Embedding: a Unifying Concept for Recursive Program Design

C. Rosselló
J. L. Balcázar
R. Peña

Report LSI–88–13

Embedding: a Unifying Concept for Recursive Program Design

Celestí Rosselló
José L. Balcázar
Ricardo Peña

Department of Software (Llenguatges i Sistemes Informàtics)
Universitat Politècnica de Catalunya
08028 Barcelona, SPAIN

e-mail
celesti@fib.upc.es (ean)
eabalqui@ebrupc51 (bitnet)
ricardo@fib.upc.es (ean)

Abstract: The concept of embedding one function into another is used to get a unified view of some well-known design techniques, such as loop derivation using invariants, recursion removal, or folding-unfolding. A design method for recursive functions based on embedding is presented and its relation to those techniques stated. Efficiency transformations are also formalized using embedding. A few examples will clarify and complement the proposed technique.

Resum: Es fa servir el concepte d'immersió d'una funció en una altra per donar una visió unificada d'algunes tècniques de disseny ben conegudes com són: derivació de bucles fent servir invariants, eliminació de la recursivitat o plegat i desplegat. Es proposa un mètode de disseny recursiu basat en la immersió i es relaciona amb les tècniques abans esmentades. La immersió també es fa servir per formalitzar les transformacions d'eficiència. Es presenten uns quants exemples per il·lustrar el mètode proposat.

1. Introduction

In the design of algorithms, two major goals are correctness and efficiency; both concepts can be formalized in a scientific, mathematically based way. Each goal requires the development of appropriate tools for the analysis of programs from the respective points of view; in particular, certain formal methods are a useful tool in the design of correct algorithms.

A "correct algorithm" is an algorithm which behaves "as expected". Thus, the formalization of the very definition of correctness requires an appropriate language in which to express the expected behavior of the algorithm (its *specification*), in order to prove that the algorithm satisfies it. Specifications are useful in two additional ways: supposedly correct algorithms have to be transformed (e.g. by translation into machine code), and transformation rules have to be shown to preserve the correctness; furthermore, the specification can be highly useful during the design, since, as expressed in [Gri], "programming is a goal-oriented activity".

In particular, a well-known methodology for the design of loops has been developed, based on the identification of appropriate invariants and the design of loops according to these invariants ([Dij_a], [Gri]). However, the verification of iterative (i.e. loop-based) algorithms is usually more complex than the verification of recursive programs, where a noetherian induction, using as hypothesis the correctness of the recursive call, may require much less verification effort. See [BrK] for a discussion of the loss of structure in the recursive-to-iterative transformations. We also discuss this topic later.

Thus, if a recursive design based on the specification of the program succeeds, the correctness of the obtained program is usually easier to argue than for iterative designs. However, in frequent cases appearing in a programming-in-the-small paradigm, attempting at direct recursive design based on the specification fails; the reason uses to be the need for the introduction of additional variables where intermediate results, which do not appear in the specification of the algorithm, must be kept. In this case, the specification gives no hint on the properties that define the meaning of these new variables, and the help obtained from it in the design is very limited.

We take here the following position: even if new variables must be introduced, we want to attempt at a recursive design. Of course, the reason is the greater simplicity of the correction arguments. A second step could be the transformation of the recursive program into an iterative one, for efficiency reasons. As argued in [BrK], proving the correctness of the recursive program must be easier since the verification of the iterative one amounts frequently to verify simultaneously the original recursive program and the soundness of the transformation.

The result of adding new variables to our program, is called an *embedding*; i.e. a new, more general function which has either more arguments, more results, or both, and which for certain values of the new arguments computes, among other data, the result of the original function. Thus, the problem of designing recursively a program for the original function can be transformed into designing a program for the embedding function. A methodology for the design of recursive programs from their specification exists; thus, our central problem is to derive the specification of the embedding.

We propose here a method for designing recursive programs by embedding, by deriving the specification of the embedding from the initial specification; we show its correctness, and demonstrate its usefulness presenting a few examples and showing how the formal methods for loop derivation are exactly the particular case of our method for the obtention of tail recursive embeddings. The issue of efficiency is also discussed, by analyzing forms of improving the efficiency of a program through the use of additional embeddings, as well as the relationship with known techniques of program transformation.

We briefly summarize the design of recursive programs from their specification in section 2; this section aims mainly at setting the stage for our work and introducing our notation. The core of the paper is section 3, where our method is presented and discussed. The relationship to program transformation and, in particular, to recursion removal, is discussed in section 4: we show there the close relationship between the specification we obtain for the embedding and the invariants of the resulting loops. We devote section 5 to the most important particular case, tail recursion; we indicate the particular way of applying the general method in order to attempt at a tail recursive final design; we discuss how this method gives as particular applications the formal derivation of loops as described in [Dij_a] and [Gri], and the folding-unfolding method for recursion removal ([BuD], [ArK]), which can be seen as an embedding transformation in which the obtained program is tail recursive. Section 6 discusses the use of embedding in improving the efficiency of programs. We close the paper with a section of conclusions. In the appendix we present a complete, unified example illustrating several steps discussed in the body of the paper.

2. Recursive Design

A recursive function is one which has invocations to itself inside its definition. A linear recursive function is the particular case in which, at most, one recursive invocation is made for each activation of the function. In what follows, we restrict our attention to this important case as it gives place to well-known transformation techniques.

Recursive design, and correctness proving of recursive programs, are based on induction principles. We summarize the main ideas about the subject, that have been exposed elsewhere (e.g. see [Ars, Sch]), adapting them to the notation that will be used in the rest of the paper.

Let **func** $f(\bar{x} : T_1)$ **ret** $(\bar{y} : T_2)$ be the function we wish to design. First of all, we must formally specify it by establishing its precondition $Q(\bar{x})$, which defines the allowed values for the arguments, and its postcondition $R(\bar{x}, \bar{y})$ that defines how the results $\bar{y}$ are related to the arguments $\bar{x}$ assuming these satisfy the precondition.

Two are the main ideas guiding the recursive design of f:

- to decompose the value $\bar{x}$ into values $\bar{x}'$, of the *same type* T_1 as $\bar{x}$, so that the solution $\bar{y}$ for $f(\bar{x})$ can be easily calculated from the solution $\bar{y}'$ of $f(\bar{x}')$. We will call $\bar{x}'$ a subproblem of $\bar{x}$.
- to ensure that data $\bar{x}'$ will be, in some well defined sense, smaller than $\bar{x}$.

Using the same argument, the solution for $\bar{x}'$ can be expressed in terms of a smaller subproblem $\bar{x}''$, and so on. The crucial point to ensure the correctness of the design is to prove that the descending sequence $(\bar{x}, \bar{x}', \bar{x}'', \ldots)$ is not infinite. In this way, there will be minimal element(s) whose solution can be calculated without further decompositions. Let us call $b_s(\bar{x})$ the predicate that characterizes these minimal values (s is for "simple"), and let $e(\bar{x})$ be the solution for f in these cases. When $\bar{x}$ is not simple enough, we decompose it to get a smaller value $\bar{x}'$. Let us call $b_r(\bar{x})$ (r for "recursive") the predicate that characterizes the non minimal values, and s the function that calculates the decomposition, so $\bar{x}' = s(\bar{x})$ (s for "successor"). From the result $\bar{y}'$ we calculate the solution for $f(\bar{x})$ by means of a new function $c(\bar{y}', \bar{x})$ (c for "combine"). This design is reflected in the generic program for f of figure 1. If c is not needed, i.e. if $f(\bar{x}) = f(s(\bar{x}))$ in the recursive case, f is called *tail recursive*.

Correctness proofs are based on noetherian induction, which is based on the concept of well-founded sets (see e.g. [LoS]). The property we want to prove is that the function satisfies its specification, i.e. that satisfies $R(\bar{x}, f(\bar{x}))$ for all $\bar{x}$ in $E = \{\bar{x} \mid Q(\bar{x})\}$, doing a finite number of recursive calls. So, we define a strict preorder relation $\prec$ on E such that $(E, \prec)$ is a well-founded set (WFS) in which $s(\bar{x}) \prec \bar{x}$ holds. An easy way to guarantee that $(E, \prec)$ is a WFS is to define a function between E and the natural numbers, $t : E \mapsto \mathbb{N}$ and then define the relation $\prec$ as follows

$$\forall a, b \in E : a \prec b \iff t(a) < t(b)$$

```
{$Q(\bar{x})$}
func $f(\bar{x})$ ret $(\bar{y})$ is
  if $b_s(\bar{x}) \rightarrow$ ret $e(\bar{x})$
  $\square$ $b_r(\bar{x}) \rightarrow$ ret $c(f(s(\bar{x})), \bar{x})$
  fi
end-func
{$R(\bar{x}, \bar{y})$}
```
Figure 1. Generic linear recursive function

where $<$ is the usual strict ordering on $\mathbb{N}$. Since all our programs will be strong-terminating (in the sense of [Dij_b]), such a morphism always exists.

Based on all the above, the steps to show the correctness of a recursive design are the following:

1. prove that $Q(\bar{x}) \Rightarrow (b_s(\bar{x}) \vee b_r(\bar{x}))$, i.e. that f is defined in all its domain.
2. prove that $Q(\bar{x}) \wedge b_r(\bar{x}) \Rightarrow Q(s(\bar{x}))$, i.e. f is always activated inside its domain.
3. establish the induction base: prove that $Q(\bar{x}) \wedge b_s(\bar{x}) \Rightarrow R(\bar{x}, e(\bar{x}))$.
4. prove the induction step: $Q(\bar{x}) \wedge b_r(\bar{x}) \wedge R(s(\bar{x}), \bar{y}') \Rightarrow R(\bar{x}, c(\bar{y}', \bar{x}))$.
5. define $t : E \rightarrow \mathbb{N}$ to convert E into a WFS, i.e. for an appropriate t, show that $Q(\bar{x}) \Rightarrow (t(\bar{x}) \in \mathbb{N})$.
6. show that $Q(\bar{x}) \wedge b_r(\bar{x}) \Rightarrow (s(\bar{x}) \prec \bar{x})$; i.e. each activation of f receives a value smaller than the previous activation.

In 12.1 of [Gri] the following principle is stated: "A program and its proof should be developed hand-in-hand, with the proof usually leading the way". This principle is also applicable in the case of recursive design. Steps 1 to 6 may be used as guidelines for the design. The recommended steps for designing f are the following:

a) study the simple case, and establish b_s and e such that the function finishes satisfying the postcondition. This fulfils proof argument 3.
b) establish b_r (this is easy from Q and b_s) and design the successor function in such a way that progress is made towards termination, i.e. towards satisfying b_s, and the invariance of Q is preserved. If this succeeds, this step satisfies arguments 1, 2, 5 and 6.
c) design function c, which modifies results $\bar{y}'$, in such a way that invariance of the postcondition R is preserved. This is equivalent to proof argument 4.

3. Design by embedding

We have informally defined what is an embedding: a generalization of a function giving another function with more parameters, more results, or both. Only the addition of parameters is useful for designing. Both the case analysis and the function body depend on the function parameters. Indeed, adding more parameters may help us in discovering new recursion relations that were imposible before. On the other hand, adding more results does not help in designing the function since neither the case analysis nor the function body depend on them.

Let f be the function we want to design and let g be a generalization of f. Their specifications can be written as

$$\begin{array}{ll} \{Q(\bar{x})\} & \{Q'(\bar{x},\bar{w})\} \\ \textbf{func}\, f(\bar{x})\ \textbf{ret}\ (\bar{y}) & \textbf{func}\, g(\bar{x},\bar{w})\ \textbf{ret}\ (\bar{y}) \\ \{R(\bar{x},\bar{y})\} & \{R'(\bar{x},\bar{w},\bar{y})\} \end{array}$$

For the sake of simplicity, we will restrict ourselves to designs that leave the parameter $\bar{x}$ unmodified. This may seem a severe restriction but it is not. If we are trying an embedding design is, probably, because a direct recursive design was not possible. So, it will be reasonable to try a design by embedding that only modifies the added parameter. However, if someone ever needs to modify $\bar{x}$ it is always possible to copy it (or part of it) in $\bar{w}$ so that the original $\bar{x}$ is left untouched.

We must specify g by writing the appropiate pre- and postconditions $Q'(\bar{x},\bar{w})$ and $R'(\bar{x},\bar{w},\bar{y})$. First, we want to use g instead of f and this means that g must satisfy the postcondition of f for some suitable precondition Q'. So, we essay to write the following implication for g

$$Q'(\bar{x},\bar{w}) \Rightarrow R(\bar{x},g(\bar{x},\bar{w}))$$

Note that R here does not depend on $\bar{w}$; thus, if we could design g according to the above equation, then we also could design f without any embedding at all. The function g will be easier to design if we manage to introduce $\bar{w}$ in R, thus weakening it. One possibility is to substitute $\bar{w}$ for some expression $\phi(\bar{x})$ in R, obtaining a weaker predicate R' such that

$$R'(\bar{x},\bar{w},g(\bar{x},\bar{w}))\Big|_{\phi(\bar{x})}^{\bar{w}} = R(\bar{x},g(\bar{x},\bar{w}))$$

If we call $P(\bar{x},\bar{w})$ the substitution equation ($\bar{w} = \phi(\bar{x})$) then $R' \wedge P \Rightarrow R$. Thus Q' may not guarantee exactly R, but the weaker R'. At some moment before the end of g the predicate P must hold to guarantee R. We can assure that P holds either before the call to g, by an appropriate initialization of the embedding parameters, or at the end of g. In the latter case we will have both R' and P in the postcondition of g, while in the former P is established independently of g and thus R' suffices as postcondition.

Now we turn our attention to the precondition. Every restriction that f puts on $\bar{x}$ must be preserved on g since there is no reason (and no need) to let g be more defined than f with respect to $\bar{x}$. It will be a restriction on $\bar{w}$, $D(\bar{w})$ (from *D*omain) that exclude those values that make R' false or undefined. Of course, the domain of $\bar{w}$ must include $\phi(\bar{x})$, i.e. $D(\phi(\bar{x})) = true$.

Let us assume that we can find A and B such that $A \wedge B \Rightarrow R'$. Then the precondition will be

$$Q'(\bar{x},\bar{w}) = Q(\bar{x}) \wedge D(\bar{w}) \wedge A$$

where A restricts even more the domain of $\bar{w}$ and may be, in some cases, a "part" of the postcondition R' which does not depend on g. If R' is not in conjunctive form this is achieved by $A = true$ and $B = R'$. In this case the precondition reduces to the domains of $\bar{x}$ and $\bar{w}$.

If R' is in conjunctive form, we can try (among other possibilities) the substitution defined by $\phi(\bar{x}) = g(\bar{x},\bar{w})$ which makes $P(\bar{x},\bar{w}) = (\bar{w} = g(\bar{x},\bar{w}))$. This is the same as saying that in some moment of the computation $\bar{w}$ will hold the final value of g. So, when this occurs we have reached the end of the recursion, both P and R' hold which implies R, and no further computation is needed; we have tail recursion. Some of the conjunctions of R' (or all of them) must appear in the precondition; it is a design decision which ones. Be aware not to put too much restrictions on Q' so that the initialization will be difficult, nor to put too few because then you will not be able to design the function. We will come back to this topic in section 5.

Now g is completely specified and may be designed. However, we are looking for a program that computes f, but we only have a program that computes g. All we have to do is to adapt the latter to the former. If we have tail recursion, P is satisfied an so is R. Then, any value $\bar{w}_0$ that satisfies $Q(\bar{x}) \Rightarrow Q'(\bar{x}, \bar{w}_0)$ guarantees that when we feed g with such value it behaves like f. If we have nontail recursion, then $P(\bar{x}, \bar{w})$ is not satisfied yet, and we must select a value $\bar{w}_0$ that guarantees $Q(\bar{x}) \Rightarrow (Q'(\bar{x}, \bar{w}_0) \wedge P(\bar{x}, \bar{w}_0))$ and initialize $\bar{w}$ to that value.

According to the above discussion, the method of recursive design by embedding consists of the following steps (it is assumed that f has been specified a priori):

1. Write the postcondition of f in the form $R(\bar{x}, g(\bar{x}, \bar{w}))$.
2. Weaken $R(\bar{x}, g(\bar{x}, \bar{w}))$ by replacing an expression $\phi(\bar{x})$ by $\bar{w}$, obtaining R'. Record the substitution in the form of a predicate $P(\bar{x}, \bar{w}) = (\bar{w} = \phi(\bar{x}))$.
3. Define the domain of $\bar{w}$ (and call it $D(\bar{w})$).
4. Choose A and B such that $A \wedge B \Rightarrow R'$.
5. Define $Q'(\bar{x}, \bar{w}) = (Q(\bar{x}) \wedge D(\bar{w}) \wedge A)$.
6. Design recursively the embedding function, and ensure that P holds upon termination.

First example

Let $a[1{:}n]$ be an array of integers with $n \geq 0$. Design a function that computes the sum of the elements in a. The specification is

$$\{true\}$$
$$\textbf{func}\ sum(a : array; n : nat)\ \textbf{ret}\ (s : int)$$
$$\{s = \sum_{i=1}^{n} a[i]\}$$

Following the design procedure we find that

$$R(a, n, g(a, n, k)) = \left(g(a, n, k) = \sum_{i=1}^{n} a[i] \right)$$

where $\bar{x} = (a, n)$ and $\bar{w} = k$. There are a few possibilites for weakening R

$R'(a, n, k, g(a, n, k))$		$P(a, n, k)$	
$g(a, n, k) = k$	with	$k = \sum_{i=1}^{n} a[i]$	(a)
$g(a, n, k) = \sum_{i=1}^{k} a[i]$	with	$k = n$	(b)
$g(a, n, k) = \sum_{i=k}^{n} a[i]$	with	$k = 1$	(c)

```
$\{1 \leq k \wedge k \leq n+1\}$
func $esum(a : array; n, k : nat)$ ret $(s : int)$ is
  if $k = n+1 \rightarrow$ ret 0
  $\square\ k \neq n+1 \rightarrow$ ret $a[k] + esum(a, n, k+1)$
  fi
end-func
$\{s = \sum_{i=k}^{n} a[i]\}$
```

Figure 2. Program for the first example

The possibilities (a) and (b) are nonsense; (a) assumes we know how to compute the sum independently of the call to g, and reduces g to a projection function. On the other hand, (b) forces us to compute exactly the same as before. The third possibility (c) is good.

The precondition of g will be $1 \leq k \wedge k \leq n+1$ because R' is not in conjuctive form. This corresponds to the factor $D(\bar{w})$ in the expression of $Q'(\bar{x}, \bar{w})$ of the method. We choose $k = n+1$, where the summation is zero, as the simple case, and the recursive case as $k \neq n+1$. The complete design is shown on figure 2.

The embedding parameter k must be initialized to some suitable value, say k_0, if we want *esum* behave like *sum*. Since $P = (k = 1)$ the only possible value for initializing k is 1, and the equality $sum(a, n) = esum(a, n, 1)$ holds.

Once the specification of the embedding is obtained, the design decisions are up to the programmer. In this example a linear recursive program was obtained. But no one is committed to do so. A program based on the divide and conquer paradigm could equaly well be designed, leading us to multiple recursion.

Second example

Design a function that computes the integer part of the square root of a natural number. Its specification is

$$\{true\}$$
$$\textbf{func}\ root(n : nat)\ \textbf{ret}\ (r : nat)$$
$$\{r^2 \leq n \wedge n < (r+1)^2\}$$

First we have, with $\bar{x} = n$ and $\bar{w} = a$

$$R(n, g(n, a)) = (g(n, a)^2 \leq n \wedge n < (g(n, a) + 1)^2)$$

It seems that the only possible substitution is the constant 1. In the next example we will follow this way. But now try to substitute the whole function g. This gives

$$R'(n, a, g(n, a)) = (a^2 \leq n \wedge n < (a+1)^2)$$
$$P(n, a) = (a = g(n, a))$$

Since R' is in conjunctive form we can break in into two parts and one of them will be guaranteed by the precondition. Call A the first conjunction and B the second one (but it could be equally well the opposite way). The domains of the natural-valued parameters n and a are both *true* so the precondition is $Q'(n, a) = A = (a^2 \leq n)$.

```
{$a^2 \le n$}
func $eroot(n, a : nat)$ ret $(r : nat)$ is
   if $n < (a+1)^2 \rightarrow$ ret $a$
   □ $n \ge (a+1)^2 \rightarrow$ ret $eroot(n, a+1)$
   fi
end-func
{$r^2 \le n < (r+1)^2$}
```
Figure 3. Program for the second example

Now we begin the design of the function. The simple case is defined by B because $A \wedge B \Rightarrow R'$. When we fulfil R' we must return the value of $g(n,a)$ but P defines it to be a. So we have not only the specification, but also the definition of $b_s(n,a)$ and $e(n,a)$ (see figure 1). The complete program is shown in figure 3.

Since P is satisfied at the same time as the postcondition, it does not define any set of possible initial values for the embedding parameter (as the previous example does). Any value of the embedding parameter a that satisfies the precondition is a valid initialization. The simplest is $a_0 = 0$, but note that we may compute an approximation of the square root by some other means and initialize a to this approximation, provided that it satisfies the precondition. So, if we want to keep things simple, we must look for a constant or a simple initializing function.

Third example

Take the same example as before but this time substitute the constant 1 for an embedding parameter a. The postcondition will be

$$R'(n, a, g(n,a)) = (g(n,a)^2 \le n \wedge n < (g(n,a)+a)^2)$$

with $P(n,a) = (a = 1)$. Although R' is in conjunctive form, none of the conjuncts can be moved to the precondition, since then the precondition of g will depend of g. So, as in the first example, we will have nontail recursion. The precondition will consist only of the domains of n and a; there is no restriction over n but we must avoid the value 0 for a since then R' evaluates to false. The specification of this new embedding is

```
{$a \ge 1$}
func $eroot(n, a : nat)$ ret $(r : nat)$
{$r^2 \le n \wedge n < (r+a)^2$}
```

The first conjunct in R' is satisfied trivially if we return 0. This forces us to ensure that $n < a^2$, and we take it as the simple case. The recursive case is, thus, $n \ge a^2$ and progressing towards the simple case requires to increase a, e.g. by doubling it. The relation between $eroot(n,a)$ and $eroot(n, 2*a)$ is

$$eroot(n,a) = \begin{cases} s & \text{si } n < (s+a)^2 \\ s+a & \text{si } n \ge (s+a)^2 \end{cases} \quad \text{where} \quad s = eroot(n, 2*a)$$

The complete algorithm is shown if figure 4.

Adapting *eroot* for the computation of *root* is simple since P requires $a = 1$, so $root(n) = eroot(n, 1)$. It is interesting to note that $eroot(n,a) = a\lfloor\sqrt{n}/a\rfloor$.

```
{$a \geq 1$}
func $eroot(n, a : nat)$ ret $(r : nat)$ is
   if $n < a^2 \rightarrow$ ret $0$
   $\Box\ n \geq a^2 \rightarrow$ def $s \equiv eroot(n, 2 * a)$;
                  if $n < (s + a)^2 \rightarrow$ ret $s$
                  $\Box\ n \geq (s + a)^2 \rightarrow$ ret $s + a$
                  fi
   fi
end-func
{$r^2 \leq n \wedge n < (r + a)^2$}
```
Figure 4. Program for the third example

```
func $f(\bar{x})$ ret $(\bar{y})$ is
   var $\bar{z}$
   $\bar{z} := \bar{x}$;
   do $b_r(\bar{z}) \rightarrow \bar{z} := s(\bar{z})$ od;
   $\bar{y} := e(\bar{z})$;
   do $\bar{z} \neq \bar{x} \rightarrow \bar{z} := is(\bar{z}); \bar{y} := c(\bar{y}, \bar{z})$ od;
   ret $\bar{y}$
end-func
```
Figure 5. Generic transformation of a linear recursive function

4. Recursion removal

We address in this section the question of the construction of iterative programs equivalent to the recursive embeddings obtained by our method. We concentrate on the case that the obtained embedding has linear recursion. Although the transformation of recursive programs into iterative ones is a well-studied topic, and even more in the case of linear recursive programs ([Ars], [ArK], [BuD], [BrK]), we expect to convince the reader that the remarks presented in this section are useful for understanding of the benefits of our method. The discussion will also be useful to develop in the next section some considerations regarding the case of tail recursion and its relationship to the formal derivation of loops.

We follow here the patterns of transformation proposed in [BrK]. More discussion considering [ArK] and [BuD] will be presented in the next section. In particular, we want to point out some comments regarding the invariants of the loops obtained by the transformation of linear recursive programs into iterative ones. Consider the recursive scheme of figure 1.

The equivalent iterative scheme of figure 5 is proposed in [BrK], with small notational adjustments. It consists of two consecutive loops, corresponding respectively to the forward computation and the backward computation. The auxiliar function $is(\bar{z})$ recovers the value of $\bar{z}$ corresponding to the previous recursive call; sometimes it can be computed directly if s is injective, otherwise the successive values of $\bar{z}$ must be kept into an appropriate data structure (a stack), and the function is operates on it.

```
func $g(\bar{x}, \bar{w})$ ret $(\bar{y})$ is
    $\{Q'(\bar{x}, \bar{w})\}$
    if $b_s(\bar{x}, \bar{w}) \rightarrow$ ret $e(\bar{x}, \bar{w})$
    $\square$ $b_r(\bar{x}, \bar{w}) \rightarrow$ ret $c(g(\bar{x}, s(\bar{w})), \bar{x}, \bar{w})$
    fi
    $\{R'(\bar{x}, \bar{w}, \bar{y})\}$
end-func
```

Figure 6. Generic linear recursive embedding

```
func $g(\bar{x}, \bar{w})$ ret $(\bar{y})$ is
    var $\bar{z}$
    $\bar{z} := \bar{w}$;
    do $b_r(\bar{z}) \rightarrow \bar{z} := s(\bar{z})$ od;
    $\bar{y} := e(\bar{x}, \bar{z})$;
    do $\bar{z} \neq \bar{w} \rightarrow \bar{z} := is(\bar{z}); \bar{y} := c(\bar{y}, \bar{x}, \bar{z})$ od;
    ret $\bar{y}$
end-func
```

Figure 7. Generic transformation of a linear recursive embedding

Invariants for these loops are provided in [BrK] in terms of the activation sequence, using the auxiliar function *is* and also an auxiliar functional *iter* to denote the successive iterations of the function s on the inital values $\bar{x}$. These invariants are rather general, and have the form of an existential quantifier asserting that $\bar{z}$ can be obtained from $\bar{x}$ by iteration of s, an universal quantifier asserting that for none of the previously found values of $\bar{z}$ the simple case predicate $b_s(\bar{z})$ evaluates to true, and for the second loop a third clause asserting that the current value of $\bar{y}$ is the result of f on the current value of $\bar{z}$. In most particular applications, these general invariants have to be rewritten into more precise forms, closer to the actual application, and usually much simpler. In case of tail recursion, the second loop is unnecessary, since after the first assignment to $\bar{y}$ the postcondition holds and $\bar{y}$'s value is invariant through the second loop.

Now assume that, for a given function $f(\bar{x})$, an embedding $g(\bar{x}, \bar{w})$ has been designed according to our method:

$\{Q(\bar{x})\}$	$\{Q'(\bar{x}, \bar{w})\}$
func $f(\bar{x})$ **ret** $(\bar{y})$	**func** $g(\bar{x}, \bar{w})$ **ret** $(\bar{y})$
$\{R(\bar{x}, \bar{y})\}$	$\{R'(\bar{x}, \bar{w}, \bar{y})\}$

where Q' and R' are as before. Assume g designed with linear recursion as in figure 6. We are assuming that $\bar{x}$ is not modified from one activation of g to the next. If modifications are needed, a copy of $\bar{x}$ should be initially included in $\bar{w}$.

Its iterative version is given in figure 7, where *is* is such that $s(is(\bar{z})) = \bar{z}$ (whether it is implemented directly or using a stack).

Now we claim that the predicates computed during the development provide valuable information for the design of invariants for the loops; indeed, Q' is the core of the invariant for the forward computation loop, while R' (with $\bar{y}$ instead of g) is the core of the invariant for the backward computation loop. To see this, observe that the recursive design of g has to guarantee that the precondition holds before the recursive call. Therefore, $Q'(\bar{x},\bar{z}) \wedge b_r(\bar{z}) \Rightarrow Q'(\bar{x}, s(\bar{z}))$, which is the invariance of Q'. At the end of the first loop, since Q' implies $b_s \vee b_r$ but b_r does not hold, b_s holds and therefore $R'(\bar{x},\bar{z},\bar{y})$ holds before the second loop by the correctness of the nonrecursive case $e(\bar{x},\bar{z})$. Finally, the correctness of the recursive design of g guarantees that $R'(\bar{x},s(\bar{z}),\bar{y}) \Rightarrow R'(\bar{x},\bar{z},c(\bar{y},\bar{x},\bar{z}))$ when $b_r(\bar{x},\bar{z})$ holds, and using $s(is(\bar{z})) = \bar{z}$ this argument can be transformed into the proof of the invariance of R': $R'(\bar{x},\bar{z},\bar{y}) \Rightarrow R'(\bar{x},is(\bar{z}),c(\bar{y},\bar{x},is(\bar{z})))$ when $b_r(\bar{x},is(\bar{z}))$ holds. It may be the case that ensuring b_r requires to add to the invariant of the first loop clauses guaranteeing that for all values $is(\bar{z})$, $b_r(\bar{x},is(\bar{z}))$ is true; in most cases, however, this is easy to do using the domain clause $D(\bar{z})$ which is part of $Q'(\bar{x},\bar{z})$.

We are interested in invariants in verification not only as a formal game, useful but sometimes infeasible to play, but as a powerful tool to understanding the program and its behavior, as well as for documentation purposes in communication between programmers. In this sense, the assertions Q' and R' as partial invariants, although possibly insufficient for a formal proof, convey a lot of very useful information compared to the amount of work required to find them, if our methodology is followed; moreover, completing them up to formally sufficient invariants should not be a difficult task. The usual reason for the insufficiency is that, in the recursive version, the correctness argument regards simultaneously each iteration of the forward loop with the corresponding iteration of the backward loop, while the verification of the iterative program must record all necessary information about each iteration in the first loop, collecting these facts for use during the verification of the second loop. The hints provided in [BrK] may be followed in case that difficulties are found during the verification.

Fourth example

We end this section by pursuing further the third example of section 3, presenting its iterative version and discussing the appropriate invariants. Its specification was:

$$\{a \geq 1\}$$
$$\textbf{func}\ eroot(n, a : nat)\ \textbf{ret}\ (r : nat)$$
$$\{r^2 \leq n \wedge n < (r+a)^2\}$$

Following the scheme above, from the recursive solution in figure 4 of section 3 we obtain the algorithm of figure 8. In it, the new parameter of embedding is no longer necessary in the heading and therefore has been omitted, being initialized to 1 as section 3 points out.

The invariance of Q' is immediate. Trying to establish the invariance of R', it is easy to see that a piece of additional information is needed, namely that a is always even; since it is being halved each iteration, the invariant for the second loop must declare a to be a power of 2. Finally, in order that the modified invariant holds before entering the second loop, we must add the same fact to the invariant for the first loop; its invariance is again immediate. This completes the verification process.

Observe that the fact that a is a power of two was unnecessary for the verification of the recursive version. Now we need to record all the facts achieved at each iteration of the first loop, in order to use them at each iteration of the second loop.

```
func $eroot(n : nat)$ ret $(r : nat)$ is
    var $a : nat$
    $a := 1$;
    $\{Q' : a \geq 1\}$
    do $n \geq a^2 \rightarrow a := 2 * a$ od;
    $r := 0$;
    do $a \neq 1 \rightarrow$
    $\{R' : (r^2 \leq n \wedge n < (r+a)^2)\}$
        $a := a$ div $2$;
        if $n < (r+a)^2 \rightarrow skip$
        $\Box\ n \geq (r+a)^2 \rightarrow r := r + a$
        fi
    od;
    $\{R : (r^2 \leq n \wedge n < (r+1)^2)\}$
    ret $r$
end-func
```

Figure 8. Iterative program for the fourth example

5. Tail recursion

It can be observed from the examples in the previous sections that the proposed methodology may yield tail recursive embeddings. Since this particular case is important from the point of view of efficiency, we discuss in this section how to obtain tail recursive embeddings, when possible, via our methodology, and how this relates to the formal derivation of loops from their invariants as described by [Dij_a] and [Gri]. The method consists of obtaining R' as above, but keeping the complete condition R as postcondition; this usually requires to perform a certain kind of substitutions in the design of R' and to select a stronger Q', as we shall see.

Let us consider the formally necessary facts to obtain a tail recursive design. The central point is, of course, that as soon as any recursive call ends, by reaching the nonrecursive case, its result must fulfil postcondition R, and not just postcondition R'. Therefore, during this section, the postcondition for the embedding will be always R. However, we keep the name R' for an auxiliary predicate playing exactly the same role as the R' of the previous sections. Now it must be possible to design a nonrecursive case

$$\Box\ b_s(\bar{x}, \bar{w}) \rightarrow e(\bar{x}, \bar{w})$$

such that the value $e(\bar{x}, \bar{w})$ can be returned as a result of the very first recursive call; therefore $R(\bar{x}, e(\bar{x}, \bar{w}))$ must hold.

Step 2 of the methodology presented in section 3 suggests the use of following equation for designing $R'(\bar{x}, \bar{w}, g(\bar{x}, \bar{w}))$:

$$(P(\bar{x}, \bar{w}) \wedge R'(\bar{x}, \bar{w}, g(\bar{x}, \bar{w}))) \Rightarrow R(\bar{x}, g(\bar{x}, \bar{w}))$$

where $P(\bar{x}, \bar{w})$ usually has the form $\bar{w} = \phi(\bar{x})$ indicating the substitution made on $R(\bar{x}, g(\bar{x}, \bar{w}))$. If the result $e(\bar{x}, \bar{w})$ returned by the nonrecursive case must satisfy

$R(\bar{x}, g(\bar{x}, \bar{w}))$, the natural way of achieving it is ensuring that it satisfies $P(\bar{x}, \bar{w}) \wedge R'(\bar{x}, \bar{w}, g(\bar{x}, \bar{w}))$. Let us see how to satisfy this.

An appropriate embedding is obtained by selecting some of the components of the (vectorial) variable $\bar{w}$, and making them correspond to simple expressions using each of the components of $g(\bar{x}, \bar{w})$, for instance $g(\bar{x}, \bar{w})$ itself. In this way, the substitution of $\bar{w}$ for $\phi(\bar{x})$ must capture *all* the occurrences of $g(\bar{x}, \bar{w})$ in $R(\bar{x}, g(\bar{x}, \bar{w}))$. This yields a predicate R' in which g does not appear, so that in the implication $P \wedge R' \Rightarrow R$ above only $P(\bar{x}, \bar{w})$ depends of $g(\bar{x}, \bar{w})$. Then usually simple manipulation allows one to isolate the conditions imposed by $P(\bar{x}, \bar{w})$ on the values of $g(\bar{x}, \bar{w})$, so that

$$P(\bar{x}, \bar{w}) \iff (P'(\bar{x}, \bar{w}) \wedge g(\bar{x}, \bar{w}) = e(\bar{x}, \bar{w}))$$

for some expression $e(\bar{x}, \bar{w})$. This provides easily the nonrecursive case, by selecting b_s such that $b_s \Rightarrow P'(\bar{x}, \bar{w})$ and in this case returning $e(\bar{x}, \bar{w})$ as result. Furthermore, in order to get R' we need that $Q'(\bar{x}, \bar{w}) \wedge b_s$ implies all of $R'(\bar{x}, \bar{w})$, which is now independent of g since all the occurrences disappeared in the substitution. The particular case that $Q'(\bar{x}, \bar{w})$ coincides with all of $R'(\bar{x}, \bar{w})$ should not be discarded, since it may sometimes be a useful decision.

Now, we argue that the nonrecursive case is correct, since $R'(\bar{x}, \bar{w})$ is implied altogether by $Q'(\bar{x}, \bar{w})$ and b_s, and the returned value is $g(\bar{x}, \bar{w}) = e(\bar{x}, \bar{w})$, which added to $b_s \Rightarrow P'(\bar{x}, \bar{w})$ guarantees $P(\bar{x}, \bar{w}) \wedge R'(\bar{x}, \bar{w}, g(\bar{x}, \bar{w}))$ and therefore $R(\bar{x}, g(\bar{x}, \bar{w}))$. The design should be completed by finding initial values of $\bar{w}$ establishing $Q'(\bar{x}, \bar{w})$ for the first call, and a successor function $s(\bar{w})$ which progresses towards the nonrecursive case and maintains $Q'(\bar{x}, \bar{w})$ true. A reasonable guard $b_r(\bar{x}, \bar{w})$ for the recursive case is usually required to show that $Q'(\bar{x}, s(\bar{w}))$ holds; observe that $Q'(\bar{x}, \bar{w})$ must imply $b_s(\bar{x}, \bar{w}) \vee b_r(\bar{x}, \bar{w})$, for instance by taking $b_r(\bar{x}, \bar{w}) = \neg b_s(\bar{x}, \bar{w})$.

Fifth example

The second example in section 3 corresponds to this particular way of applying the methodology. Let us develop one more example, again for computing the integer part of the square root of a natural number. Recall its specification from section 3

$$\{true\}$$
$$\textbf{func}\ root(n : nat)\ \textbf{ret}\ (r : nat)$$
$$\{r^2 \leq n \wedge n < (r+1)^2\}$$

Here the input $\bar{x}$ corresponds to n. This time, let us use a "vectorial" embedding, i.e. let us introduce two variables: $\bar{w} = \langle a_1, a_2 \rangle$.

$$R(n, g(n, a_1, a_2)) = (g(n, a_1, a_2)^2 \leq n \wedge n < (g(n, a_1, a_2) + 1)^2)$$

The substituted formula is $\phi(\bar{x}) = \langle g(n, a_1, a_2), g(n, a_1, a_2) + 1\rangle$ which, according to our indications, gives a predicate R' in which g does not occur. We obtain

$$R'(n, a_1, a_2) = ({a_1}^2 \leq n \wedge n < {a_2}^2)$$
$$P(n, a) = (\langle a_1, a_2 \rangle = \langle g(n, a_1, a_2), g(n, a_1, a_2) + 1\rangle$$

As expected, P has two parts. One of them is $a_1 = g(n, a_1, a_2)$, and indicates that upon reaching the nonrecursive case, the variable a_1 contains the value to be returned; the

```
{$a_1{}^2 \leq n \wedge n < a_2{}^2$}
func $eroot(n, a_1, a_2 : nat)$ ret $(r : nat)$ is
  if $a_2 = a_1 + 1 \rightarrow$ ret $a_1$
  $\square$ $a_2 \neq a_1 + 1 \rightarrow$ ret $eroot(n, move(n, a_1, a_2))$
  fi
{$r^2 \leq n \wedge n < (r+1)^2$}
end-func
```
Figure 9. Program for the fifth example

other is $a_2 = g(n, a_1, a_2) + 1$, which can be expressed equivalently as $a_2 = a_1 + 1$, and gives the predicate detecting that the nonrecursive case has been reached. The complete formula $(a_1{}^2 \leq n \wedge n < a_2{}^2)$ can be installed as precondition Q', and we obtain a scheme such as that of figure 9.

There, the function *move* decreases the distance between a_1 and a_2, e.g. by halving the interval and setting to the middle point either a_1 or a_2, in a way that preserves the truth of Q'. Thus we obtain a function that searches for the square root by binary search. The design of *move* is easy since we have its specification.

Comparison with formal derivation of loops

It is interesting to compare the design procedure just described with the formal derivation of iterative programs [Dij], [Gri]. More precisely, in 16.1 of [Gri] the three first (and most useful) suggested ways of weakening a predicate are:

1. Deleting a conjunct.
2. Substituting a constant by a variable.
3. Enlarging the range of a variable.

However, enlarging the range of a variable can be seen as a particular case of deleting a conjunct, that which specifies the smaller range; and substituting a constant (or other expression) by a variable, say $\phi(\bar{x})$ by $\bar{w}$, can be seen as transforming first the postcondition R into $R' \wedge (\phi(\bar{x}) = \bar{w})$, and then enlarging the range of a variable by deleting a conjunct, obtaining the invariant. Thus, a unified view of the three methods consists of introducing, if necessary, a new variable, and then deleting a conjunct, which will later become the exit condition for the loop. The initialization must satisfy the remaining conjuncts. All this coincides with the methodology presented above; of course, the coincidence is not casual.

Indeed, if the methodology presented here is used to obtain a tail recursive function, we may use the deleted conjunct as a guard for the nonrecursive case, which corresponds to the exit condition for the loop; we may use the remaining part R' as precondition Q', and during the design care should be taken that the guard for the recursive case b_r and the precondition Q' force the new precondition of the recursive call to hold: $b_r(\bar{x}, \bar{w}) \wedge Q'(\bar{x}, \bar{w}) \Rightarrow Q'(\bar{x}, s(\bar{w}))$. This condition coincides exactly with the fact that Q' is invariant for the loop **do** $b_r(\bar{x}, \bar{w}) \rightarrow \bar{w} := s(\bar{w})$ **od**, which is indeed the iterative version of the tail recursive program we obtain. Finiteness follows in both the tail recursive and the iterative programs from the same argument, namely that $s(\bar{w})$ indeed progresses towards termination, as shown by the decrement of the appropriate natural-valued bound function t (or as decrement of $\bar{w}$ in the preorder induced by t).

Hence, we argue that our methodology for the derivation of recursive embeddings can be seen as a generalization of the formal derivation of loops as described in [Gri], by considering the derivation of a loop as the derivation of a tail recursive program whose iterative version is the same loop, and whose correctness is proved using the same arguments.

The fact that the more general methodology is useful is shown by the example in section 4, where the design of a single recursive embedding yields, when translated into iterative programming, both loops at once, while the formal derivation of a loop directly yields the second one, and a new argument on the proper initialization is required for designing the first one (see [Dij_a], pp. 63–65). Thus we believe that the design of non-tail recursion is useful for formal derivation of loops in that it presents more clearly the internal structure of the designed programs.

A look to folding-unfolding

The folding-unfolding method, as described in [BuD], allows in some cases, to find a generalization of an already designed nontail recursive function, and to deduce from it the invariant of the single iterative loop that computes the function. This amounts to find a recursive-to-recursive transformation from a nontail version to a tail one. The generalization consists, among other things, of introducing new variables in the loop which will keep track of some partial computations. Equivalently, the same effect could be achieved by adding new parameters to the recursive function. So, the method can be seen as a particular case of embedding. In [ArK], a systematic way of finding that generalization, together with a smooth method to derive, first the generalization, then the tail recursive version and then, the iterative one, are given. We will summarize it, following [ArK], applying the transformations to an abstract program and, then, show how the design could be achieved using the method proposed in this paper.

Let the program of figure 1 in section 2 be the initial nontail recursive version. The generalization proposed in [ArK] consists of doing a first-order unification of the term representing the recursive case of f, introducing variables $\bar{x}$ and $\bar{w}$ to match the subtrees with operators other than c and f. This unification gives origin to the generalized function g that, if all goes right, will be tail recursive. In the case of figure 1 in section 2, $g(\bar{x}, \bar{w}) = c(f(\bar{x}), \bar{w})$.

Under some suitable conditions for the additional parameters $\bar{w}$, the function g will compute $f(\bar{x})$. The condition in our abstract program is $\bar{w}$ to be a neutral element $\bar{w}_0$ of c. So, $f(\bar{x}) = g(\bar{x}, \bar{w}_0)$. The second step is to unfold f in the definition of g obtaining, for the recursive case, the expression $c(c(f(s(\bar{x})), \bar{x}), \bar{w})$. If operators in c satisfy some nice properties, the expression can be folded back to match the general structure of g for the recursive case. In our program, the property we need for c is associativity, obtaining the equivalent term $c(f(s(\bar{x})), c(\bar{x}, \bar{w}))$, that after folding gives $g(s(\bar{x}), c(\bar{x}, \bar{w}))$. The unfolding-folding process is applied also to the simple case of g, finally obtaining the abstract program of figure 10. The recursive function g is transformed to an iterative one in the usual way, giving the program of figure 11.

The invariant of the single loop, following [BrK], is $Q(\bar{z}) \wedge (g(\bar{x}, \bar{w}_0) = g(\bar{z}, \bar{w}))$. Using the definition of g, this is equivalent to $Q(\bar{z}) \wedge (f(\bar{x}) = c(f(\bar{z}), \bar{w}))$.

Sixth example

Let us apply the method to the program, obtained by embedding, of figure 2, which is nontail recursive. The generalization is, obviously,

$$g(a, n, k, s) = s + esum(a, n, k)$$

```
func $g(\bar{x}, \bar{w})$ ret $(\bar{y})$ is
    if $b_s(\bar{x}) \rightarrow$ ret $c(e(\bar{x}), \bar{w})$
    $\square$ $b_r(\bar{x}) \rightarrow$ ret $g(s(\bar{x}), c(\bar{x}, \bar{w}))$
    fi
end-func
```

Figure 10. Tail recursive function after the folding-unfolding process

```
func $f(\bar{x})$ ret $(\bar{y})$ is
    var $\bar{z}, \bar{w}$
    $\langle \bar{z}, \bar{w} \rangle := \langle \bar{x}, \bar{w}_0 \rangle;$
    do $b_r(\bar{x}) \rightarrow \langle \bar{z}, \bar{w} \rangle := \langle s(\bar{z}), c(\bar{z}, \bar{w}) \rangle$ od;
    ret $c(e(\bar{z}), \bar{w})$
end-func
```

Figure 11. Iterative version of the program in figure 10

```
func $g(a : array; n, k : nat; s : int)$ ret $(sum : int)$ is
    if $k = n + 1 \rightarrow$ ret $s$
    $\square$ $k \neq n + 1 \rightarrow$ ret $g(a, n, k + 1, s + a[k])$
    fi
end-func
```

Figure 12. Folding-unfolding of the program in figure 2

```
func $esum(a : array; n, k : nat)$ ret $(sum : int)$ is
    var $j : nat; s : int$
    $\langle j, s \rangle := \langle k, 0 \rangle;$
    do $j \neq n + 1 \rightarrow \langle j, s \rangle := \langle j + 1, s + a[j] \rangle$ od;
    ret $s$
```

Figure 13. Iterative version of the program in figure 12

with $s_0 = 0$, obtaining the program of figure 12, whose iterative version, with s initialized to 0, is in figure 13.

The postcondition of this last version is that of the initial one, i.e. $sum = \sum_{i=k}^{n} a[i]$ and the invariant, according to the above, is

$$1 \leq k \leq j \leq n + 1 \wedge \sum_{i=k}^{n} a[i] = s + \sum_{i=j}^{n} a[i]$$

Note that this expression gives, implicitly, the value of parameter s, which represents the *already calculated* part of the function: $s = \sum_{i=k}^{j-1} a[i]$.

Folding-unfolding in our framework

The folding-unfolding method can be seen as a particular case of recursive design using embedding. Not only the precondition and postcondition of the generalized function g can be derived using our method, according to the steps given in sections 2 and 3, but also, we know in advance that the function will be tail recursive. Of course, the method assumes that the design of g follows the usual patterns of recursive design summarized in section 2, and it does not attempt to design g by some other means. In this aspect, folding-unfolding is superior, as it delivers a completely designed function. However, we feel that the essence of folding-unfolding is to find the appropriate embedding and this, using the heuristic of [ArK], is preserved in our approach. The important point, we think, is to have in a single method, well-known strategies for designing programs such as the Dijkstra-Gries derivation calculus, or the folding-unfolding transformation method, as particular cases.

First of all, note that, in the general embedding method, we are assuming that original parameters $\bar{x}$ do not change along the activations of the generalized function $g(\bar{x}, \bar{w})$. This is not true in folding-unfolding, where both $\bar{x}$ and $\bar{w}$ are modified from one activation of g to the next one. This forces us to introduce a "copy" of $\bar{x}$, say $\bar{x}'$, that would have to be considered part of the additional parameters $\bar{w}$. For the sake of clarity we will write it explicitly in what follows.

Let $Q'(\bar{x}, \bar{x}', \bar{w})$ and $R'(\bar{x}, \bar{x}', \bar{w}, g(\bar{x}, \bar{x}', \bar{w}))$ be, respectively the pre- and postcondition of g. As we are trying to get g tail recursive, the postcondition has to be the same as that of the original function f, that is

$$R'(\bar{x}, \bar{x}', \bar{w}, g(\bar{x}, \bar{x}', \bar{w})) = R(\bar{x}, g(\bar{x}, \bar{x}', \bar{w}))$$

The usual substitution for $\bar{w}$ in these cases is an expression containing $g(\bar{x}, \bar{x}', \bar{w})$. Let us try $\bar{w} = g(\bar{x}, \bar{x}', \bar{w})$, that is, $\bar{w}$ conveys the desired result when g terminates. Note that it is not possible to use $\bar{x}'$ for this purpose, as $\bar{x}'$ is assumed to be a copy of $\bar{x}$ and to end with a value $\bar{x}_s$ that satisfes the simple case of both f and g; i.e. $b_s(\bar{x}_s) = \mathit{true}$ (see figures 1 and 10). Up to now we have

$$R'(\bar{x}, \bar{x}', \bar{w}) \wedge (\bar{w} = g(\bar{x}, \bar{x}', \bar{w})) \Rightarrow R(\bar{x}, g(\bar{x}, \bar{x}', \bar{w}))$$

Next, to derive the precondition $Q'(\bar{x}, \bar{x}', \bar{w}) = Q(\bar{x}) \wedge D(\bar{x}') \wedge D(\bar{w}) \wedge A$, we have to express R' as a conjunction of two predicates A and B such that $A \wedge B \Rightarrow R'$. Note, by the way, that $D(\bar{x}') \Rightarrow Q(\bar{x}')$ as $\bar{x}'$ is a copy of $\bar{x}$ that has, as initial value, a value in $Q(\bar{x})$ and, as final value, $\bar{x}_s$. To find A and B, we make use of the [ArK] heuristics and write $g(\bar{x}, \bar{x}', \bar{w}) = c(f(\bar{x}'), \bar{w})$. We know that, when g terminates, R' will be satisfied. That occurs in the simple case $b_s(\bar{x})$ and, in this case, f returns $e(\bar{x})$. Then, we rewrite R' in the following conjunctive form

$$(g(\bar{x}, \bar{x}', \bar{w}) = c(f(\bar{x}'), \bar{w})) \wedge b_s(\bar{x}') \Rightarrow R'(\bar{x}, \bar{x}', \bar{w})$$

Note, by the way, that it will also be true the following

$$b_s(\bar{x}') \Rightarrow g(\bar{x}, \bar{x}', \bar{w}) = \bar{w} = c(e(\bar{x}'), \bar{w})$$

That is, $e(\bar{x}')$ is, for the simple case $\bar{x}'_s$, a neutral element of c.

Now, we take as A the first term (the second will only be true at termination of g), and obtain finally as precondition for g

$$Q'(\bar{x}, \bar{x}', \bar{w}) = Q(\bar{x}) \wedge D(\bar{x}') \wedge D(\bar{w}) \wedge (g(\bar{x}, \bar{x}', \bar{w}) = c(f(\bar{x}'), \bar{w}))$$

This can be expressed in a more useful way by noting that $\bar{y} = g(\bar{x}, \bar{x}', \bar{w}) = f(\bar{x})$ satisfies $R(\bar{x}, \bar{y})$, and $\bar{y}' = f(\bar{x}')$ satisfies $R(\bar{x}', \bar{y}')$.

The initial values of $\bar{x}'$ and $\bar{w}$ will be derived, as usual, from the equation

$$Q(\bar{x}) \Rightarrow Q'(\bar{x}, \bar{x}'_0, \bar{w}_0)$$

The [ArK] heuristic gives us a simple possibility: $\bar{x}'_0 = \bar{x}$ and $\bar{w}_0$ the neutral element of c as, in this case, $g(\bar{x}, \bar{x}', \bar{w}) = f(\bar{x})$, but the equation is more general in the sense that leaves place for other valid initializations.

Sixth example (revisited)

Let us apply all the above to the same example used for illustrating the folding-unfolding method. The postcondition of the program of figure 2, translated to make disappear the bounded variable s would be

$$R(a, n, k) = ((g(a, n, k, j, w) = \sum_{i=k}^{n} a[i])$$

We substitute the expression $g(a, n, k, j, w)$ by w and obtain

$$R'(a, n, k, j, w) = (w = \sum_{i=k}^{n} a[i])$$

We use the [ArK] heuristics to get: $A = (g(a, n, k, j, w) = w + esum(a, n, j))$ and $B = (j = n + 1)$. It is easy to see that $D(j) = (k \leq j \wedge j \leq n + 1)$ and $D(w) = true$. So, arranging A in a more convenient way, we get for the precondition of g

$$Q'(a, n, k, j, w) = ((1 \leq k \leq j \leq n + 1) \wedge \sum_{i=k}^{n} a[i] = w + \sum_{i=j}^{n} a[i]$$

From these predicates, and knowing that the final value of j is $n + 1$ it is fairly easy to design g and obtain the tail recursive program of figure 12. The obvious initial values for j and w are, by [ArK], $j = k$ and $w = 0$. But, if we know, for instance, that $k < n+1$ the values $j = k + 1$ and $w = a[k]$ will also do the job.

6. Efficiency through embedding

When we design algorithms following formal methodologies we sometimes obtain solutions that are correct but are far away from what is used to call an efficient solution. This is a serious drawback that is often argued against formal methods. Then, most programmers obtain an efficient algorithm using informal heuristics that may lead to incorrect results.

Among other reasons, the inefficiency of algorithms comes from the use of costly operations instead of cheaper ones or from the repetition of nearly the same computations instead of reusing previous results. In all these situations, it is useful to maintain the necessary information that will enable us to avoid repeated calculations. In recursive design, this information is maintained in additional parameters or in additional results depending on where it is needed; i.e. if it is needed before or after the recursive call respectively. The embedding technique turns out to be very appropriate to solve both situations: the *parameter embedding* and the *result embedding*.

Parameter embedding

Parameter embedding is aproppriate when the costly or inefficient operations are before the recursive call; with respect to the figure 1 those represented by the functions b_s, b_r, e, and s. This has been proposed elsewhere (e.g. [DoM]). We formalize this technique in our framework.

For each costly expression that we are going to optimize, a new parameter must be added to the function making a new embedding. The type of the parameter must be that of the expression and the precondition must be completed with an equality relation between the parameter and the expression it substitutes.

Assume that $f(\bar{x})$ is a function (or an embedding) and $\phi(\bar{x})$ is the expression appearing in the body of f that is to be optimized. A new function $g(\bar{x},\bar{w})$ must be defined and its precondition set to $Q'(\bar{x},\bar{w}) = (Q(\bar{x}) \wedge \bar{w} = \phi(\bar{x}))$, where $Q(\bar{x})$ is the precondition of f. The postcondition does not change because we want the same function. Now g is designed following the very same design as f; i.e. copying its body almost exactly but with two exceptions. First, all occurrences of $\phi(\bar{x})$ are replaced by $\bar{w}$. This preserves correction since the precondition ensures that $\bar{w} = \phi(\bar{x})$ holds. Second, the successor function s must be completed so that the recursive call satisfies the precondition for $\bar{w}$ too. Note that we may use $\bar{w}$ in this calculation.

This parameter embedding gives us a more efficient function if reestablishing the precondition for it is less expensive than calculating the expression that it substitutes.

When we want to use the embedding instead of the function all we have to do is to initialize $\bar{w}$ to $\phi(\bar{x})$. Since initializations are often simple expressions, it is very possible that $\phi(\bar{x})$ will evaluate also to a simple expression.

Seventh example

Looking at the third example (figure 4), we see that each call computes the expression a^2. Introduce a new parameter d such that $d = a^2$ and add this equality to the precondition. The resulting specification for the embedding is

$$\{a \geq 1 \wedge d = a^2\}$$
$$\textbf{func}\ \mathit{eeroot}(n, a, d : \mathit{nat})\ \textbf{ret}\ (r : \mathit{nat})$$
$$\{r^2 \leq n \wedge n < (r+a)^2\}$$

Accordingly to figure 4, we replace all occurrences of a^2 by d. Note that the expression $(s+a)^2$ that follows the recursive call may be expanded and the term a^2 replaced by d. Then, to reestablish the precondition on d we must feed the recursive call with $(2*a)^2 = 4*d$. The program is shown on figure 14.

Unfortunately, the result is not very efficient since the products r^2 and $r*a$ still appear in the program. There is no way to carry in another parameter the value of r^2 or of $r*a$ since both depend on the value returned by the recursive call. So, we cannot optimize this algorithm any more with a parameter embedding. However, we will see in a moment that a *result embedding* solves this problem.

Result embedding

As we have seen in the above example, the parameter embedding technique does not work if the expressions to be optimized follow the recursive call. With respect to figure 1, those expressions that are part of the function c.

```
{$a \geq 1 \wedge d = a^2$}
**func** $eeroot(n, a, d : nat)$ **ret** $(r : nat)$ **is**
   **if** $n < d \rightarrow$ **ret** $0$
   $\square\ n \geq d \rightarrow$ **def** $r \equiv eeroot(n, 2 * a, 4 * d)$;
               **if** $n < r^2 + 2 * r * a + d \rightarrow$ **ret** $r$
               $\square\ n \geq r^2 + 2 * r * a + d \rightarrow$ **ret** $r + a$
               **fi**
   **fi**
**end-func**
{$r^2 \leq n \wedge n < (r + a)^2$}
```
Figure 14. Optimized program for the seventh example

The mechanism for a result embedding is identical to that of a parameter embedding. Simply define a new result for each expression in c that you want to optimize, and state in the postcondition that the embedding will return the expression as an additional result.

More formally, let $f(\bar{x})$ **ret** $(\bar{y})$ be a function (or any kind of embedding) and let $\phi(\bar{x}, \bar{y})$ be the expression to be optimized. A new function $g(\bar{x})$ **ret** $(\bar{y}, \bar{z})$ must be defined and its postcondition set to $R'(\bar{x}, \bar{y}, \bar{z}) = (R(\bar{x}, \bar{y}) \wedge \bar{z} = \phi(\bar{x}, \bar{y}))$, where $R(\bar{x}, \bar{y})$ is the postcondition of f, and $\bar{z}$ is the new result. Now design g following the design of f but, as in the case of parameter embedding, be aware that you may use $\bar{z}$ instead of $\phi(\bar{x}, \bar{y})$ everywhere after the recursive call; and do not forget to reestablish the postcondition returning the additional result. Of course, you may use $\bar{z}$ in this calculation. Again, the result embedding is more efficient if reestablishing the postcondition is less expensive than computing the substituted expression.

Seventh example (continued)

As you can see in figure 14 there are two expressions that we were not able to optimize using parameter embedding. So, since there are two expressions, define two new results, one for each expression. If we make $s = r^2$ and $t = r * a$ then the specification of the result embedding is

$$\{a \geq 1 \wedge d = a^2\}$$
$$\textbf{func}\ eeeroot(n, a, d : nat)\ \textbf{ret}\ (r, s, t : nat)$$
$$\{r^2 \leq n \wedge n < (r + a)^2 \wedge s = r^2 \wedge t = r * a\}$$

Once the function is specified, design it, using the new results. Next, compute the values that must be returned in s and t to fulfil the postcondition. The final program is shown on figure 15. Note that it uses only sums and shifts (products or divisions by two).

The relation between the original function and the three embeddings is

$$\begin{aligned} root(n) &= eroot(n, 1) \\ &= eeroot(n, 1, 1) \\ &= \pi_1(eeeroot(n, 1, 1)) \end{aligned}$$

where π_1 denotes projection of the first component.

```
func $eeeroot(n, a, d : nat)$ ret $(r, s, t : nat)$ is
   if $n < d \rightarrow$ ret $\langle 0, 0, 0\rangle$
   ▯ $n \geq d \rightarrow$ def $\langle r, s, t\rangle \equiv eeeroot(n, 2 * a, 4 * d)$;
                    if $n < s + t + d \rightarrow$ ret $\langle r, s, t$ div $2\rangle$
                    ▯ $n \geq s + t + d \rightarrow$ ret $\langle r + a, s + t + d, t$ div $2 + d\rangle$
                    fi
   fi
end-func
```
Figure 15. Program for the seventh example with result embedding

```
{$d \geq 1$}
func $froot(n, d : nat)$ ret $(s, t : nat)$ is
   if $n < d \rightarrow$ ret $\langle 0, 0\rangle$
   ▯ $n \geq d \rightarrow$ def $\langle s, t\rangle \equiv froot(n, 4 * d)$;
                    if $n < s + t + d \rightarrow$ ret $\langle s, t$ div $2\rangle$
                    ▯ $n \geq s + t + d \rightarrow$ ret $\langle s + t + d, t$ div $2 + d\rangle$
                    fi
   fi
end-func
{$t^2 \leq d * n < (t + d)^2 \wedge t^2 = s * d$}
```
Figure 16. Final version for the seventh example

Note that if our purpose is to compute $root(n)$ (and not to use any of the embeddings alone) we will always compute the root via the call $eeeroot(n, 1, 1)$. Since, in this case, the first and the third result receive the same value, we may suppress the parameter a and the result r because both of them are useless. Neither the definition of cases nor any expression depends on a or r (except themselves). The figure 16 shows the program *froot* that arises from this simplification. The pre- and postcondition have been rewritten to take into account the simplification. Also we have the relation $root(n) = \pi_2(froot(n, 1))$, where π_2 denotes projection of the second component. It is interesting to note that $\pi_2(froot(n, d)) = d\lfloor\sqrt{n/d}\rfloor$ (compare with the third example).

Call merging

Another situation where efficiency can be gained through embedding arises when, in the recursive case, two (or more) functions are activated with the same actual parameters. Then it is possible to replace both functions by only one with the same parameters as the other two and with the results of both functions. In this way, a new function is created with only one recursive call. This usually gives a lower complexity cost. An example will clarify the idea.

Eighth example

An array of integers $a[1{:}n]$ is given such that $n \geq 0$ and a function is to be designed so that it returns *true* if some element of the array equals the sum of all elements that precede it. Otherwise, the function should return *false*.

```
{$true$}
func $sum\_prec?(a : array; n : nat)$ ret $(b : bool)$ is
    if $n = 0 \rightarrow$ ret $false$
    $\square$ $n > 0 \rightarrow$ ret $sum\_prec?(a, n-1) \vee (a[n] = sum(a, n-1))$
    fi
end-func
{$b = \exists_{i=1}^{n}(a[i] = \sum_{j=1}^{i-1} a[j])$}

{$true$}
func $sum(a : array; n : nat)$ ret $(s : int)$ is
    if $n = 0 \rightarrow$ ret $0$
    $\square$ $n > 0 \rightarrow$ ret $sum(a, n-1) + a[n]$
    fi
end-func
{$s = \sum_{k=1}^{n} a[j])$}
```
Figure 17. Typical solution for the eighth example

```
func $both(a : array; n : nat)$ ret $(b : bool; s : int)$ is
    if $n = 0 \rightarrow$ ret $\langle false, 0\rangle$
    $\square$ $n > 0 \rightarrow$ def $\langle b, s\rangle \equiv both(a, n-1)$;
                 ret $\langle b \vee (a[n] = s), s + a[n]\rangle$
    fi
end-func
```
Figure 18. Linear cost solution for the eighth example

The straightforward design of the problem involves two recursive functions: one for computing the sum of the elements and another that checks if an element equals the sum of its precedents. The specification and the program are shown on figure 17.

Unfortunately this program has a $O(n^2)$ complexity cost. But looking at the two functions we see that their respective recursive calls have the very same actual parameters, and also the very same case analysis. A new function can be defined that is a result embedding of the above two. The specification will collect both specifications, removing duplicate parameters and/or results.

```
{$true$}
func $both(a : array; n : nat)$ ret $(b : bool; s : int)$
{$b = \exists_{i=1}^{n}(a[i] = \sum_{j=1}^{i-1} a[j]) \wedge s = \sum_{i=1}^{n} a[j])$}
```

The design of the embedding follows the case analysis and recursive call of both functions and is shown on figure 18. The embedding allowed us to obtain a linear cost program, thus improving the earlier solution. Different solutions are proposed in the appendix.

7. Conclusions

We have presented a method of recursive design by embedding, based on obtaining in a formal way the specification of the desired embedding, from which the design can be conducted. Since the design step is itself a well-studied subject, our contribution relies on the formal derivation of the specification of the embedding. The addition of parameters or results is required frequently in recursive design, and the initial specifications give no hint on the properties that these parameters must fulfil. Thus, design methods which require a specification to work from it must be complemented by methods of finding the specification in case of embedding. We have proposed tools for deriving this specification.

Several examples of the application of this method have been presented. Furthermore, we have argued its usefulness by showing that other methods of design, such as formal derivation of loops from their invariants or the folding-unfolding transformation method, may be viewed as particular cases of our method, in which the obtained program is tail recursive. Thus the case of tail recursion has been discussed in depth, due to its special significance. We have also shown that the specification of the embedding is closely related to the invariants of the loops after the transformation into iterative programs.

The use of embedding to enhance the efficiency of programs has been also discussed; the addition of parameters or of results, such as merging of recursive calls, has been studied. The combination of all methods in an example of a complete design is presented in the appendix.

A main motivation for this work stems for the teaching of recursive programming in the Programming Methodology course at the authors' University. In this course, derivation of loops according to the methods of [Gri] is one of the central topics. Trying to teach to program in a scientific way requires to develop in the students the attitude of rigorous reasoning. Since recursive design is another of the topics of the lectures, a coherent approach requires to develop similar methods for the design of correct recursive programs; for a given specification, the design methods described in section 2 suffice, but when an embedding is required the available methods were not sufficient. This was clear from the fact that many students had trouble in identifying correctly the new specification. Being aware of no methods for guiding this task, we have developed ours. We expect the work reported here to be useful in the teaching of good programming practices.

References

[Ars] Arsac, J. *Les bases de la programmation*, Dunod, Paris 1983.

[ArK] Arsac, J.; Kodratoff., Y. Some Techniques for Recursion Removal from Recursive Functions. *ACM Trans. on Programming Languages and Systems* 4, 2 (Apr. 1982), 295–322.

[BrK] Broy, M.; Krieg-Brückner, B. Derivation of Invariant Assertions During Program Development by Transformation. *ACM Trans. on Programming Languages and Systems* 2, 3 (July 1980), 321–337

[BuD] Burstall, R.; Darlington, J. A Transformation System for Developing Recursive Programs. *Journal of the ACM* 24, 1 (Jan. 1977), 44–67.

[Dij_a] Dijkstra, E. *A Discipline of Programming*, Prentice-Hall 1976.

[Dij_b] Dijkstra, E. EWD673 On Weak and Strong Termination. In: *Selected Writings on Computing: A Personal Perspective*, Springer-Verlag 1982, 355–357.

[DoM] Dosch, W.; Möller, B. Seminar on Functional Programming, Madrid, October 1987.

[Gri] Gries, D. *The Science of Programming*, Springer-Verlag 1981.

[LoS] Loeckx, J.; Sieber, K. *The Foundations of Program Verification*. Wiley-Teubner Series in Computer Science, John Wiley & Sons Ltd. 1984.

[Sch] Scholl, P. *Algorithmique et représentation des données 3. Récursivité et arbres*, Masson, Paris 1984.

Appendix

Here we develop an example that uses embedding in the design, efficiency and transformation to iterative steps.

Problem

Given an array of integers $a[1{:}n]$ where $n \geq 0$, design a function that returns *true* if some element of the array equals the sum of all elements that precede it. Otherwise, the function should return *false*.

Design step

The specification of the function is

$$\{true\}$$
$$\textbf{func}\ sum\_prec?(a : vector; n : nat)\ \textbf{ret}\ (b : bool)$$
$$\{b = \exists_{i=1}^{n}(a[i] = \sum_{j=1}^{i-1} a[j])\}$$

An embedding similar to that of the first example (see figure 2) gives

$$\{1 \leq k \leq n+1\}$$
$$\textbf{func}\ esum\_prec?(a : vector; n, k : nat)\ \textbf{ret}\ (b : bool)$$
$$\{b = \exists_{i=k}^{n}(a[i] = \sum_{j=1}^{i-1} a[j])\}$$

The case analysis is also identical since the embedding is very much the same. The reason is the strong similarity that exists between the two examples: both operate on an array and both compute an associative function extended to all the elements of the array; the former computes the integer sum and the latter, a boolean sum. The program for the embedding is shown on figure A–1.

The program is incomplete since the summation in the recursive case is not implemented. If we resort to a function for implementing the sum we will have an overall cost of $O(n^2)$ since the sum function will have a linear cost. Of course, we may design it but it is an innecessary extra work because in a later step we will try to optimize that expression.

Tail recursion embedding step

Following section 5, we will find the embedding that gives us a tail recursive function. Let us call it *tsum_prec?*, and write the postcondition

$$R(a, n, k) = (tsum\_prec?(a, n, k, l, u) = \exists_{i=k}^{n}(a[i] = \sum_{j=1}^{i-1} a[j]))$$

where l is introduced because our method forbids the modification of parameters other than the embedding ones. The parameter u appears as a consequence of the generalization strategy of [ArK] (see section 5). Next we replace $tsum\_prec?(a, n, k, l, u)$ by u to get

$$R'(a, n, k, l, u) = (u = \exists_{i=k}^{n}(a[i] = \sum_{j=1}^{i-1} a[j]))$$

```
func $esum\_prec?(a : vector; n, k : nat)$ ret $(b : bool)$ is
    if $k = n + 1 \rightarrow$ ret $false$
    $\square\ k \leq n \rightarrow$ ret $esum\_prec?(a, n, k + 1) \vee (a[k] = \sum_{j=1}^{k-1} a[j])$
    fi
end-func
```

Figure A–1. Program for the design embedding

```
func $tsum\_prec?(a : vector; n, k, l : nat; u : bool)$ ret $(b : bool)$ is
    if $l = n + 1 \rightarrow$ ret $u$
    $\square\ l \leq n \rightarrow$ ret $tsum\_prec?(a, n, k, l + 1, u \vee (a[l] = \sum_{j=1}^{l-1} a[j]))$
    fi
end-func
```

Figure A–2. Program after the tail recursive embedding

```
func $etsum\_prec?(a : vector; n, k, l : nat; u : bool; s : int)$ ret $(b : bool)$ is
    if $l = n + 1 \rightarrow$ ret $u$
    $\square\ l \leq n \rightarrow$ ret $etsum\_prec?(a, n, k, l + 1, u \vee (a[l] = s), s + a[l])$
    fi
end-func
```

Figure A–3. Program after the efficiency embedding

Using the heuristics suggested by [ArK] we obtain A and B as

$$A = (tsum\_prec?(a, n, k, l, u) = (u \vee esum\_prec?(a, n, l)))$$
$$B = (l = n + 1)$$

The domains of l and u are easily obtained as $k \leq l \leq n + 1$ and *true* respectively. Now we have all the information for writing the precondition of the tail recursive function *tsum_prec?* as

$$Q'(a, n, k, l, u) = (1 \leq k \leq l \leq n + 1 \wedge \exists_{i=k}^{n} S = (u \vee \exists_{i=l}^{n} S))$$

where $S = (a[i] = \sum_{j=1}^{i-1} a[j])$. From the above precondition is now easy to get a tail recursive design. The result is shown on figure A–2.

From the precondition it is clear that the simplest initializations will be $l = k$ (the computation starts at k) and $u = false$ (the neutral element for $\vee$). Note that our program is incomplete again since we have not implemented the summation yet. This example shows that the method remains valid even with incomplete designs.

Efficiency step

Since the expression to be optimized appears before the recursive call, a parameter embedding will be used. Define a new parameter s, and take $s = \sum_{j=1}^{l-1} a[j]$. Next, add this equation to the precondition. Of course, the postcondition is left unchanged. Replace the summation by s and reestablish the precondition for s (using s, of course). The program is shown on figure A–3. This step could equally well be performed before the tail recursion embedding without affecting the final result.

```
func $sum\_prec?(a : vector; n : int)$ ret $(b : bool)$ is
    var $k, l : nat; u : bool; s : int$
    $\langle k, l, u, s\rangle := \langle 1, 1, false, 0\rangle$;
    do $l \leq n \rightarrow \langle k, l, u, s\rangle := \langle k, l+1, u \vee (a[l] = s), s + a[l]\rangle$ od;
    ret $u$
end-func
```

Figure A–4. Iterative version for the stated problem

Recursion removal step

Following section 4 we obtain the iterative version of the program of figure A–3. All additional parameters that we have introduced in the successive embeddings must now be set to an appropriate initial value. From the design step we have that the simplest initialization for k is 1. In the tail recursion step we got $l = k$ and $u = false$. And from the efficiency step we know that $s = \sum_{j=1}^{l-1} a[j]$, i.e. $s = 0$ since $l = k = 1$. Putting together all the initializations we obtain the program of the figure A–4.

In the iterative version we see that the variable k is now superfluous and may be omitted. Finally, the invariant for the single loop of the program of figure A–4 is the precondition for the last embedding (see section 4), which is

$$1 \leq l \leq n + 1 \wedge s = \sum_{j=1}^{l-1} a[j] \wedge \exists_{i=1}^{n} S = (u \vee \exists_{i=l}^{n} S)$$

where $S = (a[i] = \sum_{j=1}^{i-1} a[j])$. The invariant has been simplified replacing k by 1 (its initial value) because k is not modified.