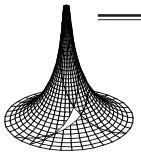


12/2 (2014), 185–199
tmcs@math.klte.hu
http://tmcs.math.klte.hu

Teaching
Mathematics and
Computer Science

Force of summation

Tibor Gregorics

Abstract. Programming theorems are important tools of programming methodology. By using analogous programming techniques, the solutions of different tasks can be created easily and fast based on programming theorems. Perhaps the summation is the simplest programming theorem that is widely-known among the programmers but once and for all the most various tasks can be solved by this theorem. The aim of the present paper is to investigate the summation programming theorem. Several different abstract levels of this theorem will be defined and the problem types that can be solved based on summation are going to be described. We will underline those points of a programming theorem that make a theorem general and that are not defined in advance, just later during its application, when the solution of a problem is derived from the theorem.

Key words and phrases: analogous programming, programming theorem, enumerator.

ZDM Subject Classification: P50.

1. Introduction

Programming theorems are used frequently to plan algorithms. A programming theorem is a pattern, a problem-program (task-algorithm) pair where a program can solve the problem. All the tasks that are similar to the problem of a theorem can be solved on the basis of the algorithm of the theorem. Programming theorems (summation, counting, maximum selection, and linear searching, etc. [1] [4]) are well-known by all programmers but only a few of them know that these theorems can be expressed in multiple ways. Most programmers consider programming theorems as sample solutions. When they want to solve a task that is similar to the problem of a theorem, they try to repeat the same activities that

created the program of the theorem. [2] Thus programming theorems support their algorithmic way of thinking that is used to construct the algorithm of their task.

However, there exists another method to create programs based on programming theorems. This is derivation. [1] [7] Starting from the exact comparison of the task to be solved and of the problem of the candidate programming theorem, the program of the theorem has to be updated according to the differences between the task and the problem. [3] [5] Thus, without algorithmic way of thinking, the program by which the new task is solved can be produced almost automatically. This method is faster and guarantees the correctness of the algorithm but it requires the formal description of the task. The whole of this paper can be articulated from this single point of view, i.e. when algorithms are planned with derivation.

Perhaps the summation is the simplest programming theorem. Both its goal and its algorithm are clear, most programmers use it in their practice. At first glance we cannot think how variously the summation can be applied and how many different problems can be solved with it. For example, we can do summation over the elements of an array, over the elements given by an appropriate function, or over the elements provided by a special activity, an enumeration [4]. The only common characteristic of these usages is that the operator of the summation is defined on these elements.

According to the way that the task of the programming theorem is generalized, different versions of the theorem can be obtained. [6] In this paper three versions of the programming theorem of summation are going to be defined. Many types of tasks that can be solved based on these versions will be investigated, so we will show the force of summation.

2. The three levels of the programming theorem of summation

One of the common features of programming theorems is that they process a sequence of elementary values. The way these values are produced may differ. Summations may be distinguished according to these three levels. The summation operator of all three levels is the mapping $+ : H \times H \to H$ but on the second and the third level a corresponding function can create the elements of the set H from the enumerated values, the elements which must be added to the result.

Since the most widely-known sequence is the one-dimensional array, the first level of the programming theorem of summation in array will be defined. On

this level this array directly provides the elements that must be summed. The summation operator is defined over the elements of this array.

A higher level is when an appropriate function gives the elements that must be processed. The domain of this function is always an interval of integers. (Hereafter $[m \,..\, n]$ denotes the integer interval $[m, n] \cup \mathbb{Z}$.) This function is more universal than an array: each array can be interpreted as a function over integer interval. In fact in this case, the elements of the interval $[m..n]$ are enumerated and a function (f) is given that maps from these elements to the set (H) over which the summation operator is defined.

The third level is when the elements are provided by a special activity, an enumeration. The enumerator is an object that disposes the four enumeration operators: $First()$, $Next()$, $End()$, $Current()$. These operators permit the iteration of the elements that must be processed. The elements of an array can be iterated like the proper divisors of a natural number. A function is needed that maps from these elements to the set (H) over which the summation operator is defined. This point of view gives more universal definitions of programming theorem of summation.

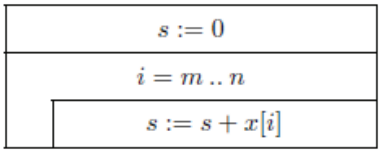
2.1. Summation in array

Problem : An array over an integer interval $m \,..\, n$ is given where its elements are derived from the set H. (Notation of this array is $array([m \,..\, n], H)$). There is also an associative operator $(+ : H \times H \to H)$ with a left-hand zero element. Let us call it as summation operator. (The stucture $(H, +)$ is a monoid.) We need to calculate the sum of the elements of the array.

Specification :

Input	:	$x : array([m \,..\, n], H)$	(input variable)
Output	:	$s : H$	(output variable)
Precondition	:	$(x = x_0)$	(x_0 is an arbitrary array)
Postcondition	:	$(x = x_0 \wedge s = \sum_{i=m}^{n} x[i])$	

Algorithm :



where $i : \mathbb{Z}$ is an auxiliary variable.

2.2. Summation over interval

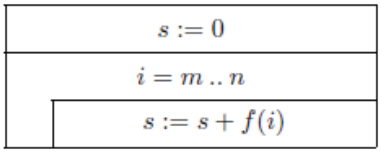
Problem : Let $m..n$ denote an integer interval. A function $f : [m..n] \to H$ where H is an arbitrary set is given. There exists an associative summation operator over the set H with a left-hand zero element $(+ : H \times H \to H)$. We need to calculate the sum of the values created by the function f over the interval $m \,..\, n$. [1] [7]

We remark that in the concrete cases the definition of the function $f : [m \,..\, n] \to H$ must be defined as a part of the specification. However this function is not a sort of input data but an important slice of the relation between the input and output data; this relation is defined in the postcondition.

Specification :

Input	:	$m : \mathbb{Z}, n : \mathbb{Z}$	(input variables)
Output	:	$s : H$	(output variable)
Precondition	:	$(m = m_0 \wedge n = n_0)$	(m_0, n_0 are arbitrary integers)
Postcondition	:	$(m = m_0 \wedge n = n_0 \wedge$	$s = \sum_{i=m}^{n} f(i))$

Algorithm :



where $i : \mathbb{Z}$ is an auxiliary variable.

Often, the computation of the values of this function does not depend on only the elements of the interval $m \,..\, n$ but also on other input data of the problem. However these extra input data are fixed thus they will not become the arguments of the function but the part of its computation only. In other words, according to the concept of Lambda calculus, $f(i) = \lambda i[F(i, data)]$ where the function $F : [m \,..\, n] \times Input \to H$ is given in the concrete problem.

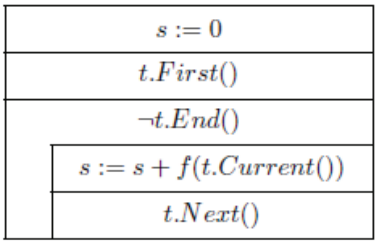
2.3. Summation on enumerator

Problem : An enumerator is given that can iterate the elements of a finite non-empty sequence which belong to the set E ($enor(E)$ denotes the type of this enumerator). A function is given $f : E \to H$ where H is an arbitrary set with an associative summation operator $(+ : H \times H \to H)$ with a left-hand zero element. We need to calculate the sum of the values created by the function f over the elements of the enumeration. [4] [5] [7]

Specification :

Input	:	$t : enor(E)$	(input variable)
Output	:	$s : H$	(output variable)
Precondition	:	$(t = t_0)$	(t_0 is an enumerator)
Postcondition	:	$(t = t_0 \wedge s = \sum_{e \in t_0} f(e))$	

Algorithm :



3. Applications of the summation

Now a number of examples will be shown how the different levels of the summation can be applied in programming.

3.1. Summation tasks with array

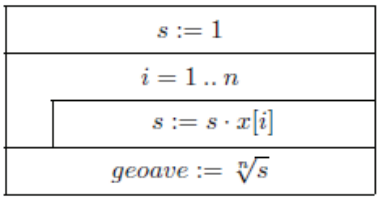
This version of the programming theorem of the summation can be used for example when the sum of an array of real numbers ($\mathbb{R}$) is needed to calculate their average (i.e. arithmetical mean). In this case the summation operator is a simple addition, so this is a traditional usage of the summation.

But this very version helps to compute the geometrical mean of the array, too.

Input	:	$x : array([1 \,..\, n], \mathbb{R})$
Output	:	$geoave : \mathbb{R}$
Precondition	:	$(x = x_0 \wedge even(n) \rightarrow \prod_{i=1}^{n} x[i] \geq 0)$
Postcondition	:	$(x = x_0 \wedge s = \prod_{i=1}^{n} x[i] \wedge geoave = \sqrt[n]{s})$

where $s : \mathbb{R}$ is an auxiliary variable.

However in this case the summation operator must be substituted with the multiplication of real numbers which is associative and its zero element is 1. Now the set H is $\mathbb{R}$ and the interval $m \,..\, n$ is the interval $1 \,..\, n$. This process is the first step of the sequential construct where the second step computes the value of $\sqrt[n]{s}$.



Sometimes the tasks like maximum selection can be solved with the summation programming theorem if the set H is well-ordered with regard to a totally order relation, i. e. all non-empty subsets of H have got a minimum. In this case the operator $max : H \times H \to H$ can be introduced. For all $a, b \in H$

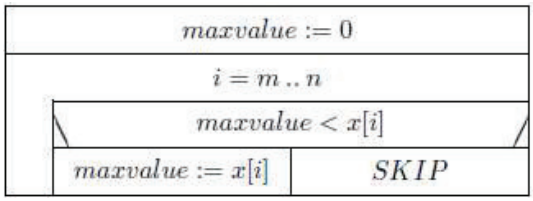
$$max(a,b) = \begin{cases} a & \text{if } a \geq b \\ b & \text{otherwise} \end{cases}$$

This operator is associative and its zero element is the minimum of the set H. For example, if the maximal element of an array including natural numbers ($\mathbb{N}$) must be found, then the specification of this problem is the following:

$$\begin{array}{lcl} Input & : & x : array([m \,..\, n], \mathbb{N}) \\ Output & : & maxvalue : \mathbb{N} \\ Precondition & : & (x = x_0) \\ Postcondition & : & (x = x_0 \wedge maxvalue = \overset{n}{\underset{i=m}{\mathrm{MAX}}}\, x[i]) \end{array}$$

where $\overset{n}{\underset{i=m}{\mathrm{MAX}}}\, x[i] = max(\ldots max(max(x[m], x[m+1]), x[m+2]) \ldots, x[n])$.

The solution of this problem can be derived from the summation so that the variable s is the $maxvalue$, the set H is $\mathbb{N}$, the summation operator is the operator max, its zero element is 0, and instead of the assignment $maxvalue := max(maxvalue, x[i])$ an alternative construct is allowed to be written:



The next task is a typical decision problem. An array including logical values is given and the question is if there exists a true value in that array. The type of the output variable is $\mathbb{L}$ ($\mathbb{L} = \{false, true\}$). In the postcondition of this problem, instead of $s = \exists i \in [m \,..\, n] : x[i]$ the form $s = x[m] \vee \ldots \vee x[n]$ or $s = \vee_{i=m \,..\, n} x[i]$

can be also written. Thus the solution of this problem can also be derived from the summation so that the set H is $\mathbb{L}$, the summation operator is the operator $\vee : \mathbb{L} \times \mathbb{L} \to \mathbb{L}$, and its zero element is $false$. (The solution of the dual task of the this problem, where the question is if there are all values of that array are true, can be solved analogously. In this case the summation operator is the operator $\wedge : \mathbb{L} \times \mathbb{L} \to \mathbb{L}$, and its zero element is $true$.)

We must remark that there exists a better programming theorem that can solve the problem above than the summation. This is the linear searching. However, if two problems must be solved on the same array (one of them a decision like this and the other a simple summation), then it is worth to rally their loops into one loop. This concentration is simple if the structures of both loops are corresponded with each other, namely both of them are counted loops. But the loop of linear searching is not a counted loop. It is more difficult to rally with a counted loop like our decision derived from summation is used.

When the result of a task is a sequence (especially an array), then it must be often connected element by element. The operator concatenation can connect two sequences ($\oplus : E^* \times E^* \to E^*$ where E^* is the set of the finite sequences made of the elements of the set E). This operator is associative and its zero element is the empty sequence. The solutions of all these problems can be derived based on summation. The simplest problem that can be solved this way is copying.

3.2. Summation tasks over interval

The main difference between the first and this version of summation is that the elements which are produced, are not in an array but a function can create them.

A typical usage of this level is the calculation of the factorial. In this task there is no array at all, so the first level of the summation is not suitable, only the second one. The set H of the programming theorem is the set of natural numbers, the interval $m \,..\, n$ is the interval $2 \,..\, n$, the summation operator is the multiplication operator over natural numbers and the function f is the identical mapping.

Another situation is hold if we must count the even elements in an array that contains integers ($\mathbb{Z}$). The function F of the postcondition maps from $[m \,..\, n] \times array([m..n], \mathbb{Z})$ to $\mathbb{N}$ and for all $i \in [m..n]$ and for all $x \in array([m..n], \mathbb{Z})$ gives 1 if $x[i]$ is even, and gives 0 otherwise. The $f(i)$ can be corresponded to $\lambda i[F(i, x)]$. Here another famous programming theorem arises: this is the counting. A more general function can be used if the values of $f(i)$ are not just 1 or 0. The type

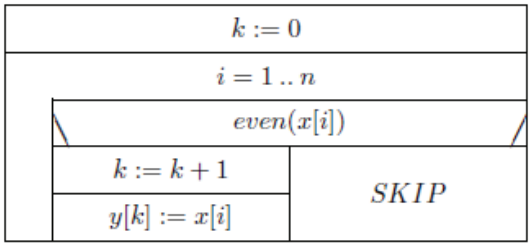
of these tasks can also be called as conditional summation. It may be a separate programming theorem, too.

The first version can process only one element of the array in one step. The function f of the second version allows the process of pairs or triples of elements. For example, only the second version of the summation can be used to solve the problem where the sum of the differences of the neighboring pairs of an array is needed.

Only the second version allows the solution of the problems whose aim is to create one or several sequences except copying of a sequence. These problems (assortments and separations) can be described with the operator concatenation and a function mapping to a sequence. For example, let us solve the problem of the selection of the even integers from a given array.

$$\begin{array}{lcl} \mathit{Input} & : & x : \mathit{array}([1\,..\,n], \mathbb{Z}) \\ \mathit{Output} & : & y : \mathit{array}([1\,..\,n], \mathbb{Z}), k : \mathbb{Z} \\ \mathit{Precondition} & : & (x = x_0) \\ \mathit{Postcondition} & : & (x = x_0 \wedge y[1..k] = \mathop{\Large\oplus}_{i=1}^{n} F(i, x)) \end{array}$$

Here the summation operator is the concatenation ($\oplus : \mathbb{Z}^* \times \mathbb{Z}^* \to \mathbb{Z}^*$).The $y[1..k]$ denotes the first k elements of the array y that is a sequence of integers. (If k is less than 1 then $y[1..k]$ is the empty sequence.) The function F of the postcondition maps from $[1\,..\,n] \times \mathit{array}([1\,..\,n], \mathbb{Z})$ to $\mathbb{Z}^*$ and for all $i \in [1\,..\,n]$ and for all $x \in \mathit{array}([1\,..\,n], \mathbb{Z})$ gives $< x[i] >$ if $x[i]$ is even, and gives $<>$ otherwise. (The $<>$ is the empty sequence.) The $f(i)$ can be corresponded to $\lambda i[F(i, x)]$, the interval $m\,..\,n$ is $2\,..\,n$, and instead of the assignment $s := s \oplus f(i)$ an alternative construct must be written:



where $\mathit{even} : \mathbb{Z} \to \mathbb{L}$ recognizes even numbers.

The second version of the summation is worth being used as well if a composite problem must be solved, for example, when the sum of the maximum values of the rows of a matrix is needed. This problem can be solved with an introduction

of an auxiliary array of course when firstly the maximum values can be stored into this array and secondly the sum of this array can be calculated by the first level of the summation. But a better solution will be got with the second level of the summation. In this case the values of the function f shows the maximum values of the rows of the matrix (the rows are numbered from m to n). When the second level of the summation evaluates the $f(i)$ it will call a maximum selecting procedure to compute this value. This computation of this maximum selecting is embedded in the summation. This solution does not need an auxiliary array storing maximum values and its running time is also better.

Let us look at the following task. There exist n different jobs with their investment costs (inv is an array) and their expected profits ($prof$ is another array). A job can be undertaken if its investment cost less than or equal to our current capital. We have got some initial capital ($init$) and try to undertake the jobs in their given order. If our current capital is large enough to undertake the next job then our capital will be decreased with the investment cost of the very job but increased with its profit. How much is the total investment cost of the jobs that can be undertaken?

$$\begin{array}{lcl} Input & : & inv : array([1\,..\,n], \mathbb{N}), prof : array([1\,..\,n], \mathbb{N}), init : \mathbb{N} \\ Output & : & total : \mathbb{N} \\ Precondition & : & (inv = inv_0 \wedge prof = prof_0 \wedge init = init_0) \\ Postcondition & : & (Precondition \wedge total = \sum_{i=1}^{n} cost(i, inv, prof)) \end{array}$$

This specification is based on two functions:

$$cost : [1\,..\,n] \times array([1\,..\,n], \mathbb{N}) \times array([1\,..\,n], \mathbb{N}) \to \mathbb{N}$$

$$cost(i, inv, prof) = \begin{cases} inv[i] & \text{if } inv[i] \leq capit(i-1, inv, prof) \\ 0 & \text{otherwise} \end{cases}$$

The $capit(i-1)$ denotes the current capital after the first $i-1$ jobs.

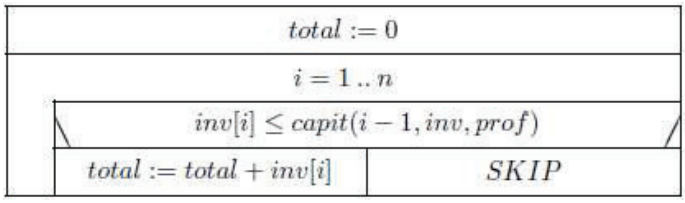
$$capit : [0\,..\,n] \times array([1\,..\,n], \mathbb{N}) \times array([1\,..\,n], \mathbb{N}) \to \mathbb{N}$$

$$capit(0, inv, prof) = init$$

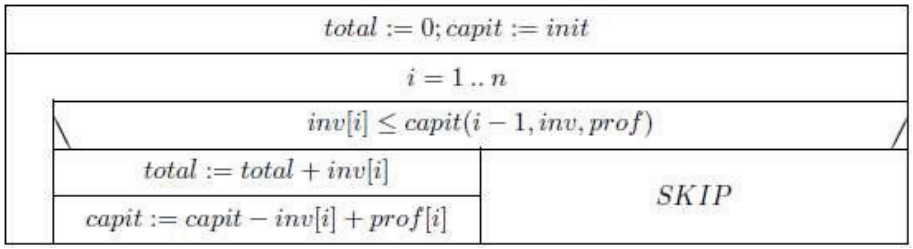
$$capit(i, inv, prof) = \begin{cases} capit(i-1, inv, & prof) - inv[i] + prof[i] \\ & \text{if } inv[i] \leq capit(i-1, inv, prof) \\ capit(i-1, inv, & prof) \\ & \text{otherwise} \end{cases}$$

The solution of this problem can be derived from the summation so that the interval $m\,..\,n$ is $1\,..\,n$, the set H is $\mathbb{N}$, the function f is equal to the function cost, the summation operator is the addition, the variable s is the $total$ and instead

of the assignment $total := total + \lambda i[cost(i, inv, prof)]$ an alternative construct must be written:



The function *capit* was defined recursively. In this case its values can be computed with a simple program transformation since the domain of this function fits the interval of the counted loop of the summation. This transformation introduces the variable *capit* that contains the values that are given by the function *capit*.



3.3. Summation tasks on enumerator

When the elements in a sequential input file or a sequence or another container must be processed, the summation of this third level is needed. Let us, for example, select into a sequential output file the even integers from a sequential input file including integers!

A sequential input file is a special finite sequence on which only one operator can be executed: this is the reading. This operator can pop the first element out of the sequential input file, so the length of the file is decreased. A sequential output file is also a finite sequence which can be initialized as empty sequence and can be extended with a new element.

Input	:	$x : infile(\mathbb{Z})$
Output	:	$y : outfile(\mathbb{Z})$
Precondition	:	$(x = x_0)$
Postcondition	:	$(y = \underset{e \in x_0}{\oplus} f(e))$

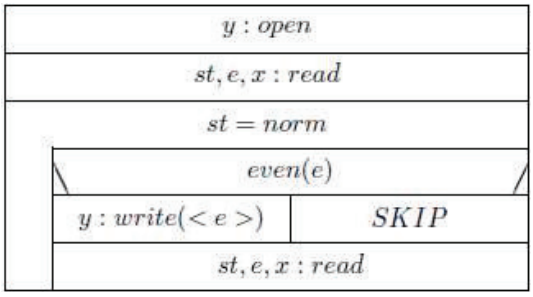
The $e \in x_0$ symbolizes the enumerator that can iterate all elements of x_0. The set E of the programming theorem is equal to $\mathbb{Z}$, the function f of the postcondition maps from $\mathbb{Z}$ to $\mathbb{Z}^*$ and for all $e \in \mathbb{Z}$

$$f(e) = \begin{cases} < e > & \text{if } even(e) \\ <> & \text{otherwise} \end{cases}$$

where $even : \mathbb{Z} \to \mathbb{L}$ recognizes even numbers.

In this solution the enumerator can iterate the elements of the sequential input file. This traversal uses the operator $st, e, x : read$ where x is the sequential input file, e is the element that is read out from the beginning of the file ($e \in \mathbb{Z}$), st is the status of the reading: its value is $norm$ if the reading was successful (x was not empty before reading) otherwise $abnorm$. The operators $First()$ and $Next()$ are equal to this reading, the operator $End()$ gives back the value of expression $st = abnorm$, and the operator $Current()$ gives back the value of e.

The $y : open$ resets the file y as an empty sequence, the $y : write(f(e))$ can concatenate the element $f(e)$ to the end of y. Instead of the assignment $y : write(f(e))$ an alternative construct is written: if e is even, then $y : write(< e >)$, otherwise there is nothing to do.



An enumerator may be very difficult. For example, we can iterate all divisors of a natural number [4], or the balances of banking transactions of clients in a month if these transactions are sorted by account numbers, or the elements of two sequential files so that they run into one sequence if the files are strictly monotone sorted. Based on this last enumeration, the union of the integers of two sequential files can be produced [7]:

$$\begin{array}{lcl} Input & : & x, y : infile(\mathbb{Z}) \\ Output & : & z : outfile(\mathbb{Z}) \\ Precondition & : & (x = x_0 \wedge y = y_0 \wedge x_0 \uparrow \wedge y_0 \uparrow) \\ Postcondition & : & (z = \mathop{\oplus}_{e \in \{x_0\} \cup \{y_0\}} f(e)) \end{array}$$

The $x \uparrow$ denotes that the elements of sequential file x_0 are strictly monotone sorted. The x_0 is the set of the elements of $\{x_0\}$. The $e \in \{x_0\} \cup \{y_0\}$ symbolizes the enumerator that can iterate all elements of x_0 and y_0 but every element can occur in this enumeration only once. The summation operator is the concatenation.

Now the set E of the programming theorem is equal to $\mathbb{Z}$, the function f of the postcondition maps from $\mathbb{Z}$ to $\mathbb{Z}^*$ and for all $e \in \mathbb{Z}$

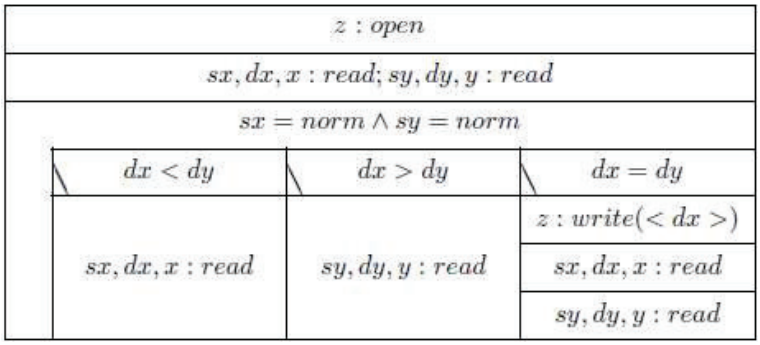
$$f(e) = \begin{cases} <> & \text{if } e \in \{x_0\} \wedge e \notin \{y_0\} \\ <> & \text{if } e \notin \{x_0\} \wedge e \in \{y_0\} \\ < e > & \text{if } e \in \{x_0\} \wedge e \in \{y_0\} \end{cases}$$

The operator $First()$ can be substituted with a reading from x ($sx, dx, x : read$) and a reading from y ($sy, dy, y : read$). In general the operator $End()$ will be *true* if both status of readings (sx and sy) are abnorm but now, because of the concrete task, the $End()$ will be true if either sx or sy is *abnorm*.

<table>
<tr><td colspan="4">$z : open$</td></tr>
<tr><td colspan="4">$sx, dx, x : read; sy, dy, y : read$</td></tr>
<tr><td colspan="4">$sx = norm \wedge sy = norm$</td></tr>
<tr><td></td><td>$dx < dy$</td><td>$dx > dy$</td><td>$dx = dy$</td></tr>
<tr><td></td><td>$dz := dx$</td><td>$dz := dy$</td><td>$dz := dx$</td></tr>
<tr><td></td><td>$dx < dy$</td><td>$dx > dy$</td><td>$dx = dy$</td></tr>
<tr><td></td><td>$SKIP$</td><td>$SKIP$</td><td>$z : write(< dz >)$</td></tr>
<tr><td></td><td>$dx < dy$</td><td>$dx > dy$</td><td>$dx = dy$</td></tr>
<tr><td></td><td rowspan="2">$sx, dx, x : read$</td><td rowspan="2">$sy, dy, y : read$</td><td>$sx, dx, x : read$</td></tr>
<tr><td></td><td></td><td>$sy, dy, y : read$</td></tr>
</table>

The body of the loop in the algorithm above shows several steps: getting the value of $Current()$, computing its f value, writing it into the output file (this is the step $s := s + f(e)$ of the summation), and executing the operator $Next()$. All of them can be implemented by a three branches alternative construct. The operator $Current()$ gives back the less value among two values (dx and dy) which have been read at last because this is the next value of the enumeration. If this element is a common element of x_0 and y_0, then the function f gives back the sequence containing only this element otherwise gives back the empty sequence. The operator $Next()$ reads a new element instead of the one that the operator

Current() gave back. After minor modification the final version of this algorithm is the following:



4. Conclusions

It does not need to be proved that the success (and often efficiency) of a solution based on derivation depends on the degree of the universality of the programming theorems. A good programming theorem should be adequately universal so that the class of the tasks to be solved is wide enough. But the theorem must preserve some specialty in order that it can be identified in a simple way.

All our versions of summation that were presented in this paper correspond to these principles. Usages of the first level of the summation show how variously the summation operator can be substituted. The second level adds to this a function that can be implemented in several ways. The third level creates the possibility to process an arbitrary sequence of elementary values.

The versions shown are not different programming theorems but didactic levels of the teaching of summation. This distinction helps us to underline the abstract features of the summation. Nevertheless the different versions of the summation are not independent. All problems that can be solved at some level of the summation can be also derived from other higher levels.

The parameters of the different levels of the summation are summarized in the table below. It shows which parameter of a certain version of summation has got constraints and which one can be substituted arbitrarily. The first version is the most concrete instance of the summation because it has got only three parameters but the collection that is enumerated must be an array. In the second version four parameters can be found but the collection is always an integer interval

m..n, and the set E is $\mathbb{Z}$. The function $f : \mathbb{Z} \to H$ is more general than the array $array([m \,..\, n], H)$ of the first version. The third level is the most general version with five free parameters where the enumeration contains the elements of the set E.

parameters	first version	second version	third version
E	$E = H$	$\mathbb{Z}$	general
$enor(E)$	enumerator of an $array([m \,..\, n], H)$	enumerator of $m \,..\, n$	general
H	general	general	general
$f : E \to H$	identity	general	general
$+ : H \times H \to H$	general	general	general

The degree of the universality of the programming theorem is measured in the number of the tasks that can be solved based on the programming theorem. We could see that many types of tasks can be solved with application of the summation. Not only can many tasks be solved based on one of the versions of the summation but a couple of programming theorems are derived from the summation. Counting, for example, is a special summation, however every programmer uses it as if it were another theorem. In many education course, conditional summation, copying, assortment and separation appear as separate programming theorems but all of them are special summation. We remark that a special summation is worth being introduced as a new programming theorem if its algorithm differs from the algorithm of the general summation.

Finally, the possibility of further generalization of the summation is going to be investigated. All tasks discussed before have been based on an associative operator $+e_i$ ($+ : H \times E \to H$ with a left-hand zero element. If the associative property were given up, an operational order should be fixed in the specification so that the first element of the enumeration must be processed at first, then the second element etc. According to this, the formula $s = \sum_{i=m \,..\, n}$ must be changed to $s = (\ldots(f(e_1) + f(e_2)) + \ldots + f(e_n))$ where the elements e_1, ..., e_n are enumerated by t. This computation can be described by the processing of the sequence $< e_1, \ldots, e_n >$, i.e. $s = F(< e_1, \ldots, e_n >)$ where the function $F : E^* \to H$ can be defined recursively: for all $i \in [1 \,..\, n]$ $F(< e_1, \ldots, e_i >) = F(< e_1, \ldots, e_i - 1 >) + f(e_i)$ and $F(<>)$ is the zero element of the summation operator. If the summation operator had no zero element, the value of $F(<>)$ should be defined in a unique way. Moreover, the homogeneous summation operator can be changed to an inhomogeneous operator $\tilde{+} : H \times E \to H$. The

problem mentioned above is more general than the problem of 2.3. where this inhomogeneous summation operator can be defined as $h\tilde{+}e = h + f(e)$ for all $h \in H$ and $e \in E$. It is easy to see that the algorithm of the summation on enumerator can essentially solve the task $s = F(< e_1, \ldots, e_n >)$ so we can introduce a new more universal version of the summation programming theorem.

References

[1] Á. Fóthi, *Bevezetés a programozáshoz*, ELTE Eötvös Kiadó, 2005 (in Hungarian).

[2] T. Gregorics and S. Sike, *Generic algorithm patterns*, Proceedings of Formal Methods in Computer Science Education FORMED 2008, Satellite workshop of ETAPS 2008, 141–150, Budapest March 29, 2008.

[3] Sz. Csepregi, A. Dezső, T. Gregorics and S. Sike, *Automatic Implementation of Service Required by Components*, *ETH Technical Report* **567**, PROVECS'2007 Workshop (2007).

[4] T. Gregorics, Programming theorems on enumerator, *Teaching Mathematics and Computer Science, Debrecen* **8**, no. 1 (2010), 89–108.

[5] T. Gregorics, Analogous programming with a template class library, *Teaching Mathematics and Computer Science, Debrecen* **10**, no. 1 (2012), 135–152.

[6] T. Gregorics, Abstract levels of programming theorems, *Acta Universitatis Sapientiae, Informatica* **4**, no. 2 (2012), 247–259.

[7] T. Gregorics, *Programozás 1. kötet Tervezés*, ELTE Eötvös Kiadó, 2012 (in Hungarian).

TIBOR GREGORICS
DEPT. SOFTWARE TECHNOLOGY AND METHODOLOGY
FACULTY OF INFORMATICS, ELTE
1117 BUDAPEST PÁZMÁNY PÉTER SÉTÁNY 1/C

E-mail: gt@inf.elte.hu

(Received January, 2014)