Annales Univ. Sci. Budapest., Sect. Comp. 17 (1998) 155-172

JOINING PROGRAMMING THEOREMS A PRACTICAL APPROACH TO PROGRAM BUILDING

É. Harangozó, P. Szlávi and L. Zsakó
(Budapest, Hungary)

Abstract. We sketch a method of program building which allows a solution to be found, even for complex problems, that is known to be correct.

The method is based on (1) mathematical tools used in algebraic-predicate calculus; (2) abstract elements of programming theorems and (3) some elementary program transformations. We briefly summarize these basic elements in order to demonstrate the program-building method.

In the program model we describe what we regard as the task and the task specification and highlight those features of the sequences which we use in later sections. We employ the well-known approach of using the combination of commonly-occurring tasks and their specifications as programming theorems. By expressing each theorem as a function with a particular domain and range, we obtain an excellent tool for combining elements to yield new theorems. Building programming theorems mechanically on each other in this way often results in an extremely complex program structure, but this can be greatly simplified by automatic transformations which we give as program transformations.

Our program-building procedure is as follows:

First we specify the basic task. For a programmer with some experience, it is quite easy to read from the specification which elementary theorems should be applied in what sequence. Since the application of a theorem means the execution of a function, the joining of theorems is a substitution of the range of one function (the result of the theorem) into the domain of another. This gives rise to the second step, the decomposition of the original task into appropriate subtasks. The subtasks are then connected as determined by one of the theorems or structural features of data. This step occasionally involves application of program transformation. The third step is to deal with each of these elementary theorems individually. This paper demonstrates how to join two theorems together.

1. Introduction

1.1. Programming Model

In a top-down approach we design the solution using three forms of program structure, namely **sequential composition** of (two) instructions, **conditional branch** (selection) and **repetitive execution** of the instruction (loop). The choice depends on the specification of the task.

The task always refers to some group of data. One element of this group is defined by known data (input) and the other by goal data (output). The direct product of the value set of this data group is called **state space** (SS).[1] Each point of this space represents a snapshot of the progress of the program and is called a state, with coordinates known as variables. (Some of these state-coordinates may turn out to have no elementary value.) The variables are in effect projections of the SS.

Example

SS :	$A = BxCx...xH$
state of SS:	$a = (b, c, ..., h)$ where $b \in B$, $c \in C, ..., h \in H$
variable of SS:	$\beta : A \rightarrow B$ projection

A **task** means a relation (F) which is part of the set of the state-pairs ($F \subseteq AxA$). So the first thing is to establish the SS and the possible relations between its coordinates. Then we determine which point of the SS the program starts from. This set can be termed as precondition, and the result set as the postcondition. We give both parts of the set using the customary logical formulae.

[1] The SS generally grows beyond our initial conception as the task proceeds - usually in every refining step.

• We summarize the applied mathematical concepts and notations as follows:

N_0	- 0,1,2,... (set of positive integer numbers and zero)
H	- any *set* (with set operations)
H^N	- set of sequences of H having N elements
H^* is *set* of *finite sequences* of H	- $H^* = \bigcup_{k=0}^{\infty} H^k$
L	- set of logical values: $\{\uparrow, \downarrow\}$
H is *ordered set*	- there exists an order relation on H

In what follows we do not use the concepts of sequences and sets in their general sense but expect them to meet the following criteria:

1. Both set H and sequence S are part of A^*, so both of them consist of the elements of A.
2. The elements of set H may not be repeated, although repetition may of course occur in a sequence. We make use of the set operations intersection ($\cap$), union ($\cup$), subtraction ($\setminus$), inclusion ($\subseteq$) and membership ($\in$) in their usual senses. (We do not define them here.)
3. The elements of sequence S are unambiguously arranged with the aid of the *indexing* operation. s_i means the i-th element of sequence S. Since all the programming theorems (PTh's) operate with one or more sequences, we present some further sequence operations and features.

Concept:	***Its definition:***
$S \in H^*$ is *ordered sequence*	**$\exists \leq$: HxH→L total order relation and $\forall$ i (1≤i<N): $\leq(s_i, s_{i+1}) = \uparrow$ (with conventional notation: $s_i \leq s_{i+1}$)**
	$IsOrderedSequence(X) := \begin{cases} True & \forall i,j: i<j \Rightarrow x_i \leq x_j \\ False & otherwise \end{cases}$

• input and output

It is not possible to predict the full SS of the task from its 0specification, but only some of its projections - those dimensions that are important for us as problem solvers. The structure of the SS will come to light when we engage in solving the task.

It is useful to observe some rules when we specify the input and output: to use suggestive names for the range set of all data types; and to decompose data types into some "standard" basis sets, namely the *logical* (**L**), the *integer* (Z) the *non-negative integer* (N_0), the *string* (**String**) and the set of the *"abstract"* *enumerated* types that are created by us. We do this decomposing by means of *direct product, union, intersection* and *iteration* operations.

By virtue of observing the above rule we can easily join the algorithmic types with the concrete types in the programming language. In the programming language the pairs of decomposition tools given in the previous paragraph are the *record*, the *variant record (record with variant part)* and some sequence-like structure (e.g. array, file, etc.), respectively.[2]

- pre- and postcondition (its aim and connection with the algorithm)

As we have seen the pre- and postcondition fix the task and therefore, more or less unambiguously, fix the program, too. We would expect the algorithm of the adequate program to be easy to create from the previous two assertions. Let us examine how far our expectations may be realized. We usually link the precondition with a detail of the program whose aim is to check whether the precondition is satisfied by the input data. (Since this is normally simple to do, we will not deal with it here.) However, neither extracting information from the postcondition nor transforming it into an appropriate algorithm is a simple problem. We want to show that there are also features of the postcondition from which traces of some macroscopic algorithms (those of the programming theorems) are detectable. Having identified these PTh's in the specification, we only have to join them into a single, suitable algorithm.

1.2. Concepts of programming theorems

We use the term "PTh" in the customary sense [1,5,6] of an assertion: the **algorithm** is correct with respect to the **task** we have specified. In other words, it is a solution of the specified task. So let us denote input of the task by x and output of the task by y, and its pre- and post-conditions by $pre(x)$ and $post(x,y)$ respectively. We say that a program P, implementing the algorithm, is a solution of the task if

$$\forall x : \; pre(x) \overset{P}{\Rightarrow} post(x,y).$$

(Here we apply the notation of first-order predicate calculus.)[3]

The aim of using PTh's is to become able to assemble algorithms from macroscopic elements which are undoubtedly correct. Since PTh's contain the

[2] In this paper the types of the sequences do not play a significant role and so we will assume a suitable **array** of maximum size.

[3] We employed the usual notation [2]: $q(x) \overset{u(x)}{\Rightarrow} p(x')$, in other words this means that x holds the predicate $q(x)$ before executing the program $u(x)$, then x' holds the predicate $p(x')$ after executing the program.

algorithm of the solution as well as the specification, we need do no more than compose and join these "algorithm patterns" properly.

If we consider PTh's as abstract functions mapped from the subset of input data into the subset of output data (**ProgrammingTheorem**: {Input} $\rightarrow$ {Output}), then it will be possible for us to read certain obvious relationships immediately.

1.3. Programming theorems

In what follows we will list only those PTh's that are central to our purpose. To each of them is assigned a unique identifier that simplifies subsequent references to them. (For example, the notation *ptC* means the programming theorem Copying.) [4]

Firstly, we describe PTh, as a function with its *domain* and *range*. Then we specify it adding *input, output, pre-* and *postcondition*. Finally, we add an algorithm which satisfies the specification. The algorithm is defined by Pascal-like procedure, in which input and output of the task appear as parameters, but we do not write the function-like input data in the header of the procedure. Note: we will not deal with the proof of PTh's in this section.

Remark. In the algorithms we use

1. *three type definitions:*

```
H_Seq_Type = Array [1..MaxN] of H_E_Type;
G_Seq_Type = Array [1..MaxN] of G_E_Type;
Word_Seq_Type = Array [1..MaxN] of Word;
Bool_Func_Type = Function (H_E_Type):  Boolean;
H_G_Func_Type = Function (H_E_Type):  G_E_Type;
G_H_G_Func_Type = Function (G_E_Type,H_E_Type):G_E_Type;
E_G_E_Func_Type = Function (E_E_Type,G_E_Type):E_E_Type;
H_Ord_Op_Type = Infix Operator (H_E_Type,H_E_Type):  Boolean;
```

where MaxN is a predefined integer constant, and Word is a type of non-negative integers, furthermore H_E_Type, G_E_Type and E_E_Type are any types (according to set H, G and E respectively);

2. *and some predefined local variable, e.g.*

```
Var I: Word;
```

$ptC: \ Copying(N_0, H^*, H \rightarrow G) : G^*$

`Input`	$N \in \mathbf{N}_0,\ X \in H^*,\ f : H \to G$
`Output`	$Y \in G^*$
`PreCond.`	-
`PostCond.`	$\forall i \in [1, N] :\ y_i = f(x_i)$

Algorithm:

```
Procedure Copying(N:Word; X:H_Seq_Type; f:H_G_Func_Type;
                  Var Y: G_Seq_Type);
Begin
   For I:=1 to N do Y[I]:=f(X[I]);
End;
```

ptSC: SequentialComputing$(N_0, H^*, G, GxH \to G) : G$[4]

`Input`	: $N \in \mathbf{N}_0,\ X \in H^*,\ F : H^* \to G$ [5]
`Output`	: $S \in G$
`PreCond.`	: $\exists F_0 \in G \wedge \exists f : GxH \to G \wedge$ $F(x_1, ..., x_N) = f(F(x_1, ..., x_{N-1}), x_N) \wedge F() = F_0$
`PostCond.`	: $S = F(x_1, ..., x_N)$

Algorithm:

```
Procedure SequentialComputing(N:Word; X:H_Seq_Type;
                  F0:G_E_Type; f:G_H_G_Func_Type;
                  Var S:G_E_Type);
Begin
   S:=F0;
   For I:=1 to N do S:=f(S,X[I]);
End;
```

ptLS: LinearSeeking($\mathbf{N}_0, H^*, H \to L) : (L, \mathbf{N}_0)$

`Input`	: $N \in \mathbf{N}_0, X \in H^*, T : H \to \mathbf{L}$
`Output`	: `Exist`$\in \mathbf{L}$, `Number` $\in \mathbf{N}_0$
`PreCond.`	: -
`PostCond.`	: `Exist`$\equiv (\exists i \in [1, N] : T(x_i)) \wedge$ `Exist`$\Rightarrow$ `Number`$\in [1, N] \wedge T(x_{Number})$

Algorithm:

[4] A special case of this theorem is the well-known theorem Sum.

[5] From the point of view of thhe algorithm the function F, or rather the function f annd the constant F_0 represennting F in the precondition, is considered "a priori input".

```
Procedure LinearSeeking(N:Word; X:H_Seq_Type; T:Bool_Func_Type;
                 Var Exist:Boolean; Var Number:Word);
Begin
      I:=1;
      While (I<=N) and not (T(X[I])) do I:=Succ(I);
      Exist:=(I<=N);
      If Exist then Number:=I
End;
```

ptCt: Counting($\mathbf{N}_0, H^*, H \to L$) : $\mathbf{N}_0$

Input : $N \in \mathbf{N}_0, X \in H^*, T : H \to \mathbf{L}, \chi : \mathbf{L} \to \{0,1\}$,
$\chi(x) = \begin{cases} 1 & \textit{if } x = \textit{True} \\ 0 & \textit{if } x = \textit{False} \end{cases}$

Output : HowMany$\in \mathbf{N}_0$
Precond. : -
PostCond. : HowMany$= \sum_{i=1}^{N} \chi(T(x_i))$

Algorithm:

```
Procedure Counting(N:Word; X:H_Seq_Type; T:Bool_Func_Type;
                 Var HowMany:Word);
Begin
      HowMany:=0;
      For I:=1 to N do If T(X[I]) then HowMany:=HowMany+1;
End;
```

PtMxS: MaximumSearching($\mathbf{N}_0, H^*, HxH \to L$) : $\mathbf{N}_0$ [6]

Input : $N \in \mathbf{N}_0, X \in H^*, <: HxH \to \mathbf{L}$
Output : Max $\in \mathbf{N}_0$
PreCond. : $N > 0 \wedge$ IsOrderedSet(H)
PostCond. : Max $\in [1, N] \wedge \forall i \in [1, N] : x_i \le x_{Max}$

Algorithm:

```
Procedure MaximumSearching(N:Word; X:H_Seq_Type;
                 <:H_Ord_Op_Type; Var Max:Word);
Begin
    Max:=1;
```

6 *This theorem is referred to as "searching for the maximum value".*

```
    For I:=2 to N do If X[Max]<X[I] then Max:=I;
End;
```

1.4. Program transformations

The purpose of program transformations [3,4] is to simplify a program structure and eliminate redundancy and inefficiency which arises from mechanical derivation of the program. The original program and the result of the transformation are semantically equivalent, either without conditions or under conditions which we shall indicate. Two programs given as algorithms are equivalent if both of them realize the same mapping. [2]

The equivalence of the program pair is formally provable. (We shall not discuss this further here because it does not belong to our main theme.) We will make use of them, however, and refer to them by symbols PTx.

We do not aspire to completeness and content ourselves to listing a few program transformations as facts, without proofs. All the five statements below have a simple structure. They begin as follows:

"Statement: a program **P1** equivalent to a program **P2**."

followed by both programs. For brevity we summarize five program transformations in the table on the following page. [7]

2. Joining programming theorems

It often happens that we have to use PTh's sequentially. However by joining PTh's, the program generally becomes much simpler, more compact and more effective than the original, derived mechanically from the specification. In this work we investigate the joining of two PTh's. They will be grouped by a dominant theorem. [4]

[7] We state here that "x is the only input parameter in a(x)".

<table>
<tr><th>P1</th><th></th><th>P2</th></tr>
<tr><td colspan="3">PT1: without any conditions</td></tr>
<tr><td>x:=k; y:=l(x);
While π(x) do
Begin
  y:=g(x,y); x:=i(x);
End;
x:=k; z:=m(x);
While π(x) do
Begin
  z:=h(x,z); x:=i(x);
End;</td><td>⇔</td><td>x:=k; y:=l(x); z:=m(x);
While π(x) do
Begin
  y:=g(x,y);
  z:=h(x,z); x:=i(x);
End;</td></tr>
<tr><td colspan="3">PT2: without any conditions</td></tr>
<tr><td>y:=f(x); z:=g(y)</td><td>⇔</td><td>z:=g(f(x))</td></tr>
<tr><td colspan="3">PT3: with followig condition: $\forall$ p predicate: $p(x) \overset{s:=a(x)}{\Rightarrow} p(x') \wedge p(s) \overset{P1(s)}{\Rightarrow} p(s')$ [7]</td></tr>
<tr><td>If a(x)<b then c:=a(x);</td><td>⇔</td><td>s:=a(x); If s<b then c:=s;</td></tr>
<tr><td colspan="3">PT4: without any conditions</td></tr>
<tr><td>While T(f(x)) do
Begin
  a(x)
End;</td><td>⇔</td><td>ft:=f(x);
While T(ft) do
Begin
  a(x); ft:=f(x);
End;</td></tr>
<tr><td colspan="3">PT5:without any conditions</td></tr>
<tr><td>x:=1;
While (x<=v) and not T(x) do
Begin
  x:=Succ(x)
End;
l:=x<=v;</td><td>⇔</td><td>x:=0; l:=False;
While (x<v) and not l do
Begin
  x:=Succ(x); l:=T(x);
End;</td></tr>
</table>

To understand what this means, consider the general steps that have to be taken from the specification to the finished program.

0. The task is given by its specification: input, output, pre- and postcondition.

1. To decompose the postcondition into a set of postconditions of suitable PTh's.

2. To prove the correctness of the decomposition.

3. To apply theorems by their own subspecification, independently from each other.

4. To simplify the resulting program using program transformations.

2.1. Joining with theorem copying

Any PTh's can be built by **ptC** (Copying), since all that is required is to replace original input data x_i by $g(x_i)$.

For example, if we need to take the sum of squares of a number sequence, the program consists of two parts, the first coming from $ptC(\forall i : x_i \to x_i^2)$, and the second one from $ptSC$ (sequential computing: $(y_1, ..., y_n) \to \sum y_i$). Let us look at a combination of two theorems in general.

```
Input     : N ∈ N0, X ∈ H*, F : G* → E, g : H → G
Output    : S ∈ E
PreCond.  : ∃F0 ∈ E ∧ ∃f : ExG → E∧
            ∀Y ∈ G* ∧ F(y1, ..., yN) = f(F(y1, ..., yN-1), yN)∧
            F() = F0
PostCond. : S = F(g(x1), ..., g(xN))
```

Input : $N \in \mathbf{N}_0, X \in H^*, F : G^* \to E, g : H \to G$

Output : $S \in E$

PreCond. : $\exists F_0 \in E \wedge \exists f : ExG \to E \wedge$
$\forall Y \in G^* \wedge F(y_1, ..., y_N) = f(F(y_1, ..., y_{N-1}), y_N) \wedge$
$F() = F_0$

PostCond. : $S = F(g(x_1), ..., g(x_N))$

Deduction:

Statement:

$$\text{PostCond.} \Longleftrightarrow S = F(y_1, ..., y_N) \quad (1)$$

$$\wedge\, y_1 = g(x_1) \wedge y_2 = g(x_2) \wedge \ldots y_N = g(x_N) \quad (2)$$

Proof. The proof is very simple: we only have to substitute $y_i = g(x_i)$ for (1).

The deduction of the algorithm. Relation (1) is the postcondition of **ptSC** concerning the sequence Y, and relation (2) is the postcondition of **ptC** concerning the sequence X. Because the latter operates on the input sequence of the task it is to be applied first. It produces the input sequence Y of **ptSC**. So joining these two programming theorems

S:=*SequentialComputing*(N,*Copying*(N,X,g),F_0,f)

produces the algorithm below:

```
Procedure CopyingWithComputing(N:Word; X:H_Seq_Type;
                      F0:  E_E_Type; f:E_G_E_Func_Type;
                      g:H_G_Func_Type;
                      Var S:E_E_Type);
Begin
   For I:=1 to N do Y[I]:=g(X[I]);
   S:=F0;
   For I:=1 to N do S:=f(S,Y[I]);
End;
```

To simplify the algorithm we transform it with **PT1**:

```
  ...
Begin
   S:=F0;
   For I:=1 to N do
   Begin
      Y[I]:=g(X[I]); S:=f(S,Y[I]);
   End
End;
```

Let us continue simplifying with **PT2**, which is applicable now:

```
Begin
   S:=F0;
   For I:=1 to N do S:=f(S,g(X[I]));
End;
```

As a second example, let us investigate the joining of copying and searching for the maximum value (i.e. maximum-searching). Now, we have to find the value of the biggest element and its index.

Input : $N \in \mathbf{N}_0, X \in H^*, g : H \to G$
Output : $\texttt{Max} \in \mathbf{N}_0$, $\texttt{MaxValue} \in G$
PreCond. : $N \geq 1 \wedge \texttt{IsOrderedSet}(G)$
PostCond. : $\texttt{Max} \in [1, N] \wedge \forall i \in [1, N] : g(x_{Max}) \geq g(x_i) \wedge$
$\texttt{MaxValue} = g(x_{Max})$

Deduction:

$$\text{PostCond.} \Longleftrightarrow \texttt{Max} \in [1, N] \wedge \forall i \in [1, N] : y_{Max} \geq y_i \quad (1)$$
$$\wedge$$
$$y_1 = g(x_1) \wedge y_2 = g(x_2) \wedge \ldots y_N = g(x_N) \quad (2)$$
$$\wedge$$
$$\texttt{MaxValue} = y_{Max} \quad (3)$$

(1) is the postcondition of $ptMxS$ concerning the sequence Y; (2) is the postcondition of ptC concerning the sequence X; and (3) is realizable with a simple assignment statement. Because the second one operates on the input sequence of the task again it is to be applied first. It produces the input sequence Y of $PtMxS$. So joining of these two programming theorems

$$Max := MaximumChoosing(N, Copying(N, X, g), <)$$

produces the algorithm below:

```
Procedure MaximumCopying(N:Word; X:H_Seq_Type;
                  Var Max:Word; Var MaxValue:G_Type);
Begin
   For I:=1 to N do Y[I]:=g(X[I]);
   Max:=1;
   For I:=2 to N do If Y[Max]<Y[I] then Max:=I;
   MaxValue:=Y[Max]
End;
```

Separating the case of I=1 from the first loop, the algorithm is simplifyable with **PT1**:

```
...
Begin
   Y[1]:=g(X[1]); Max:=1;
   For I:=2 to N do
   Begin
     Y[I]:=g(X[I]); If Y[Max]<Y[I] then Max:=I;
   End;
   ...
End;
```

Let us transformate the result solution so that the formula MaxValue=y_{Max}, in the postcondition, is to be invariant. Then we get

```
...
Begin
   Y[1]:=g(X[1]); Max:=1; MaxValue:=Y[Max];
   For I:=2 to N do
   Begin
     Y[I]:=g(X[I]);
     If Y[Max]<Y[I] then Begin Max:=I; MaxValue:=Y[Max] End;
   End;
End;
```

Let us continue to simplify the last algorithm with **PT2**, and - of course - we have dropped the statement Y[1]:=g(X[1]). Moreover, we can drop the multiple reference to X[Max], if we apply **PT3** and **PT2** ("inversely"). So we reach the final version:

```
Begin
    Max:=1; MaxValue:=g(X[Max]);
    For I:=2 to N do
    Begin
      If MaxValue < g(X[I]) then
      Begin Max:=I; MaxValue:=g(X[Max]) End;
    End;
End;
```

2.2 Joining with the theorem counting

The theorem **ptCt** (Counting) is usually combined with seeking[8]-like theorems, e.g. **ptLS**. We can ask the following questions in this case, "Is there at least K pieces of elements of the feature T in the sequence?" or "Which element of the feature T is just K-th in the sequence?", etc.

The most general case is the joining of the theorem counting and the theorem linear seeking. In similar manner we can perform the deriving in the other two cases, since algorithms of both of them are parts of the algorithm of seeking.

Input : $N \in \mathbf{N}_0, X \in H^*, K \in \mathbf{N}_0, T : H \to \mathbf{L}, \chi : \mathbf{L} \to \{0,1\}$,

$$\chi(x) = \begin{cases} 1 & \text{if x=True} \\ 0 & \text{if x=False} \end{cases}$$

Output : $\mathtt{Exist} \in \mathbf{L}$, $\mathtt{Number} \in \mathbf{N}_0$

PreCond. : $K \in [1, N]$

PostCond. : $\mathtt{Exist} = \left(\sum_{i=1}^{N} \chi(T(x_i)) \geq K\right) \wedge$

$\mathtt{Exist} \Rightarrow \mathtt{Number} \in [1, N] \wedge T(x_{Number}) \wedge$

$\sum_{i=1}^{Number} \chi(T(x_i)) = K$

Deduction:

$$\text{PostCond.} \Longleftrightarrow \mathtt{Exist} \equiv (\exists j \in [1, N] : \sum_{i=1}^{j} \chi(T(x_i)) = K) \wedge$$

$$(\mathtt{Exist} \Rightarrow \mathtt{Number} \in [1, N] \wedge T(x_{Number})) \wedge \quad (1)$$

$$\sum_{i=1}^{Number} \chi(T(x_i)) = K \quad (2)$$

[8] Here we refer to some theorems that are widely used, but we did not mention in the Introduction section; e.g. the so-called *deciding* and *choosing theorems*.

We have to prove the following statement:

$$\sum_{i=1}^{N} \chi(T(x_i)) \geq K \Leftrightarrow \exists j \in [1,N] : \sum_{i=1}^{j} \chi(T(x_i)) = K.$$

Both of the formulae are valid if and only if $K \leq N$. Since $\chi(T(x_i)) = 0$ or 1, the above formulae are surely false if $j < K$, and the sum monotonically increases with increasing j. Furthermore, j can exceed K, if it takes K, too, since its value rises at most 1.

The assertion (1) is a variant of the postcondition of *ptLS*, and the assertion (2) reminds us of the postcondition of *ptCt*. (The last theorem has changed: we have replaced *'K'* with *'HowMany'* and *'Number'* with *'N'*.) Firstly, let us concentrate our attention on the application of *ptLS*. We start from the algorithm, as a frame, associated with the theorem. Its third parameter is a function of logical value, where we include a relation. On the one side of the relation is the result of *ptCt*, and on the other side of it is constant K ($Counting(j,X,T) \geq K$). It is obvious that *ptCt* concerns sequence $(x_1, ..., x_j)$ which continually grows. Thus, the algorithm - written by functional notations - :

$$(\text{Exist}, \text{Number}) := LinearSeeking(N, X, j \in [1,N] : Counting(j,X,T) \geq K)$$

produces the algorithm below:

```
Procedure SeekingWithCounting(N:Word; X:H_Seq_Type;
                  T:Bool_Func_Type;
                  Var Exist:Boolean; Var Number:Word);
Begin
   ...
   I:=1;
   While (I<=N) and (Count(I,X,T)<K) do I:=Succ(I);
   Exist:=(I<=N);
   If Exist then Number:=I;
End;
```

where we have to make the algorithm of the function `Count` to (2) which satisfies the postcondition:

$$\texttt{Count(j,X,T)} = \sum_{i=1}^{j} \chi(T(x_i)) \tag{3}$$

```
Function Count(J:Word;X:H_Seq_Type;T:Bool_Func_Type):Word;
Begin
   ...
   HowMany:=0;
   For L:=1 to J do If T(X[L]) then HowMany:=HowMany+1;
   Count:=HowMany;
End;
```

It appears to be difficult to simplify because *ptCt* is in the condition of the loop. T can be helped with *PT*4.

```
Begin
   ...
   I:=1; HowMany:=Count(1,X,T);
   While (I<=N) and (HowMany<K) do
   Begin
     I:=Succ(I); HowMany:=Count(I,X,T)
   End;
   Exist:=(I<=N);
   If Exist then Number:=I;
End;
```

Note that the function Count acquired an extra condition in the last procedure, i.e. it also has to respond to the case of I=N+1. This anomaly comes from the fact that if a logical expression consists of more than one part and its value has been decided on the way (evaluation), then it is unnecessary to continue any calculations, in fact some errors can occur in consequence of it. Therefore the call of function Count(N+1,X,T) did not appear beside I≤N in the first version. Let us solve this problem by means of *PT*5. Now perform some substitutions:

```
'x:=1'       ⇒ 'I:=1; HowMany:=Count(1,X,T)'
'x:=Succ(x)' ⇒ 'I:=Succ(I); HowMany:=Count(I,X,T)'
'T(x)'       ⇒ 'HowMany>=K'
'x:=0'       ⇒ 'I:=0; HowMany:=0'
```

```
Begin
   ...
   I:=0; HowMany:=0; Exist:=False;
   While (I<N) and not Exist do
   Begin
     I:=Succ(I); HowMany:=Count(I,X,T); Exist:=(HowMany>=K)
   End;
```

If `Exist` **then** `Number:=I;`

End;

There is no program transformation to continue simplifying. We have found additional inherent connections. So let us formulate the invariant statement of the top-level loop. In general the invariant assertion of seeking is

$$\forall j \in [1, i-1]: \quad \neg T(x_j),$$

updated this assertion we get

$$\forall j \in [1, i-1]: \quad \neg \texttt{Count}(j, X, T) \geq K.$$

Considering that the body of the loop consists of a single instruction `I:=` `:=Succ(I)`, it is easy to find a connection between `Count(J,X,T)` and `Count(J+1,X,T)`.

Statement:

$$\texttt{Count}(\texttt{j}, \texttt{X}, \texttt{T}) = \texttt{L} \overset{I:=Succ(I)}{\Rightarrow} \texttt{Count}(\texttt{j}+\texttt{1}, \texttt{X}, \texttt{T}) = \texttt{L} \vee \\ \texttt{Count}(\texttt{j}+\texttt{1}, \texttt{X}, \texttt{T}) = \texttt{L}+\texttt{1}$$

Proof. Let us investigate the corollary of the statement above:

$$\texttt{Count}(\texttt{j}+\texttt{1}, \texttt{X}, \texttt{T}) = \texttt{L} \vee \texttt{Count}(\texttt{j}+\texttt{1}, \texttt{X}, \texttt{T}) = \texttt{L}+\texttt{1}. \tag{4}$$

In consequence of the statement (3) and the definition of the function χ

$$(4) \Leftrightarrow (\texttt{Count}(\texttt{j}, \texttt{X}, \texttt{T}) = \texttt{L} \wedge \neg \texttt{T}(\texttt{X}_{j+1})) \vee \\ (\texttt{Count}(\texttt{j}, \texttt{X}, \texttt{T}) = \texttt{L} \wedge \texttt{T}(\texttt{X}_{j+1})).$$

Considering the condition part of the statement above it is equivalent to (identical formula)

$$(\neg \texttt{T}(\texttt{x}_{j+1}) \vee \texttt{T}(\texttt{x}_{j+1})) \Leftrightarrow \texttt{True}.$$

It is advantageous for us, because the function `Count(j,X,T)` is usable without recalculating it.

Now we are allowed to declare that the body of the loop in previous version of algorithm and the following detail of the program are evidently equivalent:

`HowMany:=Count(I,X,T);`

If `T(X[I])` **then** `HowMany:=HowMany+1;`

Finally we can write the algorithm in its final form, in which the function `Count` does not appear:

```
Begin
   ...
   I:=0; HowMany:=0; Exist:=True;
   While (I<N) and Exist do
   Begin
     I:=Succ(I);
     If T(X[I]) then HowMany:=HowMany+1;
     Exist:=(HowMany≥K)
   End;
   If Exist then Number:=I;
   End;
```

3. Conclusions

Lack of space prevents us from listing any more examples but we hope that we have succeeded in showing the essence of the method. As we have seen, it is possible to derive the program with basic programming tools and elementary mathematical formulae. The set of tools for the operation consists of about twenty program theorems and at most ten program transformations. In addition to the tools we have mentioned, the method requires knowledge of first-order predicate calculus notation and the ability to formulate invariate assertions (for the management of loops).

References

[1] **Fóthi Á.**, *Bevezetés a programozáshoz (Introduction into programming)*, Tankönyvkiadó, Bp., 1983.

[2] **Varga L.**, *Programok analízise és szintézise (The analysis and the synthesis of programs)*, Akadémiai Kiadó, 1981.

[3] **Szlávi P. és Zsakó L.**, *Módszeres programozás: Programozási bevezető (Systematic programming: Introduction to programming)*, Eötvös Loránd University, Faculty of Science, Department of General Computer Science, 1993.

[4] **Szlávi P. és Zsakó L.**, *Módszeres programozás: Programozási tételek (Systematic programming: Program theorems)*, Eötvös Loránd University, Faculty of Science, Department of General Computer Science, 1993.

[5] **Dahl O.J., Dijkstra E.W. and Hoare C.A.R.**, *Structured programming,* Academic Press, 1972.

[6] **Dijkstra E.W.**, *A discipline of programming,* Prentice-Hall, Englewood Cliffs, New Jersey, 1976.

É. Harangozó, P. Szlávi and L. Zsakó
Department of General Computer Science
Eötvös Loránd University
VIII. Múzeum krt. 6-8.
H-1088 Budapest, Hungary
papne@ludens.elte.hu, szlavi@ludens.elte.hu, zsako@ludens.elte.hu