Tail Recursion by Using Function Generalization

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

ABSTRACT
The design of tail recursive algorithms may require thinking about iteration rather than recursion. This paper provides a methodology for deriving tail recursive functions that is based on declarative programming and the concept of function generalization, which allow to avoid iterative thinking.

Categories and Subject Descriptors
D.1.1 [Programming techniques]: Applicative (Functional) Programming—Recursion; K.3.2 [Computers and Education]: Computer and Information Science Education—Computer Science Education

General Terms
Algorithms

Keywords
Design of Algorithms, Generalization, Recursion, Tail Recursion, Nested Recursion

1. BACKGROUND
The design of many tail recursive algorithms doesn’t rely on key concepts inherent in recursion, such as induction, problem decomposition, or functional abstraction [1]. Alternatively, it may involve thinking about the status of variables and parameters (and how it changes with the execution flow), due to its strong relationship with iteration and imperative programming.

This paper presents a methodology for deriving tail recursive functions that is based exclusively on declarative programming and the concept of function generalization. Thus, it can be used in CS1/2 programming courses to explain tail recursion without relying on iterative or imperative programming concepts. Furthermore, the approach can be covered in introductions to programming methodology, since it constitutes a program derivation technique.

2. PROPOSAL
Consider a function \( g(X) \), where \( X \) is a set of parameters, which we would like to implement using a tail recursive function \( f \). Since \( f \) must return a value in its last function call, it will need to perform certain operations and store their results in a set of additional parameters \( Y \). Thus, \( f \) is a generalization of \( g \), where \( g(X) = f(X,Y) \) for a particular set \( Y \). Since \( f \) is tail recursive, and must be defined in terms of itself with no other operations involved, we have \( f(X,Y) = f(X,Z) \), where \( X \) is a simplification of the original parameters \( X \) towards the base case. If we choose an appropriate general function \( f \), solving for \( Z \) can be straightforward, completing the tail recursive definition of \( f \).

Consider the factorial function \( g(x) = x! \). In order to use tail recursion, a new, more general tail recursive function \( f(x,y) \) must be defined. For example, we could choose \( f(x,y) = y \cdot x! \), where \( g(x) = x! \) for \( y = 1 \). Furthermore, since the base case for the factorial function occurs when \( x = 0 \), a natural simplification for \( x \) could be \( x - 1 \). In this scenario we have:

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

for some \( z \). Finally, it is easy to see from the right hand side that \( z = y \cdot x \). Thus, the resulting tail recursive function is:

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

where \( x! = g(x) = f(x, 1) \). Furthermore, other alternative general functions for \( f \) provide completely different algorithms. For instance, if \( f(x, y) = y + x! \), the resulting algorithm makes use of nested recursion.

We have employed this methodology to derive all of the functions that we traditionally cover in a CS1 course (factorial, sum of first positive integers, power, multiplication by repeated addition, change of base, sum of digits, Fibonacci numbers, etc.). We are currently conducting experiments to study the effectiveness of this technique.

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

4. REFERENCES