Yet another transformation scheme for double recursion

José L. Balcázar
Departament L.S.I.
Univ. Politècnica de Catalunya
Edif. U, Pau Gargallo 5, 08028 Barcelona, Spain
balqui@lsi.upc.es

0. Abstract
A tail recursive program (with a single recursive call per case) is derived from a generic recursive program with two independent recursive calls, under no algebraic hypothesis whatsoever. The iterative version, fully verified, is immediate. The derivation is generalized to cover also nested and multiple recursion.

1. Presentation
A rich set of program transformations exist, as a result of a very active area of research since about two decades now. The usefulness of this knowledge is beyond doubt for many researchers. Two main applications of this study are: 1/ the possibility to implement software that automatically applies such semantics-preserving transformations, and 2/ the use of these transformations by programmers to actually calculate their programs, calculating simultaneously the corresponding correctness proofs (as advocated, among many other textbooks, by [6] and [8], to name but two recent ones). We consider the present work oriented towards this second line of applications.

The specific problem we tackle here is as follows: given is a recursive program including two independent recursive calls; wanted is the derivation of a semantically equivalent iterative program, together with the corresponding invariants and bound functions. No algebraic hypothesis is available on the operations used by the programs.

Various solutions exist for the case of operations with certain good algebraic properties and/or programmed in specific ways; for instance, see the solution to some tree traversals (where sequence concatenation is associative) and other interesting problems in [6] and in [10]. A large number of program transformations is given in [2] and [9] (see also the survey [7]).

Not many solutions apply to the case in which no algebraic properties are available. And yet, it is easy to come up with simple applications in which such a transformation is necessary. As a somewhat artificial but simple example, consider testing for the balance
(e.g. AVL-like conditions) of a binary tree. In the recursive case, balance of both subtrees must be checked, and their heights compared. For the sake of efficiency, tupling should be made so that each recursive call returns simultaneously the balance of the received tree and its height. The derivation of the program is straightforward and omitted; it contains two recursive calls and simple arithmetic and boolean operations. However, its iterative version is by no means immediate, since the complex operation to be made on the results of the recursive call does not fit most of the available transformation rules.

Two known solutions apply even in the absence of nice algebraic properties. Tabulation may work [4], but raises a number of difficulties with certain abstract data types; it requires the program to move from “small” to “large” data (in the ordering proving termination of the recursion, like for a dynamic programming scheme) and ability to store the solution found for each subproblem. In the case of graphs, for instance, it is preferable to choose (and use that fact in the calculation) an adjacency matrix implementation [4]. Additionally, it may lead to unnecessarily high memory space consumption.

A much better solution for our general case is given in [5]. It is very general, in that the transformation does not request by itself a stack; this allows to apply it to algorithms that twist pointers in smart ways to keep memory space to a minimum (such as the Schorr-Waite graph-marking algorithms). This solution is based on mutually exclusive assertions that must distinguish among the different return points whenever a recursive call ends; furthermore, these assertions must be evaluable (efficiently enough) in run-time. In the absence of other information, a stack may be introduced to implement efficient tests corresponding to these assertions; but that solution caters for other possibilities.

The iterative version of the resulting program, however, is somewhat complex. It is based on two mutually (tail-)recursive subprograms that yield two nested loops, and furthermore the initially called one is not the one detecting termination; so an extra initialization loop may be required. We propose here a different solution. The disadvantage is that our proposal is not general enough, in its current form, to avoid the use of the stack (and indeed the stack is present in the transformation from the very beginning). On the other hand, we consider that the structure of the single tail-recursive program obtained is simpler and easier to understand. Additionally, as a by-product of the transformation, we obtain the invariant that proves correct the corresponding loop.

Our solution is based on two operations, equationally defined, that allow for the tail-recursive program to be derived using simple algebraic manipulations such as unfolding and folding. The iterative version, annotated with invariants, is immediate from the tail-
recursive one, by completely standard transformations.†

2. Formalization

The problem is formalized as follows: find a tail-recursive definition allowing one to compute a function \( f \) defined by the equations

\[
\begin{align*}
  f(x) &= h(x) \text{ if } S(x) \quad (1) \\
  f(x) &= c(x, f(g_1(x)), f(g_2(x))) \text{ if } \neg S(x) \quad (2)
\end{align*}
\]

where the typing is such that the equations make sense. Essentially, \( x \) ranges on some type \( T_1 \), which is the domain of \( f \), \( h \), \( S \), \( g_1 \), and \( g_2 \), and also the codomain of \( g_1 \) and \( g_2 \). \( S \) is boolean, and the codomain of \( f \), \( h \), and \( c \) is some possibly different type \( T_2 \). Of course, the nonrecursive solution \( h \) must be independent of \( f \). We will use letters \( x, y, z \) to range over \( T_1 \) and \( s, t \) over \( T_2 \).

This definition of \( f \) is assumed to be proved valid by a bound (or variant) function \( V \), valued on the natural numbers, such that \( V(g_1(x)) < V(x) \) and \( V(g_2(x)) < V(x) \) for all \( x \).

Since \( f \) may not admit at all a tail-recursive formulation itself, the solution of the problem is expected to be an embedding of \( f \). Nothing else is assumed. In particular, the correctness of the transformation must depend on no algebraic property of the operations involved. Thus, provided that the above minimal considerations are satisfied, all the operations are arbitrary, as are the types \( T_1 \) and \( T_2 \); in particular, these can be tuples. Many recursive programs with two recursive calls can be casted into this form, possibly modulo an embedding of some parameters to take care of operations made before the second recursive call. For instance, the test for perfect balance of a binary tree, with the tupling indicated in the presentation, can be expressed as

\[
\begin{align*}
  h\text{-}b(t) &= \langle 0, T \rangle \text{ if } \text{null}(t) \\
  h\text{-}b(t) &= c(h\text{-}b(\text{left}(t)), h\text{-}b(\text{right}(t))) \text{ if } \neg \text{null}(t)
\end{align*}
\]

where \( c(\langle n_0, b_0 \rangle, \langle n_1, b_1 \rangle) = \langle \max(n_0, n_1) + 1, b_0 \land b_1 \land (n_0 = n_1) \rangle \), \( T \) is the boolean constant “true”, and \text{left} and \text{right} the corresponding subtree operations. The height itself serves as a bound function.

† A simplified version of the solution to be presented here is also described, in Spanish and in a manner suitable to undergraduates, in the author’s recent textbook [1]. The author’s usual line of research being different from the present one, it may well be, however, that other published solutions, other than the mentioned ones, are unknown to him.
3. Our solution

The solution we propose is based on deriving a tail-recursive program for an embedding of \( g f \) of \( f \), through the use of the auxiliary operation \( fp \), both defined by the following equations:

\[
\begin{align*}
fp(s, []) &= s & (3) \\
fp(s, (x, t, F) : p) &= fp(f(g_2(x)), (x, s, T) : p) & (4) \\
fp(s, (x, t, T) : p) &= fp(c(x, t, s), p) & (5) \\
gf(x, s, F, p) &= fp(f(x), p) & (6) \\
gf(x, s, T, p) &= fp(s, p) & (7)
\end{align*}
\]

Here, \( T \) and \( F \) are the boolean constants “true” and “false”. The list parameter \( p \), containing 3-tuples, will work, as the calculation shows, as a pushdown stack (not surprisingly). In equation 4, the value \( t \) does not appear at the right hand side, and similarly for \( s \) and \( x \) in equations 6 and 7. We will assume that \( x_0 \) and \( s_0 \) are arbitrary values in \( T_1 \) and \( T_2 \), respectively, which we will use when applying these equations from right to left.

Now \( f(z) \) can be easily computed from \( gf \) and \( fp \) as follows: for \( s \) arbitrary, in particular for \( s_0 \),

\[
\begin{align*}
gf(x, s_0, F, []) &
\equiv 
(f \text{ equation } 6)
fp(f(x), []) \\
\equiv 
(f \text{ equation } 3)
f(x)
\end{align*}
\]

Unfolding the functions \( fp \) and \( f \) in the equations of \( gf \), a number of cases arises. Let us enumerate them:

\[
\begin{align*}
a/ \ & gf(x, s, F, p) \text{ with } \neg S(x) \\
b/ \ & gf(x, s, F, p) \text{ with } S(x) \\
c/ \ & gf(x, s, T, []) \\
d/ \ & gf(x, s, T, (y, t, F) : p)) \\
e/ \ & gf(x, s, T, (y, t, T) : p)) 
\end{align*}
\]

Once these cases are studied, it is easily shown by induction that the \( T_1 \)-type values in the 3-tuples of the list (here denoted \( y \)) fulfill always \( \neg S(y) \). Now the derivation in each case is straightforward. To give the reader an idea of how it proceeds, we complete here the case a/, the only one with some interest. Assuming \( \neg S(x) \),

\[
\begin{align*}
gf(x, s, F, p) 
\equiv 
(f \text{ equation } 6)
fp(f(x), p) 
\equiv 
(f \text{ equation } 2, \neg S(x)) \\
fp(c(x, f(g_1(x)), f(g_2(x))), p) 
\equiv 
(f \text{ equation } 5)
\end{align*}
\]
\[
fp(f(x), \langle x, f(0) \rangle, T) = \begin{cases} 
0 & \text{equation 4} \\
f(x, s_0, F) & \text{equation 6}
\end{cases}
\]

gf(0, s_0, F, \langle x, s_0, F \rangle) = \text{similar calculations yield the following results:}

\[
gf(x, s, F, p) = \begin{cases} 
gf(0, s_0, F, \langle x, s_0, F \rangle) & \text{if } S(x) \text{ (as above)} \\
gf(x, s, F, h(x), T, p) & \text{if } S(x) \text{ (use eq. 6, 1, 7)} \\
gf(x, s, T, \langle x, s \rangle, F, p) & \text{use eq. 6, 1, 7}
\end{cases}
\]

A tail-recursive definition of \(gf\), an embedding of \(f\), has been obtained. It can be seen, inductively, that the first parameter is always \(x_0\) whenever the third one is \(T\), whereas the second one is always \(s_0\) whenever the third one is \(F\). These simple observations are useful for checking that the bound function given in the next section indeed strictly decreases with each call.

The tail-recursive program and the iterative program (with invariant \(gf(x, s, b, p) = f(x)\) for the precondition \(x = X\)) are immediate, by following known techniques from [3]; for instance, the loop is:

\[
\begin{cases}
\{\text{Pre} : x = X\} \\
\text{var } y : T_1; s, t : T_2; b, bb : \text{bool}; p : \text{stack} ;
\{f(x) = f(X)\} \\
\{s, b, p \} := \langle s_0, F, \text{empty_stack} \rangle ;
\{\text{Inv} : gf(x, s, b, p) = f(x)\}
\end{cases}
\]

\text{do } -b \land \text{empty}(p) \rightarrow
\text{if } S(x) \land -b \rightarrow \langle x, s, b \rangle := \langle x, h(x), T \rangle
\land S(x) \land -b \rightarrow \langle x, p \rangle := \langle g_1(x), \text{push}(\langle x, s_0, F \rangle, p) \rangle
\text{do } -b \rightarrow \langle y, t, bb \rangle := \text{top}(p); p := \text{pop}(p);
\text{if } -bb \rightarrow
p := \text{push}(\langle y, s, T \rangle, p);
\langle x, s, b \rangle := \langle g_2(y), s_0, F \rangle
\text{fi}
bb \rightarrow s := c(y, t, s)
\text{fi}
\text{od:}
\{gf(x, s, T, \langle \_ \rangle) = f(X)\}
\text{return } s
To check partial correctness amounts to repeating the calculations leading to the tail-
recursive solution. Total correctness is proved in the next section.

4. **Bound function**

We discuss now the bound function that ensures termination of the obtained program. We assume ordering $F < T$ on $\text{bool}$, and later a function $n$ consistent with the ordering, defined by $n(F) = 0$ and $n(T) = 1$. We start by defining a (possibly coarser) bound on the original function $f$, valuated on the natural numbers, by means of an induction validated by the original bound function $V$:

\[
V'(x) = 0 \text{ if } S(x) \\
V'(x) = 1 + 3 \cdot V'(g_1(x)) + 3 \cdot V'(g_2(x)) \text{ if } \neg S(x)
\]

We also assume that the value $x_0$ arbitrarily selected in the previous section is such that $V'(x_0) = 0$ (otherwise, one can adjust $V'$ by redefining this special case, if necessary). This bound is now extended to the pushdown list by:

\[
W([ I ] ) = 0 \\
W( (x, s, F) : p) = 2 \cdot V'(x) + W(p) \\
W( (x, s, T) : p) = V'(x) + W(p)
\]

We have to account also for the boolean value, which at crucial times switches from $F$ to $T$. On parameters $x, s, b$ (boolean), and $p$, the following pair always strictly decreases (lexicographically):

\[\langle 3 \cdot V'(x) + W(p), \neg b \rangle\]

We can therefore define a bound function (valued in the natural numbers) as

\[2 \cdot (3 \cdot V'(x) + W(p)) + n(\neg b)\]

It is a routine chore to check that indeed each recursive call in the tail-recursive scheme obtained in the previous section decreases this bound function; therefore termination is guaranteed.

5. **A simplification**

In many cases a simplification is possible (although some readers might label it as a com-
plication). It is frequent that the type $T_2$ contains some specific value that is never the result of the program; sometimes, even many such values. For instance, in the example of the perfect balance of a binary tree, the output $\langle 0, F \rangle$ never arises, since the only tree of height zero, the empty tree, is actually well-balanced. Assuming the presence of such a value, together with a boolean-valued function that identifies it, we employ it to implement (or: hide) the boolean parameter of our embedding.
Let us denote that value by \( \bot \), and assume that we can test for equality to it, and that neither \( h(x) = \bot \) nor \( c(x, f(g_1(x)), f(g_2(x))) = \bot \) ever happen. Now all the pairs of a \( T_2 \) plus a boolean can get rid of the boolean; indeed, it is easy to see by inspection that, both as an argument to \( g f \) or as part of the 3-tuples in the stack, each boolean value indicates exactly whether the \( T_2 \) value preceding it is worth remembering. Using \( \bot \) in its place whenever the boolean value is false, the equations become

\[
\begin{align*}
fp(s, \bot) &= s \\
fp(s, (x, \bot) : p) &= fp(f(g_2(x)), (x, s) : p) \\
fp(s, (x, t) : p) &= fp(c(x, t, s), (x, s) : p) \text{ if } t \neq \bot \\
gf(x, \bot, p) &= fp(f(x), (x, s) : p) \text{ if } s \neq \bot \\
gf(x, s, p) &= fp(s, p) \text{ if } s \neq \bot \\
\end{align*}
\]

From these equations, appropriate calculations similar of the previous ones yield the tail-recursive definition

\[
\begin{align*}
gf(x, \bot, p) &= gf(g_1(x), \bot, (x, \bot) : p) \text{ if } \neg S(x) \\
gf(x, \bot, p) &= gf(x_0, h(x), p) \text{ if } S(x) \\
gf(x, s, \bot) &= s \text{ if } s \neq \bot \\
gf(x, s, (y, \bot) : p) &= gf(g_2(y), \bot, (y, s) : p) \text{ if } s \neq \bot \\
gf(x, s, (y, t) : p) &= gf(x_0, c(y, t, s), p) \text{ if } s \neq \bot \text{ and } t \neq \bot \\
\end{align*}
\]

The bound function can be obtained also from that of the general case by adjusting the participation of the now missing boolean component. Actually, since this boolean value is now represented by the inequality \( s \neq \bot \), we simply use it wherever the boolean was used. The auxiliary bound function on the stack is now

\[
\begin{align*}
W(\bot) &= 0 \\
W((x, \bot) : p) &= 2 \ast V'(x) + W(p) \\
W((x, s) : p) &= V'(x) + W(p) \text{ if } s \neq \bot \\
\end{align*}
\]

Thus, on parameters \( x \), \( s \), and \( p \), the bound function is

\[2 \ast (3 \ast V'(x) + W(p)) + n(s = \bot)\]

6. **Nested recursion**

The case of nested recursion can be easily treated by a very similar transformation: it suffices to consistently provide \( g_2 \) throughout with the result of the first recursive call, \( f(g_1(x)) \). Only minor changes have to be made. The initial specification becomes

\[
\begin{align*}
f(x) &= h(x) \text{ if } S(x) \\
f(x) &= c(x, f(g_1(x)), f(g_2(x, f(g_1(x))))) \text{ if } \neg S(x) \\
\end{align*}
\]

(\( 1'' \))

(\( 2'' \))

(where in most particular applications the second parameter to \( c \) would be ignored). The second equation of \( fp \) has to be modified accordingly:

\[\]
The other equations remain as before. Then it is routine to check that all the calculations can be carried over, up to the corresponding tail-recursive program. The only change is that the parameter \( s \) shows up as a parameter of \( g_2 \) in the corresponding case (equation 11" below; the other equations are exactly as before):

\[
gf(x, s, F, p) = gf(g_1(x), s_0, F, (x, s_0, F) : p) \text{ if } \neg S(x) \tag{8}
\]

\[
gf(x, s, F, p) = gf(x_0, h(x), T, p) \text{ if } S(x) \tag{9}
\]

\[
gf(x, s, T, [\ ] = s \tag{10}
\]

\[
gf(x, s, T, \langle y, t, F \rangle : p) = gf(g_2(y, s), s_0, F, \langle y, s, T \rangle : p) \tag{11''}
\]

\[
gf(x, s, T, \langle y, t, T \rangle : p) = gf(x_0, c(y, t, s), T, p) \tag{12}
\]

Notice that it is not clear anymore that the hypothesis that \( V \) witnesses termination of \( f \) via ordinal \( \omega \) (i.e., the natural numbers) is reasonable: bound functions for prominent examples of nested recursion (e.g., the Ackermann function) correspond to higher ordinals. However, such programs do not arise in practice due to the unmanageable growth of their running time (which is not bounded by any primitive recursive function).

7. **Multiple recursion**

Generalization to multiple recursion is also possible; the details become somewhat more complex. We assume that each element \( x \) of type \( T_1 \), unless \( S(x) \) holds, has a number \( M(x) \geq 1 \) of successors, \( sc(x, i) \) for \( 1 \leq i \leq M(x) \). The problem function \( f \) is now

\[
f(x) = h_1(x) \text{ if } S(x)
\]

\[
f(x) = c_1(x, f'(x, M(x))) \text{ if } \neg S(x)
\]

where \( f' \) is defined in terms of the traversal of the sequence of successors:

\[
f'(x, 1) = f(sc(x, 1))
\]

\[
f'(x, i) = c_2(f(sc(x, i)), f'(x, i - 1)) \text{ if } 1 < i \leq M(x)
\]

The generalizations are similar to the ones for the binary case. The “master” function \( gf \) is defined in the same way over the function managing the stack; but now the triples in the stack contain an integer instead of a boolean, to mark the already solved part.

\[
fp(s, [\ ]) = s
\]

\[
fp(s, \langle x, t, 1 \rangle : p) = fp(c_1(x, s), p) \text{ if } 1 = M(x)
\]

\[
fp(s, \langle x, t, M(x) \rangle : p) = fp(c_1(x, c_2(s, t)), p) \text{ if } 1 < M(x)
\]

\[
fp(s, \langle x, t, 1 \rangle : p) = fp(f(sc(x, 2)), \langle x, s, 2 \rangle : p) \text{ if } 1 < M(x)
\]

\[
fp(s, \langle x, t, i \rangle : p) = fp(f(sc(x, i + 1)), \langle x, c_2(s, t), i + 1 \rangle : p) \text{ if } 1 < i < M(x)
\]

\[
gf(x, s, F, p) = fp(f(x), p)
\]

\[
gf(x, s, T, p) = fp(s, p)
\]
The relationship between $gf$ and $f$ is as before: $gf(x, s_0, F, [\ ] = f(x)$. The tail recursion ends when the stack is empty and the boolean true. The case when $S(x)$ holds, as well as the case when the boolean parameter is true and the stack is not empty, are all quite immediate, simply by unfolding the definitions (and distinguishing the necessary subcases). When $\neg S(x)$ and the boolean parameter is false, the equation to be obtained is

$$gf(x, s, F, p) = gf(sc(x, 1), s_0, F, \langle x, s_0, 1 \rangle : p)$$

which is immediate when $M(x) = 1$ but requires somewhat harder work if $M(x) > 1$. It rests of the following fact: for all $i$ with $1 < i \leq M(x)$, and for all $t$,

$$fp(f(sc(x, i)), \langle x, f'(x, i - 1), i \rangle : p) = fp(f(sc(x, 1)), \langle x, t, 1 \rangle : p)$$

This is easily proved by induction on $i$, and has the following consequence, which is the crucial fact necessary to complete the derivation of $gf(x, s, F, p)$:

$$fp(f(sc(x, M(x))), \langle x, f'(x, M(x) - 1), M(x) \rangle : p) = fp(f(sc(x, 1)), \langle x, t, 1 \rangle : p)$$

The following tail-recursive equations for $gf$ are obtained in this way:

$$gf(x, s, F, p) = gf(sc(x, 1), s_0, F, \langle x, s_0, 1 \rangle : p) \text{ if } \neg S(x)$$
$$gf(x, s, F, p) = gf(x_0, h(x), T, p) \text{ if } S(x)$$
$$gf(x, s, T, [\ ] = s$$
$$gf(x, s, T, \langle y, t, 1 \rangle : p) = gf(x_0, c_1(y, s), T, p) \text{ if } 1 = M(y)$$
$$gf(x, s, T, \langle y, t, 1 \rangle : p) = gf(sc(y, 2), s_0, F, \langle y, s, 2 \rangle : p) \text{ if } 1 < M(y)$$
$$gf(x, s, T, \langle y, t, M(x) \rangle : p) = gf(x_0, c_1(y, c_2(s, t)), T, p)$$
$$gf(x, s, T, \langle y, t, i \rangle : p) = gf(sc(y, i + 1), s_0, F, \langle y, c_2(s, t), i + 1 \rangle : p)$$

if $1 < i < M(x)$

The bound function is a straightforward generalization of the one used for the binary case. Here we must assume first that we weight each $x$ heavy enough, well above its successors:

$$V''(x) = 0 \text{ if } S(x)$$
$$V''(x) = 1 + \sum i : 1 \leq i \leq M(x) : M(sc(x, i)) * V''(sc(x, i)) \text{ if } \neg S(x)$$

with again $V''(x_0) = 0$; we extend it to the stack by:

$$W(\ [\ ] = 0$$
$$W(\langle x, s, i \rangle : p) = (M(x) - i + 1) * V''(x) + W(p)$$

and finally define the bound function as

$$2 * ((M(x) + 1) * V''(x) + W(p)) + n(\neg b)$$

We omit here the straightforward check that it is strictly decreased by each of the tail-recursive calls.

8. Discussion

We have described a transformation that allows us to derive a tail-recursive program from a doubly recursive one, without any additional assumption. An iterative solution, annotated
with the corresponding invariant and bound function as provided by the tail-recursive program, is immediate. A brief comparison to some other published solutions has been made. A very similar solution, just slightly more general, works for the case of nested recursion. A solution for multiple recursion has been proposed as well.

An alternative statement of the problem for multiple recursion would stem from viewing a recursive call within a loop as two calls to formally different (but essentially identical) programs, much in the same way as the first-child/right-sibling binary tree representation of a multiway tree; the initial equations would be

\[
\begin{align*}
  f(x) &= h_1(x) \text{ if } S(x) \\
  f(x) &= c_1(x, f(g_1(x)), f'(x, g_2(x))) \text{ if } \neg S(x)
\end{align*}
\]

where \(g_2(x)\) is now a list of \(T_1\) values, to which \(f\) is extended by equations

\[
\begin{align*}
  f'(x, []) &= h_2(x) \\
  f'(x, y : r) &= c_2(x, f(y), f'(r))
\end{align*}
\]

However, we have not found the right generalization \(f_p\) to apply in this setting the same “master” function \(gf\), although we believe that it is possible. The problem lies in that the list of successors has to be traversed “in the reverse order”, and it is not clear at all in what place a list reversal operator could be applied to solve the problem. On the other hand, even a solution avoiding the reversal might exist in general. (Of course, algebraic assumptions on \(c_2\), such as some sort of associativity, would probably allow for solving this problem by application of well-known techniques. But this is what we want to avoid here.)

The simplification presented section 5 appears in [1], a textbook in Spanish, although there the program is casted in terms of trees, and the bound function is not discussed; this reference includes also all the calculations omitted in section 3 and some intuitive explanations of the auxiliary functions.

We have admittedly increased the “rabbitcount” by an important amount. We acknowledge that the “magic” equations doing all the work for us have been put forward without the slightest hint of how they were obtained. The author confesses that, even though he would have liked to reach them by purely syntactic natural manipulations, he was unable to. A long trial-and-error exploration, guided by purely operational issues (e.g. how a compiler may handle the double recursion) is behind this solution. Most likely, this solution is still more complex than necessary, and most likely the reason lies in this (partial) operational contamination. We would like to see the rabbitcount decreased again, and expect that a solution so obtained can be simpler than this one; but believe that it is not an easy task.
9. Acknowledgements

The referees of a previous version are entitled to the warmest thanks for their kind and detailed suggestions and (strong but) very constructive criticism. Albert Rubio deserves many thanks for proposing alternative formulations and discussing the bound function; students and (present and past) teachers of Programming Methodology courses in my departament deserve thanks for asking questions, commenting on my ideas, and offering theirs.

10. References