Deriving specifications of embeddings in recursive program design

C. Roselló
J.L. Balcázar
R. Peña

Report LSI-89-13
Deriving Specifications of Embeddings in 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 (eai)
cabalqui@ebrupc51 (bitnet)
ricardo@fib.upc.es (eai)

Resum: El disseny de funcions recursives és un tema ben estudiat. Els mètodes de disseny requereixen una especificació a priori; tanmateix, quan s'intenta un disseny per immersió no queda gens clara la relació que hi ha entre l'especificació de la immersió i la de la funció inicial. Es proposa un mètode de derivar les especificacions de les immersions i es relaciona clarament amb altres mètodes formals prou coneguts tals com la derivació de bucles o la transformació de programes. Es discuteix també l'ús de la immersió per millorar l'eficiència dels programes.

Abstract: The design of recursive functions is a well-studied subject. Design methods require a specification to start with; however, when an embedding design is attempted at, it is not clear how the specification of the embedding relates to that of the initial function. We propose a method for deriving specifications of embeddings. We show how other known formal methods, such as loop derivation or program transformation, can be related to our method in a clear fashion. The use of embedding to enhance the efficiency of programs is also discussed.

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 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. 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.

However, in frequent cases appearing in a programming-in-the-small paradigm, attempting at a direct recursive design based on the specification fails; the reason usually is 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.

For a given function, an embedding function is 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. (This approach is useful in a variety of ways, and we will not survey all of them.) A methodology for the design of recursive programs from their specification exists; thus, our central problem is to derive the specification of the embedding.

A main motivation for this work stems from teaching 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 teaching good programming practices.

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. Two brief examples are developed. Section 4 discusses the use of embedding in improving the efficiency of programs. The relationship with other techniques, such as folding-unfolding, recursion removal, and the formal derivation of loops as described in [Dij-a] and [Gri], is treated in section 5. In an appendix we present a final example that illustrates many of the ideas discussed in the paper.

2. Recursive Design

A recursive function is one which calls itself inside its definition. A linear recursive function is the particular case in which, at most, one recursive call is made for each
$$\{Q(\bar{x})\}$$

func \(f(\bar{x})\) ret (\(\bar{y}\)) is
if \(b_s(\bar{x})\) \(\rightarrow\) ret \(c(\bar{x})\)
\(\Box\) \(b_r(\bar{x})\) \(\rightarrow\) ret \(c(f(s(\bar{x}))), \bar{x}\) fi
end-func
\(\{R(\bar{x}, \bar{y})\}\)

Figure 1. Generic linear recursive function

activation of the function. In what follows, we restrict our attention to this important case.

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, BaW, 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 finite. In this way, there will be some minimal elements 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 nonminimal values, and \(s\) the function that calculates the decomposition, so \(\bar{x}' = s(\bar{x})\) (\(s\) for “successor”). Frequently \(b_r(\bar{x}) = \neg b_s(\bar{x})\), but there is no formal need that this be always so. 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. [BaW, LoS]). The property we want to prove is that the function satisfies its specification, i.e. that it satisfies \(R(\bar{x}, f(\bar{x}))\) for all \(\bar{x}\) in \(E = \{\bar{x} | Q(\bar{x})\}\), doing a finite number of recursive calls. So, we define a strict preorder relation \(<\) on \(E\) such that \((E, <)\) is a well-founded set (WFS) in which \(s(\bar{x}) < \bar{x}\) holds. An easy way to guarantee that \((E, <)\) is a WFS is to define a function between \(E\) and the natural numbers, \(t : E \rightarrow \mathbb{N}\) and then define \(<\) as \(\forall a, b \in E : a < b \iff t(a) < t(b)\) 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 \( Q(\bar{x}) \Rightarrow (b_s(\bar{x}) \lor b_r(\bar{x})) \), i.e. that \( f \) is defined in all its domain.
2. establish the induction base: prove that \( Q(\bar{x}) \land b_s(\bar{x}) \Rightarrow R(\bar{x}, e(\bar{x})) \).
3. prove that \( Q(\bar{x}) \land b_r(\bar{x}) \Rightarrow Q(s(\bar{x})), \) i.e. \( f \) is always activated inside its domain.
4. prove that \( Q(\bar{x}) \land b_r(\bar{x}) \land R(s(\bar{x}), \bar{y}') \Rightarrow R(\bar{x}, e(\bar{y}'), \bar{x})) \).
5. define \( t : E \rightarrow N \) to convert \( E \) into a WFS, i.e. for an appropriate \( t \), show that \( Q(\bar{x}) \Rightarrow (t(\bar{x}) \in N) \).
6. show that \( Q(\bar{x}) \land b_r(\bar{x}) \Rightarrow (s(\bar{x}) = \bar{x}) \), i.e. each activation of \( f \) receives a value smaller than that of the previous activation, in the preorder induced by \( t \).

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. Additional results may be useful to improve the efficiency, as explained in a later section; however, only the addition of parameters is necessary for designing. Indeed, adding more parameters may help us in discovering new recursion relations that were impossible before. On the other hand, adding more results does not help the initial design of 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

\[
\{Q(\bar{x})\}
\text{func } f(\bar{x}) \text{ ret } (\bar{y})
\{R(\bar{x}, \bar{y})\}
\]

\[
\{Q'(\bar{x}, \bar{w})\}
\text{func } g(\bar{x}, \bar{w}) \text{ ret } (\bar{y})
\{R'(\bar{x}, \bar{w}, \bar{y})\}
\]

For the sake of simplicity, we will restrict ourselves to designs that leave the parameter \( \bar{x} \) unmodified; this means that we assume that we will design a successor function (\( s \) in figure 1) which only varies the additional parameters of the embedding. This may seem a severe restriction but it is not. We are trying an embedding design, 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 parameters. 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 appropriate 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 \( g \)

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

(1)

The essence of the method is to weaken this implication. We discuss the two natural possibilities: weakening the right hand side, which we discuss first, and strengthening the left hand side, discussed later.

For the first possibility, we introduce the embedding parameters in \( R \) to give it a conjunctive form, and then we supress a conjunct. The method is to replace some expression \( \phi(\bar{x}) \) in \( R \) by \( \bar{w} \), obtaining a weaker predicate \( R_{\text{weak}} \) such that

\[
R_{\text{weak}}(\bar{x}, \bar{w}, g(\bar{x}, \bar{w}))\bigg|_{\phi(\bar{x})} = R(\bar{x}, g(\bar{x}, \bar{w}))
\]
If we call $P(\bar{x}, \bar{w})$ the substitution equation ($\bar{w} = \phi(\bar{x})$) then $R_{\text{weak}} \land P \Rightarrow R$. (It is not exactly an equivalence because the set of variables is not the same.) Now, we take as the postcondition of the embedding $R' = R_{\text{weak}}$, so that $Q'$ may not guarantee exactly $R$, but the weaker $R_{\text{weak}}$. To assure that $P$ holds as well, we must find an appropriate initialization of the embedding parameters: we must select an initial value $\bar{w}_{\text{init}}$ that guarantees $Q(\bar{x}) \Rightarrow (Q'(\bar{x}, \bar{w}_{\text{init}}) \land P(\bar{x}, \bar{w}_{\text{init}}))$.

In general, the resulting function $g$ will be nontail recursive. Indeed, when the nonrecursive condition $b_s(\bar{x}, \bar{w})$ holds, the parameter $\bar{w}$ will have changed and no longer fulfill $P$, and therefore the condition $R'$ achieved in the last recursive call will be considerably weaker than $R$.

Now we turn our attention to the precondition. Every restriction that $f$ puts on $\bar{x}$ is preserved on $g$ since in general there is no reason (and no need) to let $g$ be more defined than $f$ with respect to $\bar{x}$. (Although it may be the case that in some specific example a more defined function turns out to be useful, we have not found such a case in a large list of examples.) Additionally there may be a restriction on $\bar{w}$, $D(\bar{w})$ (from Domain) excluding those values that make $R'$ false or undefined. Of course, the domain of $\bar{w}$ must include $\phi(\bar{x})$, i.e. $D(\phi(\bar{x})) = \text{true}$. The precondition is $Q'(\bar{x}, \bar{w}) = Q(\bar{x}) \land D(\bar{w})$, and we have obtained the complete specification of the embedding.

The second possibility for weakening the implication (1) is to strengthen the left hand side. The idea that guides now the development is to try to maintain $R$ as the postcondition of the embedding function $g$. Now since $R$ does not depend on $\bar{w}$ we will obtain a tail recursive design; indeed, as soon as any call to $g(\bar{x}, \bar{w})$ terminates, regardless of the value of $\bar{w}$ (since $\bar{x}$ does not change), we have a value satisfying $R$ and therefore we have solved $f(\bar{x})$. The same result can be returned with no modification; this is the definition of tail recursion.

However, in this case more information must be provided by the precondition. We must transfer some information from $R$ to the precondition $Q'$, so that the nonrecursive case can easily obtain $R$ from $Q'$ and the guard $b_s$. We must take care that $Q'$, being the precondition of $g$, does not depend on $g$ itself. On the other hand, it must be possible to easily establish $Q'$ at the initialization.

The expression that leads the design of $Q'$ is

$$Q'(\bar{x}, \bar{w}) \land b_s(\bar{x}, \bar{w}) \Rightarrow R(\bar{x}, e(\bar{x}, \bar{w})) \quad (2)$$

where $b_s$ and $e$ are the guard and the command of the nonrecursive case as in figure 1. It turns out that the following heuristic to decide how to introduce $\bar{w}$ is very useful: devote part of the (vectorial) parameter $\bar{w}$ to convey the result in the nonrecursive case. Then the nonrecursive treatment $e(\bar{x}, \bar{w})$ reduces to return simply some components of $\bar{w}$. Let us denote by $\bar{w}_1$ the components of $\bar{w}$ that will be returned by $e(\bar{x}, \bar{w})$ and by $\bar{w}_2$ the remaining components corresponding to additional parameters. Then expression (2) becomes

$$Q'(\bar{x}, \bar{w}) \land b_s(\bar{x}, \bar{w}) \Rightarrow R(\bar{x}, \bar{w}_1) \quad (3)$$

where $Q'$ and $b_s$ are the unknowns that must be found to proceed with the design. Now it suffices to put the right hand side of (3) in conjunctive form, in order to select some of the conjuncts for $Q'$ and let $b_s$ be the remaining ones.

Of course the simplest case is when $R(\bar{x}, \bar{w}_1)$ has conjunctive form: $\bar{w} = \bar{w}_1$ suffices as embedding. If $R$ does not have conjunctive form, or if we want to add other embedding parameters, we may use again the same method as before to write $R$ as a
conjunction: substitute new parameters $\bar{w}_2$ for some expression $\phi(\bar{x}, \bar{w}_1)$ in $R$, obtain a weaker predicate $R_{\text{weak}}$ such that

$$R_{\text{weak}}(\bar{x}, \bar{w}_1, \bar{w}_2, g(\bar{x}, \bar{w})) \bigg|_{\phi(\bar{x}, \bar{w}_1)} = R(\bar{x}, \bar{w}_1)$$

and use the predicate $R_{\text{weak}} \land P$, which is now in conjunctive form, to find $Q'(\bar{x}, \bar{w})$ and $b_*(\bar{x}, \bar{w})$. Here $P(\bar{x}, \bar{w})$ is, as before, the substitution equation ($\bar{w}_2 = \phi(\bar{x}, \bar{w}_1)$).

The complete precondition will be $Q(\bar{x})$ (i.e. the domain of $\bar{x}$) and the conjuncts selected from $R_{\text{weak}} \land P$. It is possible that a domain condition $D(\bar{w})$ will also be required in the precondition, although this seems much less frequent than in the nontail recursive case.

For the initialization, any value $\bar{w}_{\text{init}}$ that satisfies $Q(\bar{x}) \Rightarrow Q'(\bar{x}, \bar{w}_{\text{init}})$ guarantees that when we compute $g(\bar{x}, \bar{w}_{\text{init}})$ we obtain $f(\bar{x})$. Observe that in general the appropriate value $\bar{w}_{\text{init}}$ is not unique.

Let us give finally a brief account of the method proposed up to now. 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 $R(\bar{x}, g(\bar{x}, \bar{w}))$, so that the goal is expressed as $Q'(\bar{x}, \bar{w}) \Rightarrow R(\bar{x}, g(\bar{x}, \bar{w}))$.
2. Weaken this expression and obtain the new precondition $Q'$ and the new postcondition $R'$.
3. If necessary, define the domain $D(\bar{w})$ of $\bar{w}$ and add it to the precondition.
4. Complete the recursive design of the embedding function.

Step 2 can be further analyzed, but this depends on whether tail recursion is attempted at.

2a. For a nontail recursive design, the weakening is achieved by replacing an expression $\phi(\bar{x})$ by $\bar{w}$, and the result $R_{\text{weak}}$ is taken as the new postcondition $R'$. It is useful to record the substitution in the form of a predicate $P(\bar{x}, \bar{w}) = (\bar{w} = \phi(\bar{x}))$ since it guides the initialization of the parameters $\bar{w}$. Usually $Q'$ reduces to $Q$ and the domain restriction $D(\bar{w})$.

2b. For a tail recursive design, $R$ stays as postcondition. At the nonrecursive case, some components $\bar{w}_1$ of $\bar{w}$ carry the result. We need $R(\bar{x}, \bar{w}_1)$ in conjunctive form; either we have it beforehand, or expressions are replaced by new embedding parameters as in case 2a to obtain $R_{\text{weak}} \land P$ in conjunctive form. Then this predicate is split into $Q'$ and $b_*$.

We present next two examples of embedding development according to this method. In the first one the postcondition is weaker than that of the original function, and nontail recursion is obtained. Later discussion in this paper will refer back again to this example. The second example illustrates the development of a tail recursive embedding.

A third example in which tail recursion is also obtained is presented in the appendix.

**Example 1**

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

$$\{ \text{true} \}$$

$$\text{func} \ root(n : \text{nat}) \ret (r : \text{nat})$$

$$\{ r^2 \leq n \land n < (r + 1)^2 \}$$
\{a \geq 1\} \\
\textbf{func} \textit{eroot}(n, a : \textit{nat}) \textbf{ret} (r : \textit{nat}) \textbf{is} \\
\quad \textbf{if} \ n < a^2 \rightarrow \textbf{ret} 0 \\
\quad \quad \Box n \geq a^2 \rightarrow \textbf{let} s = \textit{eroot}(n, 2 + a) \textbf{in} \\
\quad \quad \quad \textbf{if} \ n < (s + a)^2 \rightarrow \textbf{ret} s \\
\quad \quad \quad \Box n \geq (s + a)^2 \rightarrow \textbf{ret} s + a \\
\textbf{fi} \\
\textbf{end-func} \\
\{r^2 \leq n \land n < (r + a)^2\} \\

\textbf{Figure 2. Program for example 1}

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

\[ R(n, g(n, a)) = (g(n, a)^2 \leq n \land n < (g(n, a) + 1)^2) \]

Substituting an embedding parameter \(a\) for the constant 1, we obtain

\[ R_{\text{weak}}(n, a, g(n, a)) = (g(n, a)^2 \leq n \land n < (g(n, a) + a)^2) \]

with \(P(n, a) = (a = 1)\). We take as the new postcondition \(R' = R_{\text{weak}}\), and we will have non-tail recursion. The precondition will only be 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 \geq 1\} \\
\textbf{func} \textit{eroot}(n, a : \textit{nat}) \textbf{ret} (r : \textit{nat}) \\
\{r^2 \leq n \land n < (r + a)^2\} \\

The remaining steps are those of any recursive design, since we already have the specification. 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, \(n \geq a^2\) and progressing towards the simple case requires to increase \(a\), e.g. by doubling it. Comparing \(\textit{eroot}(n, a)\) with \(\textit{eroot}(n, 2 + a)\) and looking for a relationship between them we get

\[ \textit{eroot}(n, a) = \begin{cases} 
  s & \text{if } n < (s + a)^2 \\
  s + a & \text{if } n \geq (s + a)^2
\end{cases} \quad \text{where} \quad s = \textit{eroot}(n, 2 + a) \]

The complete algorithm is shown in figure 2.

Adapting \(\textit{eroot}\) for the computation of \(\textit{root}\) is simple since \(P\) requires \(a = 1\), so \(\textit{root}(n) = \textit{eroot}(n, 1)\). It is interesting to note that \(\textit{eroot}(n, a) = a\lfloor \sqrt{n/a} \rfloor\).
\{a_1^2 \leq n \land 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
   [] a_2 \neq a_1 + 1 \rightarrow ret eroot(n, move(n, a_1, a_2))
fi
end-func
\{r^2 \leq n \land n < (r + 1)^2\}

Figure 3. Program for example 2

Example 2

To illustrate the tail recursive case, we present a different design of the same problem of computing the integer part of the square root of a natural number. Recall its specification from example 1

\{true\}

func root(n : nat) ret (r : nat)
\{r^2 \leq n \land n < (r + 1)^2\}

First we substitute an embedding parameter \(a_1\) for the result. Since the postcondition presents conjunctive form, several possibilities for the precondition arise, for instance \(a_1^2 \leq n\); these solutions are left to the reader. For this example, we follow another more interesting way: aiming at a binary search, we introduce a second embedding parameter substituting the expression \(a_1 + 1\). We get

\[
R_{\text{weak}}(n, a_1, a_2) = (a_1^2 \leq n \land n < a_2^2) \\
P(n, a_1, a_2) = (a_2 = a_1 + 1)
\]

Then we can choose \(R_{\text{weak}}\) as the precondition and \(P\) as the nonrecursive guard. We obtain a scheme such as that of figure 3.

The function \(move\) must decrease the distance between \(a_1\) and \(a_2\) preserving the truth of \(Q'\). Since we aim at a binary search program we set either \(a_1\) or \(a_2\) to the middle point of the interval \([a_1, a_2]\). The detailed design of \(move\) is left to the reader.

4. Efficiency through embedding

When we design algorithms following formal methods we sometimes obtain solutions that are correct but are far away from what is usually called 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.

Many of the ideas in this section have been treated before, mostly on papers related to program transformation. See, for instance, [BaW], [CIP]. Additional references can be found on the very complete survey by Feather [Fea].

Parameter embedding

Parameter embedding is appropriate when the costly or inefficient operations are before the recursive call; with respect to the figure 1, those represented by the functions $b_v$, $b_r$, $e$, and $s$. This idea has been proposed elsewhere (e.g. [DoM], [PaK], [BMPP]). 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 $f(x)$ is a function (or an embedding) and $\phi(x)$ is the expression appearing in the body of $f$ that is to be optimized. A new function $g(x, w)$ must be defined and its precondition set to $Q'(x, w) = (Q(x) \land w = \phi(x))$, where $Q(x)$ 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 $\phi(x)$ are replaced by $w$. This preserves correctness since the precondition ensures that $w = \phi(x)$ holds. Second, the successor function $s$ must be completed so that the recursive call satisfies the precondition for $w$ too. Note that we may use $w$ 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 $w$ to $\phi(x)$. Since the initialization $x_{\text{init}}$ is often a simple expression, it is very possible that $\phi(x_{\text{init}})$ will also evaluate to a simple expression.

Result embedding

However, 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 $c$.

We propose a mechanism for a result embedding analogous to that of a parameter embedding: we define a new result for each expression in $c$ that we want to optimize, and we state in the postcondition that the embedding will return the expression as an additional result.

More formally, let $f(x)$ ret $(y)$ be a function (or an embedding) and let $\phi(x, y)$ be the expression to be optimized. A new function $g(x)$ ret $(y, z)$ must be defined and its postcondition set to $R'(x, y, z) = (R(x, y) \land z = \phi(x, y))$, where $R(x, y)$ is the postcondition of $f$, and $z$ 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 $z$ instead of $\phi(x, y)$ everywhere after the recursive call; and reestablish the postcondition for the additional result. Of course, you may use $z$ in this calculation. Again, the result embedding is more efficient if reestablishing the postcondition is less expensive than computing the substituted expression.
func `eeeroot(n, a, d : nat) ret (r, s, t : nat) is
    if n < d -> ret (0, 0, 0)
    □ n ≥ d -> let (r, s, t) = eeeroot(n, 2 * a, 4 * d) in
        if n < s + t + d -> ret (r, s, t div 2)
        □ n ≥ s + t + d -> ret (r + a, s + t + d, t div 2 + d)
    fi
end-func

Figure 4. Program for example 1 with optimization embeddings

Example 1 (continued)
Consider the solution "croot" of example 1 (figure 2); 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

\[
\begin{align*}
\{ & a \geq 1 \land d = a^2 \\
\text{func } & eeeroot(n, a, d : nat) \text{ ret (r : nat)} \\
\{ & r^2 \leq n \land n < (r + a)^2 \}
\end{align*}
\]

Accordingly to figure 2, 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 \((2 * a)^2 = 4 * d\).

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, a result embedding allows us to avoid also this recomputation.

Since there are two expressions to optimize, 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

\[
\begin{align*}
\{ & a \geq 1 \land d = a^2 \\
\text{func } & eeeroot(n, a, d : nat) \text{ ret (r, s, t : nat)} \\
\{ & r^2 \leq n \land n < (r + a)^2 \land s = r^2 \land t = r * a \}
\end{align*}
\]

Now, we can proceed with the design using the new results and computing the values that must be returned in \(s\) and \(t\) to fulfill the postcondition. The final program is shown on figure 4. Note that it uses only sums and shifts (products or divisions by two).

The relation between the original function and the three embeddings is

\[
\begin{align*}
\text{root}(n) &= \text{croot}(n, 1) \\
&= \text{eeeroot}(n, 1, 1) \\
&= \pi_1(\text{eeeroot}(n, 1, 1))
\end{align*}
\]

where \(\pi_1\) denotes projection of the first component.
\{d \geq 1\}

\textbf{func} \textit{froot}(n, d : \textit{nat}) \textbf{ret} (s, t : \textit{nat}) \textbf{is}

\textbf{if} n < d \rightarrow \textbf{ret} (3, 0)

\textbf{then} n \geq d \rightarrow \textbf{let} (s, t) = \textit{froot}(n, 4 \ast d) \textbf{in}

\textbf{if} n < s + t + d \rightarrow \textbf{ret} (s, t \textbf{div} 2)

\textbf{then} n \geq s + t + d \rightarrow \textbf{ret} (s + t + d, t \textbf{div} 2 + d)

\textbf{fi}

\textbf{fi}

\textbf{end-func}

\{t^2 \leq d \ast n < (t + d)^2 \land t^2 = s \ast d\}

\textbf{Figure 5. Final version for example 1}

Note that if our purpose is to compute \textit{root}(n) (and not to use any of the embeddings alone) we will always compute the root via the call \textit{ceeroot}(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. Indeed, neither the definition of cases nor any expression depends on \(a\) or \(r\) (except themselves). The figure 5 shows the program \textit{froot} that arises from this simplification. The pre- and postcondition have been rewritten to take into account the simplification. Also we have the relation \(\textit{root}(n) = \pi_2(\textit{froot}(n, 1))\), where \(\pi_2\) denotes projection of the second component. It is interesting to note that \(\pi_2(\textit{froot}(n, d)) = d[\sqrt{n/d}]\) (compare with the similar but different property of the first design of this example).

5. Relationships with other concepts

In this section we discuss the relationships of our method to some important design methods, both for recursive and for iterative programs. We discuss first how our method relates to an important recursive to recursive transformation method, namely folding-unfolding. Then we compare with the recursive-to-iterative transformation method described in [BrK], together with the invariants of the loops as described there. Finally, we compare our method with the loop derivation method as advocated in [Dij-a], [Gri].

A look to folding-unfolding

Arsac and Kodratoff [ArK] use the folding-unfolding method of [BuD] to give a systematic way of finding a tail recursive generalization of an already designed recursive function. We will summarize their proposal, and show its relationship with our tail recursive design method.

Let the program of figure 1 be the initial nontail recursive version. The generalization proposed in [ArK] is obtained by introducing variables \(\tilde{x}\) and \(\tilde{w}\) to match the subterms with operators other than \(c\) and \(f\) in the term representing the recursive case of \(f\). We get the generalization \(g(\tilde{x}, \tilde{w}) = c(f(\tilde{x}), \tilde{w})\). If we unfold \(f\) and next we can fold the whole expression back to \(g\), then we obtain the program of figure 6.

The postcondition must assert that \(g\) computes \(f(\tilde{x})\) under some suitable precondition \(Q'\). We see in figure 6 that all parameters of \(g\) are modified by the successor functions, so \(\tilde{x}\) in \(g\) is not the same as \(\tilde{x}\) in \(f\). In order to distinguish between them, let us call \(\tilde{x}'\) the first parameter of \(g\). Hence we write equation (1) in section 3 as

\[Q'(\tilde{x}, \tilde{x}', \tilde{w}) \Rightarrow g(\tilde{x}', \tilde{w}) = f(\tilde{x})\]  

(4)
\text{func } g(\bar{x}, \bar{w}) \text{ ret } (\bar{y}) \text{ is}
\begin{align*}
& \text{if } b_s(\bar{x}) \rightarrow \text{ ret } c'(c(\bar{x}), \bar{w}) \\
& \square b_r(\bar{x}) \rightarrow \text{ ret } g(s(\bar{x}), c(\bar{x}, \bar{w})) \\
& \text{fi} \\
\text{end-func}
\end{align*}

Figure 6. Program resulting from the folding-unfolding process

To see that the precondition can be provided by our method, recall that in the case of tail recursion, we have to find \( Q' \), a simple case \( B_s \), and a simple case treatment \( E(\bar{x}', \bar{w}) \) (written in upper case to avoid confusion with the simple case and treatment of \( f \)), so that (4) is weakened into

\[ Q'(\bar{x}, \bar{x}', \bar{w}) \land B_s(\bar{x}', \bar{w}) \Rightarrow E(\bar{x}', \bar{w}) = f(\bar{x}) \]  \hspace{1cm} (5)

Of course, we use the heuristic of [ArK] instead of that of section 3

\[ g(\bar{x}', \bar{w}) = c(f(\bar{x}'), \bar{w}) \]

From figure 6 we see that \( B_s(\bar{x}', \bar{w}) = b_s(\bar{x}') \) and that \( E(\bar{x}', \bar{w}) = c(e(\bar{x}'), \bar{w}) \). This is all the information we need. Recording these facts in equation (5) we obtain

\[ Q'(\bar{x}, \bar{x}', \bar{w}) \land b_s(\bar{x}') \Rightarrow c(e(\bar{x}'), \bar{w})) = f(\bar{x}) \]

as the equation for guiding the design of \( Q' \). We can easily put the right hand side into conjunctive form, by observing in the program for \( f \) that if \( b_s(\bar{x}') \) holds then \( f(\bar{x}') = e(\bar{x}') \), and thus

\[ (c(f(\bar{x}'), \bar{w}) = f(\bar{x})) \land b_s(\bar{x}') \Rightarrow c(e(\bar{x}'), \bar{w})) = f(\bar{x}) \]

One of the conjuncts is already \( b_s(\bar{x}') \); we simply take the other one as precondition

\[ Q'(\bar{x}, \bar{x}', \bar{w}) = Q(\bar{x}) \land D(\bar{x}') \land (c(f(\bar{x}'), \bar{w}) = f(\bar{x})) \]

where \( Q(\bar{x}) \) and \( D(\bar{x}') \) (a subset of \( Q(\bar{x}') \)) have been added to ensure that \( f \) is defined. As in section 3, the initial values must satisfy \( Q(\bar{x}) \Rightarrow Q'(\bar{x}, \bar{x}', \bar{w}) \), which here becomes

\[ Q(\bar{x}) \Rightarrow D(\bar{x}') \land (c(f(\bar{x}'), \bar{w}) = f(\bar{x})) \]

An easy way to do it, that can be frequently applied, is to select \( \bar{x} \) as the initial value of \( \bar{x}' \) and a neutral element for \( c \) as the initial value of \( \bar{w} \); in this case it is enough that \( \bar{w} \) behaves as a neutral element when it is composed with \( f(\bar{x}) \).

Thus, we see that the precondition for the embedding could be deduced following the method proposed here for tail recursion, simply by considering the heuristic choice proposed by [ArK] and its consequences on the nonrecursive case of the embedding.
func \( f(\overline{x}) \) ret \( (\overline{y}) \) is

\[
\begin{align*}
\text{var } & \overline{z} \\
\overline{z} & := \overline{x}; \\
\text{do } & b^r(\overline{z}) \rightarrow \overline{z} := s(\overline{z}) \text{ od;} \\
\overline{y} & := e(\overline{z}); \\
\text{do } & \overline{z} \neq \overline{x} \rightarrow \overline{z} := is(\overline{z}); \overline{y} := c(\overline{y}, \overline{z}) \text{ od;} \\
\text{ret } & \overline{y}
\end{align*}
\]

end-func

Figure 7. Generic transformation of a linear recursive function

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. 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], [DaB], [BrK], [BaW]). We follow here the patterns of transformation proposed in [BrK], in order to point out some facts regarding the invariants of the loops obtained by the transformation of linear recursive programs into iterative ones.

Consider the recursive scheme of figure 1. Figure 7 shows its equivalent iterative scheme as proposed in [BrK], with small notational adjustments. It has two consecutive loops, corresponding respectively to the forward computation and the backward computation. The auxiliary function \( is(\overline{z}) \) recovers the value of \( \overline{z} \) corresponding to the previous recursive call; sometimes it can be computed directly if \( s \) is injective, otherwise the successive values of \( \overline{z} \) must be kept in an appropriate data structure (a stack), and the function \( is \) operates on it.

Invariants for these loops are provided in [BrK] in terms of the activation sequence, using the auxiliary function \( is \) and also an auxiliary functional \( \text{iter} \) to denote the successive iterations of the function \( s \) on the initial values \( \overline{x} \). These invariants are rather general, and are based on an existential quantifier asserting that \( \overline{z} \) can be obtained from \( \overline{z} \) by iteration of \( s \), and an universal quantifier asserting that for none of the previously found values of \( \overline{z} \) the simple case predicate \( b^s(\overline{z}) \) evaluates to true. The invariant for the first loop is

\[
\exists i : i \geq 0 : (z = \text{iter}(s, i, x) \land \forall j : 0 \leq j < i : \neg b^s(\text{iter}(s, j, x))
\]

where \( \text{iter}(s, i, x) \) represents the value \( z \) obtained after applying the function \( s \) to \( x \), \( i \) many times. For the second loop a clause is added, asserting that the current value of \( \overline{y} \) is the result of \( f \) on the current value of \( \overline{z} \): \( \overline{y} = f(\overline{z}) \).

If the invariant for the second loop can be established in some easy way, the first loop may be unnecessary. In the tail recursion case, the second loop is unnecessary since after the first assignment to \( \overline{y} \) the postcondition holds and \( \overline{y} \)'s value is invariant through the second loop. Moreover, in most particular applications, these general invariants can be rewritten into more precise forms, closer to the actual application, and usually much simpler.
Now assume that, for a given function \( f(x) \), an embedding \( g(x, w) \) has been designed according to our method

\[
\begin{align*}
\{Q(x)\} & \quad \{Q'(x, w)\} \\
\text{func } f(x) \text{ ret } (y) & \quad \text{func } g(x, w) \text{ ret } (y) \\
\{R(x, y)\} & \quad \{R'(x, w, y)\}
\end{align*}
\]

where \( Q' \) and \( R' \) are as before. Assume \( g \) designed with linear recursion as in figure 8. Its iterative version is given in figure 9, where \( is \) is such that \( s(is(z)) = z \) (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 \( y \) 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, \( Q'(x, z) \land b_r(z) \Rightarrow Q'(x, s(z)) \), which is the invariance of \( Q' \). At the end of the first loop, since \( Q' \) implies \( b_s \lor b_r \) but \( b_r \) does not hold, \( b_s \) holds and therefore \( R'(x, z, y) \) holds before the second loop by the correctness of the nonrecursive case. Observe that in the case of tail recursion this completes the verification process, with \( Q' \) as invariant. For the nontail recursive case, the correctness of the recursive design of \( g \) guarantees that \( R'(x, s(z), y) \Rightarrow R'(x, z, c(y, x, z)) \) when \( b_r(x, z) \) holds, and using \( s(is(z)) = z \) this argument can be transformed into the proof of the invariance of \( R' \): \( R'(x, z, y) \Rightarrow R'(x, is(z), c(y, x, is(z))) \) when \( b_r(x, is(z)) \) 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 \( is(z) \), \( b_r(x, is(z)) \) is true; in most cases, however, this is easy to do using the domain clause \( D(z) \) which is part of \( Q'(x, z) \).

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
func eroot(n : nat) ret (r : nat) is

var a : nat
a := 1;
{Q' : a ≥ 1}
do n ≥ a² → a := 2 * a od;
r := 0;
do a ≠ 1 →
{R' : (r² ≤ n ∧ n < (r + a)²)}
a := a div 2;
if n < (r + a)² → skip
□ n ≥ (r + a)² → r := r + a
fi
od;
{R : (r² ≤ n ∧ n < (r + 1)²)}
ret r
end-func

Figure 10. Iterative program for example 1

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.

Example 1 (continued)

We end this section by pursuing further example 1, presenting its iterative version and discussing the appropriate invariants. Its specification was

\{a ≥ 1\}

func eroot(n, a : nat) ret (r : nat)

\{r² ≤ n ∧ n < (r + a)²\}

Following the scheme above, from the recursive solution in figure 2 we obtain the algorithm of figure 10. In it, the embedding parameter is no longer necessary in the heading and therefore has been omitted, after being initialized to 1.

The invariance of Q' is immediate. When we try to establish the invariance of R', we easily see that a piece of additional information is needed, namely that a is always even; since it is halved in 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. Additional discussion on this "loss of structure" of the iterative versions can be found in [BrK].

Comparison with formal derivation of loops

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

1. Deleting a conjunct.
2. Replacing 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 replacing 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 \( R' \land (\phi(\bar{x}) = \bar{w}) \), and then enlarging the range of a variable by deleting a conjunct, obtaining the invariant. Thus, a unified view of the three techniques of weakening 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.

The methodology presented here can be seen as a more abstract method than formal derivation: we work at the recursive level from where an iterative program can always be obtained. First, when a tail recursive embedding is done, 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 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:

\[
Q'(\bar{x}, \bar{w}) \land b_r(\bar{x}, \bar{w}) \Rightarrow Q'(\bar{x}, s(\bar{w})).
\]

This condition coincides exactly with the fact that \( Q' \) is invariant for the loop \( \text{do } b_r(\bar{x}, \bar{w}) \rightarrow \bar{w} := s(\bar{w}) \text{ od} \), which is indeed the iterative version of the tail recursive program we obtain. Termination follows in both the tail recursive and the iterative programs from the same argument, namely that \( s(\bar{w}) \) indeed progresses towards the simple case (the exit condition), as shown by the decrement of the appropriate natural-valued bound function \( t \) (or as decrement of \( \bar{w} \) in the preorder induced by \( t \)). Thus the tail-recursive design case is essentially the same as the formal derivation of the corresponding loop.

Second, when we have a nontail recursive embedding, the new postcondition is (almost) the invariant of the last loop of the program (the backward loop in the iterative version), and a initialization of the embedding parameters should be made to fulfill the exit condition of that loop, so that when the iterative program "simulates" the end of the outermost call we have:

\[
R'(\bar{x}, \bar{w}, \bar{y}) \land P(\bar{x}, \bar{w}) \Rightarrow R(\bar{x}, \bar{y}).
\]

On the other hand, the nonrecursive case may be viewed as the initialization of the invariant for this last loop; its guard is the condition that allows one to do it easily, although it is possible that another loop (the forward loop) will be necessary to achieve this condition.

Therefore our method provides simultaneously the last loop and, if necessary, the previous loop that serves to initialize the second one, together with the main parts of
the corresponding invariants. The usefulness of this approach is shown, for instance, by example 1, 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.

References


Appendix

Here we develop an example that uses embedding in the design, efficiency and recursion removal 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

\[
\{true\} \quad \text{func} \ \sum_{\text{prec}}?(a: \text{vector}; n: \text{nat}) \ \text{ret} \ (b: \text{bool})
\]

\[
\{b = \exists_{i=1}^{n} (a[i] = \sum_{j=1}^{i-1} a[j])\}
\]

Following section 2 we take $R$ and replace the constant 1 in the existential quantifier by an embedding parameter $k$. Then we choose $R' = R_{\text{weak}}$ and $Q' = D(k)$ where $D(k) = (1 \leq k \leq n + 1)$. The second inequality comes from the definition of $\Sigma$ (i.e. undefined if $k > n + 1$) while the first is required because $k$ must be able to take the value 1, i.e. the value it replaces. The specification of the embedding is

\[
\{1 \leq k \leq n + 1\} \quad \text{func} \ \sum_{\text{prec}}?(a: \text{vector}; n, k: \text{nat}) \ \text{ret} \ (b: \text{bool})
\]

\[
\{b = \exists_{i=k}^{n} (a[i] = \sum_{j=1}^{i-1} a[j])\}
\]

Now we proceed with the design. We take $k = n + 1$ as the simple case because then the existential quantifier gives $false$. For the recursive case we take $k \leq n$ (from $D(k)$ and the negation of the simple case). As the existential quantifier is nonvoid we split it into the first disjunct and a quantifier with the rest of disjuncts. Collecting these design decisions we arrive at the program shown on figure A-1.

However, 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 unnecessarily extra work because in a later step we will try to optimize that sum.

Tail recursion embedding step

Following section 5, let us call $\sum_{\text{prec}}?$ the appropriate generalization found using the heuristics provided by [ArK]

\[
\sum_{\text{prec}}?(a, n, k', u) = u \lor \exists_{i=k'}^{n} (a[i] = \sum_{j=1}^{i-1} a[j])
\]
func esum_prec?(a : vector; n, k : nat) ret (b : bool) is
  if k = n + 1 → ret false
  □ k ≤ n → ret esum_prec?(a, n, k + 1) ∨ (a[k] = \sum_{j=1}^{k-1} a[j])
fi
end-func

Figure A-1. Program after the design embedding

func tsun_prec?(a : vector; n, k' : nat; u : bool) ret (b : bool) is
  if k' = n + 1 → ret u
  □ k' ≤ n → ret tsun_prec?(a, n, k' + 1, u ∨ (a[k'] = \sum_{j=1}^{k'-1} a[j]))
fi
end-func

Figure A-2. Program after the tail recursive embedding

func etsum_prec?(a : vector; n, k' : nat; u : bool; s : int) ret (b : bool) is
  if k' = n + 1 → ret u
  □ k' ≤ n → ret etsum_prec?(a, n, k' + 1, u ∨ (a[k'] = s), s + a[k'])
fi
end-func

Figure A-3. Program after the efficiency embedding

We readily obtain the precondition of the tail recursive embedding as

\[ 1 \leq k \leq k' \leq n + 1 \land (u \lor \bigvee_{i=k'}^{n} (a[i] = S)) = \bigvee_{i=k}^{n} (a[i] = S) \]

where \( S = \sum_{j=1}^{i-1} a[j] \). The program after this folding-unfolding process is shown on figure A-2.

It is easy to see that the simplest initializations will be \( k' = k \) and \( u = false \) (the neutral element for \( \lor \)). Note that our program is incomplete again since we have not implemented the summation yet. This will be the concern of the next step.

**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 \( s = \sum_{j=1}^{k'-1} a[j] \). Next, add this equation to the precondition. Of course, the postcondition is left unchanged. To reestablish the precondition for \( s \) we must feed the function with the expression \( \sum_{j=1}^{(k'+1)-1} a[j] \) which, according to the precondition is rewritten as \( s + a[k'] \). The program is shown on figure A-3.

Note that this step and the tail recursion embedding step could equally well be interchanged without affecting the final result.
\textbf{func} \textit{sum\_prec?} (a : vector; n : int) \textbf{ret} (b : bool) \textbf{is}

\begin{verbatim}
  \textbf{var} k' : nat; u : bool; s : int
  \langle k', u, s \rangle := \langle 1, false, 0 \rangle;
  \textbf{do} k' \leq n \rightarrow \langle k', u, s \rangle := \langle k' + 1, u \lor (a[k'] = s), s + a[k'] \rangle \textbf{od};
  \textbf{ret} u
\end{verbatim}

\textbf{end\-func}

Figure A–4. Iterative version for the stated problem

\textit{Recursion removal step}

Following again section 5 we obtain the iterative version of the program of figure A–3. All additional parameters that we have introduced in the successive embeddings will be removed from the heading and will be set to an appropriate initial value. From the design step we know that the simplest initialization for \( k \) is 1; in the tail recursion step we got \( k' = k \) and \( u = false \); and from the efficiency step we know that \( s = \sum_{j=1}^{k'-1} a[j] \), i.e. \( s = 0 \) since \( k' = k = 1 \). Collecting all the initializations we obtain the program of the figure A–4.

Finally, the invariant for the single loop of the program of figure A–4 is the pre-condition for the last embedding (see section 5), which is

\[ 1 \leq k' \leq n + 1 \land s = \sum_{j=1}^{k'-1} a[j] \land (u \lor \sum_{i=k'}^{n} (a[i] = S)) = \sum_{i=1}^{n} (a[i] = S) \]

where \( S = \sum_{j=1}^{i-1} a[j] \). Here we have replaced \( k \) by its initialization since it does not appear in the iterative program.