PROBLEM SOLVING STRATEGIES FOR THE DERIVATION OF PROGRAMS

Jaime Bohórquez
Rodrigo Cardoso
Universidad de los Andes
Bogotá, Colombia
e-mail: jbohorqu@uniandes.edu.co
rcardoso@uniandes.edu.co

Abstract: Methods and principles inspired in problem solving strategies for program synthesis are presented. This approach complements the calculational style of programming, emphasizing the consideration of the meaning of the formulas involved along the derivation of programs.

Keywords: program development, software quality, software engineering, formal program derivation.

The main goal of computer programming is modeling and representing systems of objects or concepts in a computer to solve problems, perform calculations, and make simulations and predictions. These activities are done by implementing in a precise way the properties and interactions of the objects belonging to those systems.

With this view in mind, since a computer is essentially a symbol manipulation machine, computer programming must be considered a scientific activity. Serious expert programmers and scholars like C.A.R. Hoare, E.W. Dijkstra, D. Gries, W.H.J. Feijen and others have developed a practical approach to this subject, based on methodically developing programs from their specifications ([Hoa69], [Dij76], [Gri81], [Heh84], [Bac86], [Dij88], [Rey88]). In their approach a program and its correctness proofs are obtained hand in hand. This method of programming has two great virtues: on one side, the correctness of a program constructed this way is a subproduct of its development; on the other, it allows reasoning about programs in a non-operational way.

The refinement of this method has led to what is nowadays called the calculational style of programming, where programs are mainly derived from their specifications by means of formula manipulation ([Coh90], [Kal90]). A program together with its specification is viewed as a theorem. The theorem expresses that the program satisfies its specification.

The main goal of this paper is to enrich this method with complementary principles based on problem solving strategies. These principles will allow the programmer to interpret the calculations he is performing to obtain his program, and in addition to the calculational methods, will provide him with guidance to select the appropriate formula manipulations to proceed in order to achieve his task.

As it happens, the presentation of the proof of a mathematical theorem is many times devoid of the motivations inspired by the interpretation of the calculations performed, and leading to the line of thought that conducted to the discovery of the result proved. We consider the inclusion of this motivations an integral part of the proof document. In a similar way, we believe that the consideration of the meaning of the predicates and calculations involved in the construction of a program using the calculational style will add not only to its understanding but also to the creative skills required for its development.

Moreover, as programming in its broadest sense is a problem solving activity, and most of the time it is the real world that imposes problems to us, if appropriate solutions are desired, modeling and interpretation become unavoidable. This is particularly true when it becomes necessary to represent the behavior of a certain family of objects or concepts not directly representable in the computer by means of the available data structures.

We use the notation

\[ \begin{array}{c}
\{ C \quad \{ Q \} \quad S \quad \{ R \} \end{array} \]

to say that the program \( S \) is correct with respect to the specification \( \{ C, Q, R \} \), where \( Q \) and \( R \) are, respectively, a pre- and a postcondition, and \( C \) is a context. The latter is a condition that must hold along the block delimited by the
square brackets, i.e. it must be implicitly understood as integral part of every inner assertion (naturally, subprograms of S may be annotated in the same way). For readability, the context and the precondition are usually written in different lines. We write

\[
[C \ S \ \{R\}]
\]

when it is the case that the precondition is equivalent to T, the universally true predicate.

Variable declarations of the form

\[v_1, \ldots, v_n : T_p\]

introduce the symbol variables \(v_1, \ldots, v_n\), of type \(T_p\), and are included at the beginning of a context expression\(^1\). Therefore, their scope is the block where they are declared. If \(S\) includes another block, the variables declared in the outer block are to be left unchanged by the instructions of the inner block. We use ";" to separate declarations and to signal the end of the list of declarations.

For the sake of simplicity, the ideas presented in this article are illustrated with examples that use very basic and general expressions involving natural (nat), integer (int), and boolean (bool) values. We also use arrays\(^2\) (v.gr. \(m(i, j:\text{range})\) of nat), sets (usual mathematical notation, declared \(\text{set of } \ldots\), or simply set) and sequences (v.gr. \(<1, 3, 0>\), declared \(\text{seq of } \ldots\), or simply \(\text{seq}\)). If necessary, additional notation will be introduced to explain some examples. The code is expressed in Dijkstra's guarded command language, following closely the notations used in [Rem] and [Car93]. Functional application is denoted \(f. x\) instead of \(f(x)\), and we write \(f.g.x\) instead of \(f(g(x))\). With \(R(x := e)\) we denote a predicate syntactically identical to the predicate \(R\), except that the free occurrences of the variable \(x\) have been replaced by the value of the expression \(e\).

We restrict our discussion to consider the construction of programs that are essentially repetitions. Section 1 establishes the reduction principle, from which three main invariant derivation techniques, gradual fulfillment of the goal, information balance, and reducing the uncertainty are inferred. Sections 2, 3 and 4 describe and exemplify these strategies. Section 4 actually presents two techniques for reducing the uncertainty: narrowing the fence and balance of explored and unexplored areas, which could be considered as generalizations of classical search techniques. Section 5 collects the main results and states some conclusions.

1 THE REDUCTION PRINCIPLE

Since the sequential composition and alternation commands of the guarded command language already addresses the problem solving strategies of respectively sequential stage composition and case analysis, we shall concentrate on programming problems whose solutions require the use of repetitions. Thus, the design of suitable invariants becomes crucial in the derivation of solutions to these problems.

The repetition command answers to a very general problem solving principle that could be stated as follows:

**Reduction principle:** Find simpler versions of the problem whose solutions could be used to obtain a solution of the original problem.

Viewing the solution of a problem as the achievement of a certain goal, this principle applies when, for example, there is a formulation of the problem that allows its attainment in terms of the solution of one or more similar subgoals. In other words, the problem can be stated in a recursive way.

---

\(^1\) It is clear that variable declarations are not logic formulas. However, this small abuse of notation should not cause any confusion.

\(^2\) We allow arrays indexed by arbitrary finite sets (e.g. \(\text{range}\) in the example). Moreover, we use arrays as functions, and we use these two terms as synonyms.
We present three general strategies for invariant design, based on the reduction principle:
- Gradual fulfillment of the goal
- Information balance
- Reducing the uncertainty

Two variants of this strategy are presented: narrowing the fence and balance of explored and unexplored areas.

2 GRADUAL FULFILLMENT OF THE GOAL

The strategy of gradual fulfillment of the goal will be useful when the programming problem amounts to evaluate a recursively defined function \( F : D \rightarrow E \), at some point \( X \) in \( D \). We shall assume that the postcondition is of the form

\[
R : r = F \cdot X
\]

We apply the reduction principle to find a solution. Using an auxiliary array variable \( f(e : D) \) to hold the values of the function \( F \) over \( D \) and a set variable \( S \) to keep a subset of \( D \), we propose the following invariant:

\[
P : S \subseteq D \land f(e : S) = F(e : S)
\]

Note that \( P \) constitutes a partial fulfillment of the goal \( P \land X \in S \), which implies that \( f \cdot X = F \cdot X \) holds, and therefore \( R(x := f \cdot X) \) holds also. We choose \( X \notin S \) as a guard. Thus, we obtain:

\[
\begin{array}{l}
\{ S : \text{set}; x : E; f : \text{array}(e : D) \text{ of } E; \\
S := \emptyset; \\
\{ \text{inv } P : S \subseteq D \land f(e : S) = F(e : S) \} \\
do X \notin S \rightarrow \text{body } od; \\
r := f \cdot X \\
\{ R : r = F \cdot X \}
\end{array}
\]

To make progress towards termination, we use the recursive definition of \( F \) to modify \( S \) by including and deleting elements from it. The modification should be done in such a way that if a new element \( e \) of \( D \) is included in \( S \), the values on which \( F \cdot e \) depends through the recursive definition of \( F \) are already in \( S \).

In fact, the fulfillment of the subgoal \( X \in S \) is not necessarily trivial, and occasionally, a plan must be devised to eventually achieve this membership. Such a plan consists on the studied traversal of a certain path conformed by intermediate calculations of \( F \) on values of \( D \) leading to the calculation of \( F \cdot X \). This planning could be identified with the tabulation techniques characterizing the dynamic programming methods (cf. [Aho74], [Hor78]). The strategy of gradually fulfilling the postcondition, stated this way, roughly corresponds to the well known method of replacing a constant by a fresh variable (actually, by a set of fresh variables) ([Gri81], [Dij88], [Coh90],[Kal90]).

As an illustration of this strategy we solve the following well known problem (cf. [Aho74]):

Given a collection \( \{ A_i : 1 \leq i \leq N \} \), of \( N \) integer matrices \( (N > 0) \) with respectively \( d_i \cdot (i-1) \) rows and \( d_i \cdot i \) columns, find the minimum number of integer multiplications required to evaluate the product \( (\prod_{i=1}^{N} A_i) \) if the usual algorithm for matrix multiplication is used.

For \( 1 \leq i \leq j \leq N \) we define

\[
M \cdot i \cdot j = \text{minimum number of integer multiplications required to evaluate the product } (\prod_{k \leq i \leq j} A_k) \text{ if the usual algorithm for multiplying matrices is used.}
\]

A postcondition may be written as

\[
R : r = M \cdot 1 \cdot N
\]

It follows from the definition of \( M \) that
(1) \( M_{i, i} = 0 \), for \( 1 \leq i \leq N \).

In order to obtain a recursive relation for \( M \), we observe that any evaluation method for the product \( (\prod_{k \leq i \leq j} A_k) \) performing a minimum number of integer multiplications is necessarily \textit{factor closed}. That is, any factor calculated by such method of evaluation, in order to obtain the above mentioned product, performs itself a minimum number of integer multiplications. Otherwise, a contradiction is easily deduced. This property is a particular case of the optimality principle as it is called in the dynamic programming literature ([Hor78]). It is precisely this property what will allow us to apply the reduction principle to obtain a recursive relation for \( M \), as we shall see in a moment. Notice also that the usual algorithm to calculate the product of a \( p \times q \) integer matrix and a \( q \times r \) integer matrix requires \( p \times q \times r \) integer multiplications.

The next property follows from the two preceding remarks:

(2) \( M_{i, j} = \left( \min_{k \leq i \leq j} M_{i, k} + M_{(k+1), j} + d_{i-1} \cdot d_k \cdot d_j \right) \)

for \( 1 \leq i < j \leq N \).

Studying the equations (1) and (2) we observe that, in order to calculate \( M_{1, N} \) using these equations, it is necessary to know previously the values \( M_{i, j} \) for every pair \( (i, j) \) in the domain \( 1 \leq i \leq j \leq N \), excepting obviously \( (1, N) \). Moreover, the calculation of \( M_{i, j} \) for any of those \( i \) and \( j \) with \( i < j \) requires the previous knowledge of the values \( M_{i, k} \) and \( M_{(k+1), j} \) for \( i \leq k < j \), as can be better appreciated in the following figure:

From the discussion above, we conclude that we need a function variable \( m(i, j: 1 \leq i \leq j \leq N) \) to keep the intermediate calculations of \( M \). The invariant obtained by the strategy of \textit{gradual fulfillment of the goal} can be more easily described with the next figure:

We define the domain \( D \) as the triangular area determined by the set of integer coordinates \( \{(h, k): 1 \leq h \leq k \leq N\} \), and \( S \) as the set of integer coordinates in the shaded area pictured in the previous figure. The invariant of the repetition solving the problem may be stated as

\[ P: \quad S \subseteq D \land m(h, k: S) = M(h, k: S) \]

which means that for every pair \( (h, k) \) in \( S \), \( m \cdot h \cdot k = M \cdot h \cdot k \) holds.
The plan to attain the fact \((1, N) \in S\) consists in the gradual calculation of \(M\) on the values of \(D\), moving along the columns of the triangle from the diagonal to the top row, and from left to right. The code corresponding to the traversal can be easily obtained by applying a variation of the \textit{generalized linear search paradigm} to which we shall refer later on. We shall not give the details of the derivation.

We arrive to the following program (note that \(S\) is implicitly described by the pair \((i, j)\)):

\[
\begin{array}{l}
[i, j, r : \text{nat}; m : \text{array} (h, k : D) \text{ of nat};
\end{array}
\]
\[
\begin{array}{l}
i, j := 1, 1;
\end{array}
\]
\[
\begin{array}{l}
\{ \text{inv } P : S \subseteq D \wedge m(h, k : S) = M(h, k : S) \}
\end{array}
\]
\[
\begin{array}{l}
do \ j \leq N \rightarrow
\end{array}
\]
\[
\begin{array}{l}
\begin{array}{l}
\begin{array}{l}
\begin{array}{l}
\begin{array}{l}
\begin{array}{l}
\begin{array}{l}
\begin{array}{l}
\begin{array}{l}
\begin{array}{l}
\begin{array}{l}
\begin{array}{l}
\begin{array}{l}
\begin{array}{l}
\begin{array}{l}
\begin{array}{l}
\begin{array}{l}
\begin{array}{l}
\begin{array}{l}
\begin{array}{l}
\begin{array}{l}
\begin{array}{l}
\begin{array}{l}
\begin{array}{l}
\begin{array}{l}
\begin{array}{l}
\begin{array}{l}
\begin{array}{l}
\begin{array}{l}
\begin{array}{l}
\begin{array}{l}
\begin{array}{l}
\begin{array}{l}
\begin{array}{l}
\begin{array}{l}
\begin{array}{l}
\begin{array}{l}
\begin{array}{l}
\begin{array}{l}
\begin{array}{l}
\begin{array}{l}
\begin{array}{l}
\begin{array}{l}
\begin{array}{l}
\begin{array}{l}
\begin{array}{l}
\begin{array}{l}
\begin{array}{l}
\begin{array}{l}
\begin{array}{l}
\begin{array}{l}
\begin{array}{l}
\begin{array}{l}
\begin{array}{l}
\begin{array}{l}
\begin{array}{l}
\begin{array}{l}
\begin{array}{l}
\begin{array}{l}
\begin{array}{l}
\begin{array}{l}
\begin{array}{l}
\begin{array}{l}
\begin{array}{l}
\begin{array}{l}
\begin{array}{l}
\begin{array}{l}
\begin{array}{l}
\begin{array}{l}
\begin{array}{l}
\begin{array}{l}
\begin{array}{l}
\begin{array}{l}
\begin{array}{l}
\begin{array}{l}
\begin{array}{l}
\begin{array}{l}
\begin{array}{l}
\begin{array}{l}
\begin{array}{l}
\begin{array}{l}
\begin{array}{l}
\begin{array}{l}
\begin{array}{l}
\begin{array}{l}
\begin{array}{l}
\begin{array}{l}
\begin{array}{l}
\begin{array}{l}
\begin{array}{l}
\begin{array}{l}
\begin{array}{l}
\begin{array}{l}
\begin{array}{l}
\begin{array}{l}
\begin{array}{l}
\begin{array}{l}
\begin{array}{l}
\begin{array}{l}
\begin{array}{l}
\begin{array}{l}
\begin{array}{l}
\begin{array}{l}
\begin{array}{l}
\begin{array}{l}
\begin{array}{l}
\begin{array}{l}
\begin{array}{l}
\begin{array}{l}
\begin{array}{l}
\begin{array}{l}
\begin{array}{l}
\begin{array}{l}
\begin{array}{l}
\begin{array}{l}
\begin{array}{l}
\begin{array}{l}
\begin{array}{l}
\begin{array}{l}
\begin{array}{l}
\begin{array}{l}
\begin{array}{l}
\begin{array}{l}
\begin{array}{l}
\begin{array}{l}
\begin{array}{l}
\begin{array}{l}
\begin{array}{l}
\begin{array}{l}
\begin{array}{l}
\begin{array}{l}
\begin{array}{l}
\begin{array}{l}
\begin{array}{l}
\begin{array}{l}
\begin{array}{l}
\begin{array}{l}
\begin{array}{l}
\begin{array}{l}
\begin{array}{l}
\begin{array}{l}
\begin{array}{l}
\begin{array}{l}
\begin{array}{l}
\begin{array}{l}
\begin{array}{l}
\begin{array}{l}
\begin{array}{l}
\begin{array}{l}
\begin{array}{l}
\begin{array}{l}
\begin{array}{l}
\begin{array}{l}
\begin{array}{l}
\begin{array}{l}
\begin{array}{l}
\begin{array}{l}
\begin{array}{l}
\begin{array}{l}
\begin{array}{l}
\begin{array}{l}
\begin{array}{l}
\begin{array}{l}
\begin{array}{l}
\begin{array}{l}
\begin{array}{l}
\begin{array}{l}
\begin{array}{l}
\begin{array}{l}
\begin{array}{l}
\begin{array}{l}
\begin{array}{l}
\begin{array}{l}
\begin{array}{l}
\begin{array}{l}
\begin{array}{l}
\begin{array}{l}
\begin{array}{l}
\begin{array}{l}
\begin{array}{l}
\begin{array}{l}
\begin{array}{l}
\begin{array}{l}
\begin{array}{l}
\begin{array}{l}
\begin{array}{l}
\begin{array}{l}
\begin{array}{l}
\begin{array}{l}
\begin{array}{l}
\begin{array}{l}
\begin{array}{l}
\begin{array}{l}
\begin{array}{l}
\begin{array}{l}
\begin{array}{l}
\begin{array}{l}
\begin{array}{l}
\begin{array}{l}
\begin{array}{l}
\begin{array}{l}
\begin{array}{l}
\begin{array}{l}
\begin{array}{l}
\begin{array}{l}
\begin{array}{l}
\begin{array}{l}
\begin{array}{l}
\begin{array}{l}
\begin{array}{l}
\begin{array}{l}
\begin{array}{l}
\begin{array}{l}
\begin{array}{l}
\begin{array}{l}
\begin{array}{l}
\begin{array}{l}
\begin{array}{l}
\begin{array}{l}
\begin{array}{l}
\begin{array}{l}
\begin{array}{l}
\begin{array}{l}
\begin{array}{l}
\begin{array}{l}
\begin{array}{l}
\begin{array}{l}
\begin{array}{l}
\begin{array}{l}
\begin{array}{l}
\begin{array}{l}
\begin{array}{l}
\begin{array}{l}
\begin{array}{l}
\begin{array}{l}
\begin{array}{l}
\begin{array}{l}
\begin{array}{l}
\begin{array}{l}
\begin{array}{l}
\begin{array}{l}
\begin{array}{l}
\begin{array}{l}
\begin{array}{l}
\begin{array}{l}
\begin{array}{l}
\begin{array}{l}
\begin{array}{l}
\begin{array}{l}
\begin{array}{l}
\begin{array}{l}
\begin{array}{l}
\begin{array}{l}
\begin{array}{l}
\begin{array}{l}
\begin{array}{l}
\begin{array}{l}
\begin{array}{l}
\begin{array}{l}
\begin{array}{l}
\begin{array}{l}
\begin{array}{l}
\begin{array}{l}
\begin{array}{l}
\begin{array}{l}
\begin{array}{l}
\begin{array}{l}
\begin{array}{l}
\begin{array}{l}
\begin{array}{l}
\begin{array}{l}
\begin{array}{l}
\begin{array}{l}
\begin{array}{l}
\begin{array}{l}
\begin{array}{l}
\begin{array}{l}
\begin{array}{l}
\begin{array}{l}
\begin{array}{l}
\begin{array}{l}
\begin{array}{l}
\begin{array}{l}
\begin{array}{l}
\begin{array}{l}
\begin{array}{l}
\begin{array}{l}
\begin{array}{l}
\begin{array}{l}
\begin{array}{l}
\begin{array}{l}
\begin{array}{l}
where $\odot$ is a suitable associative (and frequently commutative) operator with identity $e$, $a$ is some value in $E$, $A$ is a finite (and usually ordered) set, $g$ and $h$ are known functions, and $b$ a known predicate. The recursive definition of $F$ suggests a natural application of the reduction principle.

With a postcondition of the form

$$\text{R: } r = F.X$$

an invariant proposed by the \textit{information balance} strategy might be of the form:

$$\text{P: } F.X = r \odot (\odot i: i \in Ag: F.i)$$

$F.X$ corresponds to the (total) information to be obtained, $r$ holds the currently computed information, and the agenda $Ag$ reminds us of the information still to be computed.

Usually, the repetition is initialized by putting $X$ (representing the original goal to be achieved) in the agenda, and assigning $e$ to $r$ (i.e. the explicit information is trivial). The body of the repetition takes off some element from the agenda and accumulates its contribution to the result into $r$. It may be necessary to put some new elements in the agenda in order to maintain the information balance. Finally, we are done when the agenda becomes empty, since then $r = F.X$.

We obtain:

$$[ r: E; Ag: \text{set};$$

$$Ag,r:= \{X\},e;$$

$$\{\text{inv P: } F.X = r \odot (\odot i: i \in Ag: F.i) \}$$

$$\text{do Ag} \neq \emptyset$$

$$\rightarrow \text{ body}$$

$$\text{ od}$$

$$\{\text{R: } r = F.X\}$$

We use the recursive definition of $F$ to change the agenda in the way suggested above. The associativity of $\odot$ permits the aggregation of the upcoming explicit values into the variable $r$.

As an example of the use of this strategy, we shall develop a program to perform a post-order traversal of a tree, storing the values of its nodes in a sequence. We shall use the following notation, borrowed in part from [Coh90].

The sequence of length 0 is called the \textit{empty} sequence and is denoted by $\varepsilon$. We denote a sequence containing one element $x$ by $[x]$. The first element of a sequence $X$, and the remaining subsequence after its deletion are denoted $hd.X$ and $tl.X$ respectively. Concatenation of sequences is denoted by $\cdot$, as in $X \cdot Y$.

To declare variables of the type \textit{binary tree}, denoted by $\text{bintree}$, we shall write expressions like $u: \text{bintree}$. The empty binary tree is denoted by $\Delta$. A non-empty binary tree $u$ is denoted by a triple $(u.l, u.v, u.r)$, where $u.l$ and $u.r$ are binary trees-the left and right subtrees of $u$-and $u.v$ is some value associated with $u$. When $u$ consists of a single node of value $v$, we just write $(u.v)$ instead of $(\Delta, u.v, \Delta)$. The boolean expression $\text{node}.u$ decides whether or not $u$ consists of a single node. The type of $u.v$ comes from the declaration. When irrelevant we omit this type.

We return to the problem of traversing a binary tree in post-order. Given a tree $T$, we define $\text{po}.T$ as the post-order sequence of $T$, by an application of the reduction principle, as

\begin{align*}
(3.1) \quad \text{po}.T &= \varepsilon & \text{if } T = \Delta \\
(3.2) \quad \text{po}.T &= \text{po}.(T.l) - \text{po}.(T.r) - [T.v] & \text{if } T \neq \Delta
\end{align*}

Given a variable $s$ of type $\text{seq}$, a postcondition may be written as

788
Using the information balance strategy, we propose the invariant

\[ P : \text{po}.T = s - (\neg t \in \text{Ag}: \text{po}.t) \]

where, in this case, \( s \) (the explicit information) holds a prefix of the sequence \( \text{po}.T \), and \( \text{Ag} \) (the implicit information) is a sequence of binary subtrees of \( T \) whose post-order sequence remains to be evaluated. The concatenation of \( s \) with the post-order sequences of the subtrees in \( \text{Ag} \) gives the sequence \( \text{po}.T \) (the total information).

Each iteration of the repetition will then process the first tree in the agenda. First, it is deleted from the agenda. Then, if it is empty, we do nothing; if it consists of a single node, the value of the node is post-concatenated to \( s \); in any other case, its post-order is given by (3.2), and the agenda \( \text{Ag} \) is changed accordingly:

\[
\begin{align*}
\text{T} & : \text{bintree}; \\
\text{s} & : \text{seq}; \text{Ag} : \text{seq of bintree}; \text{tr} : \text{tree}; \\
\text{Ag}, s & : = [T], \epsilon; \\
\{\text{inv } P : \text{po}.T = s - (\neg t \in \text{Ag}: \text{po}.t) \} \\
\text{do } & \text{Ag} <: \text{s} \rightarrow \\
& \text{tr}, \text{Ag} = \text{hd} \cdot \text{Ag}, \text{tl} \cdot \text{Ag}; \\
& \text{if } \text{tr} = \Delta \rightarrow \text{skip} \\
& \text{[ ] node.tr \rightarrow s = s - [\text{tr}.v]} \\
& \text{[ ] tr = \Delta \wedge \neg \text{node.tr} \rightarrow Ag = [\text{tr}.l] - [\text{tr}.r] - [(\text{tr}.v)] - Ag} \\
& \text{fi} \\
& \text{od} \\
\{ s = \text{po}.T \}
\end{align*}
\]

This program eventually terminates, essentially because \( \text{po} \) is a well defined function. We shall not give a formal argument to prove this.

A special case for the application of the information balance strategy occurs when only a single element agenda is needed. This case corresponds to the tail recursion methods to propose invariants as found in [Kal90].

4 REDUCING THE UNCERTAINTY

The strategy of reducing the uncertainty will be useful for the exploration of search spaces, where problems are reformulated in terms of the (possibly partial) traversal of a search space with the purpose of finding, counting, collecting or affecting elements in that space, fulfilling a certain given property.

The idea is to progress through the repetition by reducing the "uncertainty area", i.e. where the searched elements or solutions are confined. The strategy is naturally explained in terms of the reduction principle, since its application amounts to reduce the original search problem into simpler search problems (i.e. with smaller search spaces).

We may use the previously described strategies to devise two techniques:

- Gradual narrowing of the fence, and
- Balance of explored and unexplored areas.

789
4.1 Gradual narrowing of the fence

Given a search space \( E \), and a boolean function \( p \) on \( E \) such that

\[
(\exists x \in E : \ p(x))
\]

We are interested in developing a program to find an element of \( E \) for which \( p \) holds. Let us define \( ev \) as some element of \( E \) such that \( p.ev \) holds. Then, a postcondition for the problem and an invariant for a repetition to solve it, might respectively be stated as follows:

\[
\begin{align*}
R & : e = ev \\
P & : A \subseteq E \land ev \in A
\end{align*}
\]

where \( e \) is a variable of the same type as the elements of \( E \) and \( A \) is a subspace of \( E \), delimited in some sense by \( e \). We advance towards termination reducing the search area \( A \) (i.e. moving appropriately \( e \)), and eventually \( ev \) will be found. The searched for element is every time more restricted into a smaller area, and so this solution technique is called \textit{gradual narrowing of the fence}.

However, there is an inconvenience with the proposed invariant: the element \( ev \) is unknown, and therefore it cannot be used in the code of the developed repetition. We should then be careful when developing the program, and either rephrase conditions mentioning \( ev \), or look for an assertion \( P' \) implying \( P \) and not mentioning \( ev \), to use it as an invariant instead of \( P \) to develop the repetition. In any case, we may arrive to something like:

\[
\begin{array}{l}
\{\exists x \in E : p(x)\} \\
[ e : E; A : \text{set}; \\
\quad A := E; \\
\quad \{\text{inv } P : A \subseteq E \land ev \in A\} \\
\quad \text{do } "e \neq ev" \rightarrow \\
\quad \quad "\text{reduce } A" \\
\quad \text{od} \\
\quad \{R : e = ev\}
\}
\end{array}
\]

We illustrate the use of this technique designing a program to solve a generalized version of the linear search problem, which can be state as follows:

Let \( I \) be an initial segment of the natural numbers. Consider a linearly ordered search space \((E, <)\) where \( E = \{e_i : i \in I\} \), and the order relation \(<\) is defined so that \( e_i < e_j \) when \( i < j \). There is an operator \( \text{suc} \) on \( E \), such that \( \text{suc} . e_k = e_{k+1} \) for \( k \in I \) (except its last element if \( I \) is finite). In addition, there is a boolean function \( p \) defined on \( E \), and an element \( e \in E \) such that \( p.e \) holds. We are asked to find such a value.

We define \( ev = (\min k : k \in I \land p.e_k \land e_k) \), and try to find it by means of a linear search. The resulting program is a \textit{generalized linear search}. In terms of the \textit{narrowing of the fence} technique, the uncertainty area \( A \) will be \( \{e \in E : e \leq e\} \).

We arrive to the following solution (note that \( A \) may be omitted from the code, since it is determined by \( e \)):

\[
\begin{array}{l}
\{\exists x \in E : p(x)\} \\
[ e : E; \\
\quad e := e_0; \\
\quad \text{do } \neg p.e \rightarrow e := \text{suc}.e \text{ od} \\
\quad \{e = ev\}
\]
\]
A variation of this paradigm was used in the first program developed in this article, to traverse and affect the elements of the upper triangular half of a square matrix, determined by its main diagonal. In this case, the invariant should state that the "still unaffected elements" of this space are confined to a (gradually narrowing) subspace of it.

In a similar way, the technique of gradual narrowing of the fence is used to develop the saddleback search paradigm ([Kal90]).

Sometimes the generalized linear search paradigm may be used in unexpected situations. For instance, consider the problem of searching a common element in two infinite arrays \( f(k: \text{nat}) \) and \( g(k: \text{nat}) \). We know that such an element does exist, but the arrays are not necessarily ordered.

Then we can use the paradigm with the following definitions:

- **Search space:** \( E = \text{nat} \times \text{nat} \).
- **First element:** \( e_0 = (0, 0) \).
- **Successor function:** \( \text{suc.}(i, j) = (i-1, j+1) \quad \text{if} \quad i>0 \)
  \( (j+1, 0) \quad \text{if} \quad i=0 \)
- **Search predicate:** \( p.(i, j) = f.i = g.j \)

which are easily understood with the next figure (that reminds a dovetailing counting):

![Dovetailing Counting Diagram]

The resulting code is:

```plaintext
[ f, g: array (k: nat); {f \& g \neq \emptyset} ]
[ i, j: nat; x; i, j:= 0, 0;
do f.i \neq g.j \rightarrow 
  if i>0 \rightarrow i, j:= i-1, j+1
    i=0 \rightarrow i, j:= j+1, 0 
  fi
od;
x:= f.i
{x \in f \& g}
]
```

4.2 Balance of explored and unexplored areas

In contrast with the gradual narrowing of the fence technique, which assumes the existence of the searched element, the technique of balance of explored and unexplored areas allows to count, collect or determine the existence of elements of a search space fulfilling a certain given property. This is done by proposing an invariant stating the current information balance between the explored and unexplored areas of the search space.

A postcondition to determine the existence of a given element \( X \) in a search space \( E \) may be stated as follows:

\[ R: \ x = X \in E \]
where \( r \) is a boolean variable stating the membership of \( X \) to \( E \). We propose the invariant (\( U \) represents a subspace of \( E \))

\[
P: \quad U \subseteq E \land (X \in E = r \lor X \in U)
\]

which states that the membership of \( X \) to \( E \) is determined by either the value of the boolean variable \( r \) or the membership of \( X \) in \( U \). Therefore, \( r \) holds the truth of the fact of having found the element \( X \) in the subspace \( E \setminus U \). Observe that postcondition \( R \) follows from \( P \) and the fact that, either subspace \( U \) is empty, or \( X \) is found in subspace \( E \setminus U \). Notice also that the expression \( P(r, U := \text{false}, E) \) is tautological. We thus obtain:

\[
\text{[ r: bool; U: set;}
\]
\[
r, U := \text{false}, E;
\]
\[
\{ \text{inv } P: \quad U \subseteq E \land (X \in E = r \lor X \in U) \}
\]
\[
do \quad U := \emptyset \land \neg r
\]
\[
\quad \rightarrow \text{body}
\]
\[
od
\]
\[
\{ r = X \in E \}
\]
\[
]

We make progress towards termination reducing \( U \), which is done searching for \( X \). Eventually \( r \) becomes true (i.e., \( X \) is found) or \( U \) becomes empty.

The next example illustrates the use of the balance of explored and unexplored areas technique. It is related to the bounded linear search, and is stated as follows:

Given \( N \geq 0 \) and a boolean array \( b(i: 0 \leq i < N) \), we are asked to determine if there exists a smallest \( x \) in \([0..N-1]\), for which \( b \cdot x \) holds. The answer should be given through the boolean variable \( r \).

A postcondition for this problem may be written as

\[
R: \quad r = (\exists i: 0 \leq i < N: b \cdot i)
\]

Introducing a new integer variable \( x \), and denoting the expression \( (\exists i: x \leq i < N: b \cdot i) \) by \( p \cdot x \), we may use the balance of explored and unexplored areas technique to obtain an invariant \( P \) as follows:

\[
P: \quad 0 \leq x \leq N \land (p \cdot 0 = r \lor p \cdot x)
\]

Here, the search space \( E \) corresponds to the interval \([0..N-1]\) and the subspace \( U \) to \([x..N-1]\). The first conjunct of \( P \) simply presents the variable \( x \). This leads to the following program:

\[
[ \text{N: int; b: array (i:0\leq i < N);}
\]
\[
[ \text{r: bool; x: int;}
\]
\[
r, x := \text{false, 0;}
\]
\[
\{ \text{inv } P: \quad 0 \leq x \leq N \land (p \cdot 0 = r \lor p \cdot x) \}
\]
\[
do \quad x := N \land \neg r
\]
\[
\quad \rightarrow \text{if } b \cdot x \rightarrow r := \text{true}
\]
\[
\quad [] \quad \neg b \cdot x \rightarrow x := x + 1
\]
\[
\text{fi}
\]
\[
od
\]
\[
\{ r = p \cdot 0 \}
\]
\[
]

792
The balance of explored and unexplored areas technique is not only useful to develop programs to determine the existence of a given element in a search space; it may also be used to count or collect elements fulfilling a certain given property. For instance, a postcondition to count those elements of a search space $E$ for which the property $p$ holds, may be expressed as

$$R: \quad r = (\#i: i \in E : p.i)$$

where $r$ is an integer variable. Then the technique inspires an invariant like ($U$ is a subspace of $E$):

$$P: \quad U \subseteq E \land (\#i: i \in E : p.i) = r + (\#i: i \in U : p.i)$$

$P$ states that the number of elements in $E$ fulfilling property $p$ is equal to the number already calculated in $r$, plus the amount of members of $U$ for which $p$ holds.

5 CONCLUSIONS

We hope to have shown that the use of problem solving strategies constitutes a useful and interesting complementary approach to the calculational style for the derivation of programs. The novelty of this approach is based on the consideration of the meaning of the formulas involved in the program specification to guide the derivation, inspired on problem solving strategies.

The reduction principle was found to be at the core of the design of solutions requiring the use of repetitions. Gradual fulfillment of the goal, information balance and reducing the uncertainty were presented as general invariant design strategies based on this principle. We suggested domains or cases where each strategy could be applied, and gave examples to illustrate their use. Indeed, a lot of problems may be stated in the frame of these strategies, and some of the classical techniques may be also rephrased and explained (e.g. dynamic programming, changing constants by fresh variables) in terms of the mentioned strategies.

The notion of implicit information introduced in the information balance strategy seems to be important for the design of invariants. For instance, the strengthening invariants technique presented in [Kal90] corresponds to a decision to make explicit a fact which cannot be easily expressed in terms of the program variables. This is done by the introduction of a fresh variable and an invariant stating its equality with such expression.

The gradual fulfillment of the goal and the information balance strategies rely on the use of a recursive formulation of the problem. It may be said that the essential difference between them lies on the fact that the former uses the recursive relations from "right to left", and the latter from "left to right".

The main strategy to solve search problems is to reduce the area where the solutions are confined. The reducing the uncertainty strategy is naturally induced by the reduction principle. We showed two variants of this strategy, gradual narrowing of the fence and balance of explored and unexplored areas, paying special attention to examples concerning linear searches.

The derived examples for the reducing of the uncertainty strategy are themselves so general that they could be used paradigmatically as program schemes; in this case, programming reduces to identify the parameterized elements in the schemes, and expand -as a macro- the corresponding code. As a matter of fact, we believe that the conscious use of this last programming method might explain the ad-hoc techniques that traditional programmers practice, sometimes with unquestionable success.

Some of the ideas presented in this paper appeared already in [Car93] with a different organization.

Bibliography


