exam was scheduled for one week after the deadline for handing in the homework. It contained a question-
naire and a set of exercises on tail recursive functions. In particular, the exercises were: (1) power (Sec. 2.2.1); (2) sum of the elements of a vector, from index m to n, with m ≤ n; (3) sum of digits of an integer, without using the methodology; and (4), sum of digits of an integer, where using the methodology was mandatory. Students had the freedom to choose any technique to solve the first two exercises. Additionally, they were asked to solve them by using the methodology, if they had time at the end of the exam, which lasted one hour and a half. Finally, students provided a measure of the confidence on the correctness of their solutions for each exercise, from a scale of 1 to 4.

The questionnaire was related to the methodology, recursion, and formal methods. Some of the most relevant questions were:

a) Does it remind you of other techniques? Which ones?
b) Indicate the difficulty of applying the methodology
c) Do you think it helps learning recursion concepts?
d) Indicate its relative difficulty in comparison with other techniques
e) Would you use it to implement simple functions?
f) Would you rather think about accumulator variables?
g) Does it help understand the difference between imperative and declarative programming better?
h) Were you aware of the relationship between iteration and tail recursion?

Finally, students were encouraged to respond honestly to the questionnaire, as it would not affect the grade.

Table 2: Comparison of results related to the power function in the first and final exams, for students who chose to use the methodology.

<table>
<thead>
<tr>
<th></th>
<th>CS2</th>
<th>CS3</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>first test</td>
<td>final exam</td>
</tr>
<tr>
<td></td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td></td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>7</td>
<td>4</td>
</tr>
</tbody>
</table>

4. RESULTS

This section presents several results associated with the exam’s exercises, as well as the major conclusions obtained from the questionnaire. It is important to note that, before introducing the methodology, the CS2 students had studied recursion and formal methods for program derivation earlier in the semester (the CS3 students, had already covered such material).

Regarding the first two exercises, where students could choose the method they preferred for designing the algorithms, a majority used the methodology. In particular 16 of 18, and 15 out of 23 in the CS2 and CS3 classes, respectively. Furthermore, as a second option at the end of the exam, one CS2 student and five CS3 students implemented correct functions by using the methodology.

It is interesting to compare the results on the tail recursive power function that was included in the first and final exams. After having studied the methodology, the performance improved significantly, especially in the CS2 group. Besides from having less experience, this may be due to the fact that the derivation of tail recursive functions (with formal methods) was a topic included in the syllabus of a programming course they were taking. Thus, their motivation to learn tail recursion was higher (their performance on the homework exercises was better than the CS3 students’). Table 2 summarizes the comparison. The grading of the exercises in the exam was done as described in Sec.1, correct (√), partially correct (+) when the methodology was applied correctly but a minor error was detected, and incorrect (×). Each cell represents the number of students (who chose the methodology to implement the function) that obtained a particular pair of grades in the first and final exams (for example, seven of those CS2 students had an incorrect answer in the preliminary test, but a correct one in the final exam). Only two CS3 students performed worse (√ in the first test, + in the final exam), one of them had not solved the homework problems properly, while another had a minor mistake on the base case.

Regarding the 6 students who did not attend the lecture on the methodology, all of them applied it well on the final exam, although only one had solved all of the homework exercises properly. This indicates that the method is simple, and can be understood without an explicit explanation in class. Nevertheless, only two of these students chose to implement the first two exercises by using the methodology.

Finally, a comparison of the third and fourth exercise, as well as an analysis of the confidence scores did not show differences between using the methodology or not.

In relation to the items associated to the questionnaire (in which 47 students participated in) we found the following:

a) The methodology did not remind 31 students (66%) of other techniques. Although the proposed methodology is closely related to parameter immersion (which the CS2 students had seen previously in the semester, and where generalization is a key concept in order to derive recursive algorithms), surprisingly, only four students explicitly mentioned such formal method.

b) No student found the methodology to be very hard, two thought it was hard, 34 easy, and 11 very easy. This is in accordance with the results, where most students solved the exercises correctly. Additionally, several students described the methodology as fast and methodic or mechanical, as it involves a sequence of well-defined steps. According to the students, the most difficult steps are: obtaining the base cases, finding the generalization, and solving for the extra parameter. We believe they also have difficulties expressing the formulas in step 1.

c) 86% of the students thought it helped them learn recursion concepts. However, some students pointed out that they felt they already needed to know recursion to be able to apply the method. This is partly true, since it is important to recognize, for example, base cases.

d) Furthermore, in comparison with other techniques, no one believed it was much harder, five students thought it was harder, 27 easier, and 12 a lot easier. Clearly, students generally find it simpler (than formal methods). This motivates its coverage in a CS2 course (e.g., as an introduction to formal techniques).

e) 29 students (62%) mentioned they would use it to implement simple functions. However, some students indicated that the algorithms were too basic, and could be derived almost intuitively, without the need of a method.

f) Despite the previous result, 29 students preferred to
think about accumulator variables. This is perhaps not surprising, since students are generally more comfortable using iteration than recursion [3].

g) We believe the methodology provides insight into the difference between imperative and declarative programming. After using the methodology 8 understood this distinction a lot better, 9 better, 23 a bit better, while only 7 did not see an improvement.

h) Furthermore, only 8 out of the 18 CS2 students were aware of the relationship between iteration and tail recursion before seeing the methodology, in comparison to the 25 out of 29 CS3 who knew about the connection. This supports covering the methodology in a CS2 course.

5. DISCUSSION

This paper has described a methodology for deriving tail recursive algorithms that avoids iterative thinking. Its usage may strengthen students’ abilities to express mathematical or logical formulae, comprehend the concept of generalization, learn to recognize base cases, discover program derivation techniques, obtain a better understanding of the declarative paradigm, and use recursion in general.

It can be used in CS1 programming courses to explain tail recursion without relying on iterative or imperative programming concepts. However, we recommend introducing it in a CS2 course, since its application requires expressing functions with mathematical or logical formulae, understanding the concept of function generalization, or learning to recognize the size of a problem and its base cases in order to simplify it. Therefore, it can be covered while teaching recursion, functional programming, or as an introduction to formal methods, since it can be considered as a formal program derivation technique.

The design of recursive algorithms by applying generalization is not new, and plays an important role in the study of formal methods. However, the methodology focuses only on the function (postcondition), without relying on Hoare logic, which students have a lot of difficulty with (according to classroom experience). In other words, the proposed approach eliminates the added complexity associated to formal methods (use and notation), but allows students to implement correct functions with relative ease.

We have employed the proposed technique to implement almost every function we traditionally cover in a CS1 course (factorial, power, change of base, sum of the number of digits, binomial coefficients, Fibonacci numbers, etc.). Some of these functions allow different generalizations that originate distinct and unique algorithms. For example, the paper has presented a version of the factorial function based on nested recursion (note that the resulting algorithm does not have a linear recursion tree), which can be covered in class instead of the classical Ackermann, or McCarthy91 functions.

All of the tail recursive functions included in the tests and exercises can be derived by using one iteration of the methodology (i.e., the general functions only need one extra parameter). Due to limitations on time we could not explain the methodology for more complex functions (e.g., change of base, which needs two extra parameters), as well as other interesting details (choosing the appropriate formula, notation or generalization). These extra examples could be covered in two additional lecture hours, to show the methodology’s full potential, and provide an additional motivation to use it in practice (thus, avoiding the imperative approach and accumulator variables).

An important aspect of the methodology is that it is based on recursive reasoning. Consider the factorial function \( g(x) = x! \). In order to obtain a recursive linear version we must: (1) think of a simpler problem \( g(x − 1) = (x − 1)! \), and (2) find an appropriate operation (multiplication by \( x \)) to transform it back to \( x! \). By using a similar diagram, the reasoning process for the non-tail recursive algorithm can be illustrated as:

\[
g(x) \quad \longrightarrow \quad x! \\
g(x−1) \quad \longrightarrow \quad (x−1)!
\]

Furthermore, note that function \( f\_tail(x,y) \) in Fig. 1(a) calculates the more general \( f(x,y) = y\cdot x! \). The methodology arises in an effort to program such function, by thinking in a similar recursive manner. Note that we must: (1) simplify the problem size towards the base case \((x \rightarrow x−1)\) obtaining \( f(x−1,z) \), and (2) figure out what value \( z \) should take in order to obtain \( f(x−1,z) = f(x,y) \).

We believe the methodology provides an enriching point of view of recursion (and its relationship to iteration), which may reinforce student’s programming skills in general.

6. ACKNOWLEDGEMENTS

Research partially supported by the Spanish MCI project: TIN2008-04103/TSI.

7. REFERENCES