A Transformation Scheme
for Double Recursion

José L. Balcázar

Report LSI-94-8-R
A transformation scheme for double recursion

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

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.

Presentation

A rich set of program transformations exist, as a result of a very active area of research since about two decades now. Some of them, such as unfolding and folding, are fundamental and underlie many others. 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, simultaneously to calculating the corresponding correctness proof (as advocated, among many other textbooks, by [Cohen 90] and [Kaldewaij 90], to name but two recent ones).

We consider the present work oriented towards this second line of applications.

The specific problem we tackle 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 [Partsch, Pepper 86] and in [Cohen 90]. A large number of program transformations is given in [Bauer, Wössner 82] and [Partsch 90] (see also the survey [Feather 87]).

It is easy to come up with simple applications where this transformation applies. 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.

This note presents two operations, equationally defined, that allow for a 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 (omitted here). We consider that the solution presented here is satisfactory, in that all calculations are extremely simple and the outcome is easily verified by means of assertions obtained during the calculation itself.†

Formalization

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

1/ $f(x) = h(x)$ if $S(x)$
2/ $f(x) = c(x, f(g_1(x)), f(g_2(x)))$ if $\neg S(x)$

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

† 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 [Balcázar 93]. The author's usual line of research being different from the present one, it may well be, however, that some other comparably good published solution is unknown to him.
\[ h-b(t) = \langle 0, T \rangle \text{ if } \text{null}(t) \]
\[ h-b(t) = c(h-b(\text{left}(t)), h-b(\text{right}(t))) \text{ if } \neg \text{null}(t) \]

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

Our solution

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

3/ \quad fp(s, [ ] ) = s
4/ \quad fp(s, \langle x, t, F \rangle : p ) = fp(f(g_2(x)), \langle x, s, T \rangle : p )
5/ \quad fp(s, \langle x, t, T \rangle : p ) = fp(c(x, t, s), p )
6/ \quad gf(x, s, F, p ) = fp(f(x), p )
7/ \quad gf(x, s, T , p ) = fp(s, p )

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 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,

\[ sf(x, s, F, [ ]) \]
\[ \equiv \quad ( \text{equation 6/ } ) \]
\[ fp(f(x), [ ]) \]
\[ \equiv \quad ( \text{equation 3/ } ) \]
\[ f(x) \]

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

a/ \quad gf(x, s, F, p ) \text{ with } \neg S(x)

b/ \quad gf(x, s, F, p ) \text{ with } S(x)

c/ \quad gf(x, s, T, [ ]) \]
d/ \quad gf(x, s, T, \langle y, t, F \rangle : p ) )

e/ \quad gf(x, s, T, \langle y, t, T \rangle : p ) )

Once these cases are studied, it is easily shown by induction that the \( T_1 \)-type values in the 3-tuples of the list fulfill always \( \neg S \). 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 \( \neg S(x) \),
\( gf(x, s, F, p) \)
\( \equiv \) (equation 6/)
\( fp(f(x), p) \)
\( \equiv \) (equation 2/, \( \neg S(x) \))
\( fp(c(x, f(g_1(x)), f(g_2(x))), p) \)
\( \equiv \) (equation 5/)
\( fp(f(g_2(x)), \langle x, f(g_1(x)), T \rangle : p) \)
\( \equiv \) (equation 4/)
\( fp(f(g_1(x)), \langle x, s_0, F \rangle : p) \)
\( \equiv \) (equation 6/)
\( gf(g_1(x), s_0, F, \langle x, s_0, F \rangle : p) \)

Similar calculations left to the reader yield the following results:

\( gf(x, s, F, p) = gf(g_1(x), s_0, F, \langle x, s_0, F \rangle : p) \) if \( \neg S(x) \) (as above)
\( gf(x, s, F, p) = gf(x_0, h(x), T, p) \) if \( S(x) \) (use eq. 6/, 1/, 7/)
\( gf(x, s, T, \langle x, t, F \rangle : p) \) (use eq. 7/, 3/)
\( gf(x, s, T, \langle y, t, F \rangle : p) = gf(g_2(y), s_0, F, \langle y, s, T \rangle : p) \) (use eq. 7/, 4/, 5/)
\( gf(x, s, T, \langle y, t, T \rangle : p) = gf(x_0, h(x, t, s), T, p) \) (use eq. 7/, 5/, 7/)

A tail-recursive definition of an embedding of \( f \) has been obtained. The tail-recursive program and the iterative program (with invariant \( gf(x, s, b, p) = f(X) \) for the precondition \( x = X \)) are immediate, and to check their partial correctness amounts to repeating the calculations leading to it. Total correctness is proved in the next section.

**Bound function**

The bound function that ensures termination of the program obtained is nontrivial, in that it is a somewhat complex combination of otherwise very easy functions. We chose to present it as valued in the product space \( \mathbb{N} \times \mathbb{N} \times \mathbb{N} \times \text{bool} \) with a lexicographic ordering: first components are most significant; when equal, second components are considered; and so on. The ordering on \( \text{bool} \) is \( F < T \). We start by defining a (possibly coarser) bound on the original function \( f \), valued on the natural numbers, by means of an induction validated by the original bound function \( V \):

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

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 counting only the outputs of \( g_2 \) on "false" 3-tuples; also, we count the number of such 3-tuples and the total number of tuples in the stack, using appropriately defined functions:
\[ W(\text{true}) = 0 \]
\[ W((x,s,F) : p) = V'(g_2(x)) + W(p) \]
\[ W((x,s,T) : p) = W(p) \]
\[ U(\text{true}) = 0 \]
\[ U((x,s,F) : p) = U(p) + 1 \]
\[ U((x,s,T) : p) = U(p) \]
\[ H(\text{true}) = 0 \]
\[ H((x,s,b) : p) = H(p) + 1 \]

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 function is defined as
\[ (V'(x) + W(p), U(p), H(p), \neg b) \]

It is a routine chore to check that each recursive call in the tail-recursive scheme obtained in the previous section decreases (lexicographically) this bound function. Since this ordering of the product space is well-founded, termination is guaranteed.

A simplification

In many cases a simplification is possible (although some readers might label it as a complication). 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 \( (0,F) \) 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 \( gf \) 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
\[ 3/ \ \text{fp}(s, [\text{true}]) = s \]
\[ 4/ \ \text{fp}(s, (x, \bot) : p) = \text{fp}(f(g_2(x)), (x, s) : p) \]
\[ 5/ \ \text{fp}(s, (x, t) : p) = \text{fp}(c(x,t,s), p) \text{ if } t \neq \bot \]
\[ 6/ \ \text{gf}(x, \bot, p) = \text{fp}(f(x), p) \]
\[ 7/ \ \text{gf}(x, s, p) = \text{fp}(s, p) \text{ if } s \neq \bot \]

From these equations, appropriate calculations similar of the previous ones yield the tail-recursive definition
\( g_f(x, \bot, p) = g_f(g_1(x), \bot, (x, \bot) : p) \text{ if } S(x) \)
\( g_f(x, \bot, p) = g_f(x_0, h(x), p) \text{ if } S(x) \)
\( g_f(x, s, [\ ] = s \text{ if } s \neq \bot \)
\( g_f(x, s, (y, \bot) : p) = g_f(g_2(y), \bot, (y, s) : p) \text{ if } s \neq \bot \)
\( g_f(x, s, (y, t) : p) = g_f(x_0, c(y, t, s), p) \text{ if } s \neq \bot \text{ and } t \neq \bot \)

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 functions on the stack are now
\[
W([\ ] = 0 \\
W((x, \bot) : p) = V'(g_2(x)) + W(p) \\
W((x, s) : p) = W(p) \text{ if } s \neq \bot \\
U([\ ] = 0 \\
U((x, \bot) : p) = U(p) + 1 \\
U((x, s) : p) = U(p) \text{ if } s \neq \bot \\
H([\ ] = 0 \\
H((x, s, b) : p) = H(p) + 1
\]

Thus, on parameters \( x, s, \) and \( p \), the bound function is
\[
(V'(x) + W(p), U(p), H(p), s \neq \bot)
\]
valuated on the same product space as before.

**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. The literature on program transformation is wide enough (and
uses so many diverse notations) that the author is unable to absolutely guarantee that no
other similar solution to this problem existed; only that it was not found.

The simplification presented in the last section appears in [Balcázar 93], 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 and some intuitive
explanations of the auxiliary functions.

Some remarks can be made. In proving termination of loops the case seldom (never,
unless unbounded nondeterminism is allowed) arises that a preorder different from the
natural numbers is necessary. However, in our case we have used a lexicographic ordering
on a product space (actually resorting to a \( 3\omega + 2 \) ordinal). We are sure that this program
can be verified within the ordinal $\omega$, i.e. by means of a bound function with values on the natural numbers, but do not see any reasonably simple way to do it.

Generalization to multiple recursion seems clearly possible; the details must be worked out. A recursive call within a loop should be seen as two recursive calls to a formally different (but essentially identical) program, much in the same way as the first-child/right-sibling binary tree representation of a multiway tree. However, the specification needs major adjustments.

On the other hand, the case of nested recursion can be easily treated in the transformation: it suffices to provide $g_2$ with the result of the first recursive call, $f(g_1(x))$. Only minor changes have to be made: the recursive case of the initial specification becomes $f(x) = c(x, f(g_1(x)), f(g_2(x, f(g_1(x))))));$ and the second equation of $fp$ has to be modified accordingly: $fp(s, (x, t, F : p) = fp(g_2(x, s), (x, s, T) : p)$. 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 one of the cases becomes $gf(x, s, T, (y, t, F) : p) = gf(g_2(y, s), s_0, F, (y, s, T) : p)$.

Notice that it is not clear anymore that the hypothesis that $V$ witnesses termination of $f$ via ordinal $\omega$ 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).

Finally, 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 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.

Acknowledgements

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 department deserve thanks for asking questions, commenting on my ideas, and offering theirs.
References


Recent Research Reports


LSI-93-48-R "Translation equivalence via lexicon: a study on tlinks", Anna Samiotou, Irene Castellón, Francesc Ribas, and German Rigau.

LSI-93-49-R "A class-based approach to learn appropriate selectional restrictions from a parsed corpus", Francesc Ribas.

LSI-93-50-R "Quality control of software specification written in natural language", Núria Castell, Olga Slavkova, Yannick Toussaint, and Antoni Taulles.


LSI-94-1-R "Logspace and logtime leaf languages", Birgit Jenner, Pierre McKenzie, and Denis Thérien.

LSI-94-2-R "Degrees and reducibilities of easy tally sets", Montserrat Hermo.

LSI-94-3-R "Isothetic polyhedra and monotone boolean formulae", Robert Juan-Arinyo.

LSI-94-4-R "Una modelización de la incompletitud en los programas" (written in Spanish), Javier Pérez Campo.


LSI-94-6-R "Construction of the Fuce Octree model", Núria Pla-Garcia.

LSI-94-7-R "On the expected depth of boolean circuits", Josep Díaz, María J. Serna, Paul Spirakis, and Jacobo Torán.

LSI-94-8-R "A transformation scheme for double recursion", José L. Balcázar.

Copies of reports can be ordered from:

Nuria Sánchez  
Departament de Llenguatges i Sistemes Informàtics  
Universitat Politècnica de Catalunya  
Pau Gargallo, 5  
08028 Barcelona, Spain  
secrelsi@lsi.upc.es