Live & Learn

Tibor Gregorics

gt@inf.elte.hu
ELTE IK

Abstract. In this paper an interesting and surprising case study of my programming education practice is presented. This case underlines the importance of methods, standards and rules of thumb of the programming process. These elements of the programming technology can be taught well in education and they can guarantee the quality of the generated programs. However the case described in this paper brings an anomaly when a programming standard is violated during the programming process and, although it should imply that the implemented program code works badly, the program works perfectly. This anomaly is caused by a typical implementation problem: the boundary and rules of the machine representation of numbers. This anomaly is going to be analyzed and the appropriate conclusions of our case study will be deducted.

Key words and phrases: programming technology, machine representation of numbers

ZDM Subject Classification: P50

1. Introduction

This paper presents a true story that has happened at a laboratory practice of the course Programming Fundamentals led by me in the BSc of Computer Science at Eötvös Loránd University.

The foundation of our training of programming in education is to separate the different phases of the programming process. [5][6] It is particularly important that every computer programmer know exactly which part of the programming activity is being done: the specification, the planning, the implementation or the testing. Different principles are needed when the task must be analyzed or when its solution plan must be considered, and other pieces of advice become important when the implementation of the plan must be done or when the program code must be tested. The separation of these activities helps to train the professional computer programmer, who knows, for example, that it would not be good if the program code of a wrong plan were to modify instead of correcting the plan itself. But when we have got a correct plan and when the result of some test cases is wrong, we must improve the implementation. Thus it offers a lot of advantages if a task must be solved by students when its specification and abstract program are given, and students may only focus on the implementation and testing.

The knowledge of the programming language is essential in the phase of implementation but it is not exclusive. It is important to understand that this phase, which transforms the plan into a concrete program code, needs many decisions and activities that are independent of the programming language. This is why the name of this phase is “implementation” instead of “coding”. Implementation is not identical to coding. During the coding, the statements of the abstract program are translated into a concrete programming language. But the implementation contains many other activities. For example, the code that reads the input data and writes the output data cannot be absent from the final program and the reading part often implies checking the values of input data.

However, the abstract program often does not give any instructions for the way of the communication between the program and its environment, and does not contain any information about the kind of constraints that should be checked on the values read. Only the specification can show which are the input and output data, and the constraints of input data can be found in the precondition. Thus a significant amount of program code is created beyond the code of the abstract program. Similarly, the achievement of the non-functional requirements such as the improvement of memory or time complexity goes beyond mechanic coding.

Programming technology gives a lot of methods, standards and rules of thumb to help to make computer programs. The story begins with a control test when students had to implement a given triple task-specification-algorithm in C++ language and had to confirm that they understand and can use the rules of thumb of implementation taught by us. Incidentally this control test was preceded by another lesson with second-year students where the formal proof of the correctness of algorithms was practised. It came in handy that the correctness of the abstract program selected to the control test of the first-year students was proven. This proof strongly relied on the significant constraint of the precondition of specification that guaranteed the termination of the program. If somebody forgets to implement this constraint into the checking of input data, they will get a wrong solution. However my surprise was huge when one of these wrong implementations could work perfectly.

Hereinafter this case is going to be presented, the anomaly of its execution is going to be investigated and the conclusions of this analysis are going to be summarized.

2. Plans of a simple programming task

Let us consider the following task. Calculate the product of two natural numbers so that the operator multiplication cannot be used, only the addition and the subtraction.

The first step of the solution is to give the specification of the problem. It has got two input numbers that must be multiplied and one output number. The data type of these numbers is integer but all of them must be non-negative.

The formal specification with the data:

Input :	$a \in \mathbb{Z}, b \in \mathbb{Z}$
Output :	$c \in \mathbb{Z}$
Precondition :	$a \geq 0 \wedge b \geq 0$
Postcondition :	$c = a \cdot b$

The data of the problem may be also derived directly from natural numbers ($\mathbb{N}$) but in this case the precondition is empty. However, the previous form is better because it underlines that there is a very important constraint on the input data and this condition will have to be checked in the program code.

Three integer variables may be introduced: *x*, *y*, and *z*. The variables *x* and *y* will contain the input data, the result will appear in the variable *z*. The state space ($x:\mathbb{Z}, y:\mathbb{Z}, z:\mathbb{Z}$) of this problem will be denoted by *A*. This is the set of all possible states, i.e. of all possible triples of integers, and their components are referred by the variables. [4] Now the formal specification with variables can be considered:

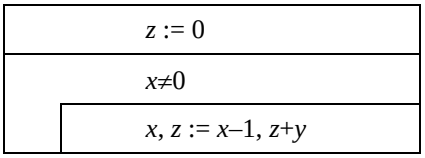
A	$= (x:\mathbb{Z}, y:\mathbb{Z}, z:\mathbb{Z})$
Pre	$= (x=a \wedge y=b \wedge x \geq 0 \wedge y \geq 0)$ where *a* and *b* are arbitrarily fixed natural numbers
Post	$= (z = a \cdot b)$

The condition $Pre{:}A \longrightarrow \mathbb{L}$ identifies the initial states of the problem and fixes arbitrary initial values (a and b) of the variables x and y. The condition $Post{:}A \longrightarrow \mathbb{L}$ describes the goal states that belong to the fixed initial states.

Two different solutions (algorithms) are going to be showed with their formal verifications. The common property of these algorithms is that they do not use temporary variables. Both algorithms are a sequence of an assignment and a loop, and the body of these loops contains a simultaneous assignment.

2.1. First plan

The structurgramme of the first algorithm is the following:



Its correctness is going to be proved based on the formal method of Hoare [1]. We must verify that the abstract program (S_1) can solve the problem, i.e. the program is completely correct regarding the specification (A, Pre, $Post$). This fact can be denoted with $\{\{Pre\}\}S_1\{\{Post\}\}$. Since the structure of the program is a sequence, it is enough to prove that $\{\{Pre\}\}\ z := 0\ \{\{Q\}\}$ and $\{\{Q\}\}loop\{\{Post\}\}$ where $Q{:}A \longrightarrow \mathbb{L}$ is an appropriate intermediate statement. (The *loop* denotes the loop of the program.) On the one hand, the criterion $\{\{Pre\}\}\ z := 0\ \{\{Q\}\}$ is true if $Pre \Longrightarrow Q^{\,z \leftarrow 0}$. The $Q^{\,z \leftarrow 0}$ means that the occurrences of z in Q must be exchanged with zero. On the other hand, instead of the criterion $\{\{Q\}\}loop\{\{Post\}\}$ it is enough to prove that the criteria $Q \Longrightarrow I$, $I \wedge x{=}0 \Longrightarrow Post$, $I \wedge x{\neq}0 \Longrightarrow t{\in}\mathbb{N}$ and, for all integer c_0, the $\{\{I \wedge x{\neq}0 \wedge t{=}c_0\}\}\ x, z := x{-}1, z{+}y\ \{\{I \wedge t{<}c_0\}\}$ are true. In these criteria the logical function $I{:}A \longrightarrow \mathbb{L}$ is an appropriate, so-called invariant condition and the function $t{:}A \longrightarrow \mathbb{Z}$ is an appropriate, so-called termination function. The last criterion can be proven with the verification of the statement $I \wedge x{\neq}0 \wedge t{=}c_0 \Longrightarrow (I \wedge t{<}c_0)^{\,x, z \leftarrow x-1, z+y}$.

Let the invariant I be the statement $x{\geq}0 \wedge z{+}x{\cdot}y{=}a{\cdot}b$, the termination function t always shows the value of the variable x, and let $Q = I$. In this case the criteria which must be proven are the following:

- $Pre \Longrightarrow I^{z \leftarrow 0}$. It is true because $Pre = (\ x{=}a \wedge y{=}b \wedge x{\geq}0 \wedge y{\geq}0\ )$ and $I^{\,z \leftarrow 0} = (x{\geq}0 \wedge x{\cdot}y{=}a{\cdot}b)$;
- $Q \Longrightarrow I$. It is fairly trivial because $Q = I$;
- $I \wedge x{=}0 \Longrightarrow Post$. It is true because $z{+}x{\cdot}y{=}a{\cdot}b$ is in I, thus $x{=}0$ implies $z{=}a{\cdot}b$;
- $I \wedge x{\neq}0 \Longrightarrow x{\in}\mathbb{N}$. It is true because $x{:}\mathbb{Z}$ and I implies $x{\geq}0$.
- $I \wedge x{\neq}0 \wedge x{=}c_0 \Longrightarrow (I \wedge x{<}c_0)^{\,x, z \leftarrow x-1, z+y}$. The consequence must be calculated in the following way: $(I \wedge x{<}c_0)^{\,x, z \leftarrow x-1, z+y} = (x{-}1{\geq}0 \wedge z{+}y{+}(x{-}1){\cdot}y{=}a{\cdot}b \wedge x{-}1{<}c_0)$. The $x{-}1{\geq}0$ comes from the loop condition $x{\neq}0$ and from $x{\geq}0$ of I. The equation $z{+}y{+}(x{-}1){\cdot}y{=}a{\cdot}b$ is the same that the equation $z{+}x{\cdot}y{=}a{\cdot}b$ in I. The condition $x{-}1{<}c_0$ is true if $x{=}c_0$.

It can be observed that neither the algorithm nor the proof its correctness uses the constraint $y{\geq}0$ of the precondition. In contrast, the constraint $x{\geq}0$, which also appears in the invariant, is indispensable to the proof of terminatiton. The constraint $x{\geq}0$ implies that the value of the variable x (this is the value of the termination function) is a natural number during the loop. However, the

body of the loop decreases this value. Thus the loop must terminate because a natural number cannot be decreased infinite times so that it remains a natural number.

2.2. Second plan

The second algorithm is a good example of a case when the solution is not born in the way of the algorithmic thinking but of the formal program synthesis. This process is also based on Hoare's method but initially the algorithm is not known. The algorithm is formed step by step so that meanwhile the criteria of the correctness derived from Hoare's method are also proven. (Here we also follow Dijkstra's and Gries' work. [2][3])

Let us suppose that a loop can solve the problem. In this case an invariant statement is needed in order to check, for example, the criterion $I \wedge \neg$*loop-condition* $\Longrightarrow$ *Post*. Almost the same invariant is selected as in the first solution but it is extended by both extra constraints of the precondition, that is $I = (x \geq 0 \wedge y \geq 0 \wedge z+x \cdot y = a \cdot b)$. It can be seen that the postcondition follows from this invariant if $x=0$ or $y=0$. Thus the loop condition must be given as $x \neq 0 \wedge y \neq 0$ and one criterion has been proven.

We need a termination function. Its value depends on the variables of the program and this value must be decreased by the body of the loop. If $t = x+y$, then its value is a natural number because I contains $x \geq 0 \wedge y \geq 0$ and it can be decreased by decreasing x, by decreasing y, or by decreasing both. Thus the simultaneous assignment $x, y := x-1, y-1$ is placed into the body of the loop. Meanwhile, two other criteria are proven: $I \wedge x \neq 0 \wedge y \neq 0 \Longrightarrow x+y \in \mathbb{N}$ and for all integer c_0 $\{\{I \wedge x \neq 0 \wedge y \neq 0 \wedge x+y=c_0 \}\}$ $x, z := x-1, z+y$ $\{\{ x+y < c_0 \}\}$. The last one is the half of the criterion $\{\{I \wedge x \neq 0 \wedge y \neq 0 \wedge x+y=c_0\}\}$ $x, z := x-1, z+y$ $\{\{I \wedge x+y<c_0\}\}$.

Now the other half of the last criterion should also be proven, that is, $\{\{I \wedge x \neq 0 \wedge y \neq 0\}\}$ $x, y := x-1, y-1$ $\{\{I\}\}$. Instead of this, it is enough to show that

$$I \wedge x \neq 0 \wedge y \neq 0 \Longrightarrow I^{x, y \leftarrow x-1,\ y-1} = (x-1 \geq 0 \wedge y-1 \geq 0 \wedge z+(x-1) \cdot (y-1) = a \cdot b) =$$

$$= (x-1 \geq 0 \wedge y-1 \geq 0 \wedge z-x-y+1+x \cdot y = a \cdot b).$$

The condition $x-1 \geq 0 \wedge y-1 \geq 0$ follows from the loop condition $x \neq 0 \wedge y \neq 0$ and from $x \geq 0 \wedge y \geq 0$ of I. However, the equation $z-x-y+1+x \cdot y = a \cdot b$ is not true. Comparing it with the equation $z+x \cdot y = a \cdot b$ of I, we can see that there is less on its left side than on the right side by $x+y-1$. Nevertheless, if the value $x+y-1$ were added to the variable z, the equation $z-x-y+1+x \cdot y = a \cdot b$ would be true and the criterion $I \wedge x \neq 0 \wedge y \neq 0 \Longrightarrow I^{x, y,\ z \leftarrow x-1, y-1, x+y-1}$ would be proven. According to this, the simultaneous assignment of the body of the loop must be extended by $z := z+x+y-1$. This modification has no impact on the criteria justified earlier: the body of the loop can still decrease the value of the termination function.

Only one criterion must be proven yet: *Pre* $\Longrightarrow$ *I*. Altough the $x \geq 0 \wedge y \geq 0$ of I comes from *Pre*, the equation $z+x \cdot y = a \cdot b$ does not. It would be true only if $z=0$ ($x=a$ and $y=b$ also come from *Pre*). Thus *Pre* $\Longrightarrow$ *I* is not true but *Pre* $\Longrightarrow I^{z \leftarrow 0}$, this can be written in the form $\{\{Pre\}\}$ $z := 0$ $\{\{I\}\}$, is true. If the precondition of the loop created before is changed to *I*, the missing criterion of the correctness of the loop will be $I \Longrightarrow I$. Thus $\{\{I\}\}$loop$\{\{Post\}\}$ will be true. If the structure of our algorithm (it is denoted by S_2) were a sequence of the assignment $z := 0$ and of the loop, the algorithm would solve the problem, hence $\{\{Pre\}\}S_2\{\{Post\}\}$ follows from $\{\{Pre\}\}$ $z := 0$ $\{\{I\}\}$ and $\{\{I\}\}$loop$\{\{Post\}\}$.

$z := 0$	
$x \neq 0 \wedge y \neq 0$	
	$x, y, z := x-1, y-1, z+x+y-1$

The constraints $x \geq 0$ and $y \geq 0$ guarantee that the loop can terminate in finite steps. These constraints imply that the value of the termination function is a natural number during the execution of the loop. As the body of the loop decreases the value of the termination function, the loop must stop in finite steps because a natural number cannot be decreased infinite times so that it remains a natural number.

3. Implementation and testing of the plans

The implementation and testing of a plan can be helped by numerous methods, standards and rules of thumb. [6]

A general principle of the implementation is that the code of the program can be made in five steps: the frame, the code of the abstract program, the reading and checking of input data, the writing of the results, and the declaration of the variables. Some rules of thumb connecting to these steps are presented below.

1. The frame of the program code depends on the selected programming language. For example in C++ the main program must be placed into the body of the function *main()*.
2. The input variables are the ones which are mentioned in the precondition of the specification.
   - An extra code is needed to read the initial value of the input variables.
   - The data type of the input variable and the extra constraint of the precondition show together what kind of data checking must be implemented after the reading.
   - The non-functional requirements (such as the foolproof reading) can influence the way of data checking.
3. The result variables can be found in the postcondition of the specification. Their values must be written before the program terminates. (Attention! The same variable may be input and output at the same time.)
4. The code of the structured abstract algorithm can be made along the embedded program structures top-down.
   - A simultaneous assignment can be substituted by simple assigments. Their order is determined by the dependency of the variables of the simultaneous assignment.
5. Before the variables are declared in the program code, it is important to see which concrete data type can substitute the famous abstract data type. For example in C++ the int can be used instead of the type of natural numbers or integers.

The effects of these rules of thumb can be seen in the correct implementation of the first plan. (Figure 1.)

Certainly, a wrong code may be also created. In my story, the typical mistake was that students forgot to implement the extra constraint of the precondition and did not build the checking of input data into the code. (Figure 2.)

```
#include <iostream>
using namespace std;

int main()
{
    int x, y, z;
    cout << "x = "; cin >> x;
    if(x<0){
        cout << "x must be a natural number!\n";
        return 1;
    }
    cout << "y = "; cin >> y;
    if(y<0){
        cout << "y must be a natural number!\n";
        return 1;
    }

    z = 0;
    while(x!=0){
        x = x - 1; z = z + y;
    }
    cout << "x*y = " << z << endl;
    return 0;
}
```

Figure 1.

The correct code of the first solution

```
#include <iostream>
using namespace std;

int main()
{
    int x, y, z;
    cout << "x = "; cin >> x;
    cout << "y = "; cin >> y;

    z = 0;
    while(x!=0){
        x = x - 1; z = z + y;
    }
    cout << "x*y = " << z << endl;
    return 0;
}
```

Figure 2.

The wrong code of the first solution

A wrong code can be detected by testing. Let us see a few suggestions for testing of the solution of simple programming tasks. [6]

1. There are two basic categories of the test cases: the black box test cases and the white box test cases. The black box test cases are based only on the specification. The white box test cases are derived from the program code.
2. Inside the black box test cases, it is worth distinguishing the valid and the invalid cases. An input data is valid if it satisfies the constraints of the precondition. Otherwise it is invalid.

3. Special test cases are generated by the properties of a data type. For example the values of the integer can be divided into three parts: positive ones, negative ones, and zero.
4. The postcondition helps to discover the special test cases. For example, at a multiplication, it is worth checking the commutative and associative properties of the multiplication operator, or the result if one of the operands is zero or a unit.

Let us return to our wrong code. It can work correctly with valid input data. (Figure 3.)

Figure 3.

The execution of the first solution with valid input data

The wrong version should work incorrectly with invalid input data, when the value of x is negative. Analyzing the abstract algorithm we would expect that the program will not terminate if $x=-2$ and $y=5$. (Figure 4.) However, in spite of our expectation, the execution of the program stops, although the running time is longer than the running time with valid input data. The most surprising fact is that the result is also correct.

Figure 4.

The execution of the first solution with one invalid input data

On top of all that, the program can work correctly with any integers. (Figure 5.)

Figure 5.

The execution of the first solution when both input data are invalid

This same phenomenon can be observed when the implementation of the second solution is analyzed. The only difference is that not only the negative x but also the negative y should cause an infinite loop if $y<x$. But it does not happen.

What may be the cause of this: the originally correct algorithm with an incorrect implementation can work correctly? Does it work correctly with only some invalid input data or with all of them?

4. Machine representation of numbers

There is no doubt that the cause of the termination of our program with invalid input data can be found among the restrictions of the representation of numbers. The smallest negative integer that can be represented on k bits in two's complement is -2^{k-1} and its code is $100\ldots000_{(2)}$. Decreasing it by 1, that is, adding the two's complement code of -1 (this is $111\ldots111_{(2)}$) we get the k+1 length code $1011\ldots111_{(2)}$. Because of the overflow, the first bit is lost and the value of the remaining code is $011\ldots111_{(2)}$, that is $2^{k-1}-1$. This is the greatest positive integer that can be represented on k bits in two's complement. According to this, when the variable x starts to decrease one by one from its initial value -2 and it reaches the smallest negative integer, the next step results in the greatest positive integer in x. Continuing the reduction of x, the value of x reaches zero and the program terminates. Nevertheless, why is the result of the wrong program correct?

Let us introduce some notations. If $k\in\mathbb{N}^+$, $n\in\mathbb{N}$, $x\in\mathbb{Z}$, and $s\in\{0,1\}^k$ (k-length sequence of bits)

$$bin : \mathbb{N}\times\mathbb{N}^+ \longrightarrow \{0,1\}^k$$

$$bin(n,k) ::= < (n \mathbin{div} 2^{k-1}) \mathbin{mod} 2 , (n \mathbin{div} 2^{k-2}) \mathbin{mod} 2 , \ldots , n \mathbin{mod} 2 >$$

$$value : \{0,1\}^k \longrightarrow \mathbb{N}$$

$$value(s) ::= \sum_{i=1\ldots k} s_i * 2^{k-i}$$

$$inverse : \{0,1\}^k \longrightarrow \{0,1\}^k$$

$$inverse(s) ::= < 1-s_1 , \ldots , 1-s_k >$$

$$inc : \{0,1\}^k \longrightarrow \{0,1\}^k$$

$$inc(s) ::= bin(value(s) + 1, k)$$

$$code : \mathbb{Z}\times\mathbb{N}^+ \longrightarrow \{0,1\}^k$$

$$code(x,k) ::= bin(x,k) \quad \text{if } x\geq 0$$

$$code(x,k) ::= inverse(\mathrm{inc}(\mathrm{bin}(|x|,k))) \quad \text{if } x<0$$

$$decode : \{0,1\}^k \longrightarrow \mathbb{Z}$$

$$decode(s) ::= value(s) \quad \text{if } s_1=0$$

$$decode(s) ::= -value(inc(inverse(s))) \quad \text{if } s_1=1$$

Denote $[x]_k$ the integer that is the value of two's complement code of the integer x on k bits. Shortly: $[x]_k ::= decode(code(x,k),k)$. Because of the overflow, it is conceivable that $[x]_k \neq x$.

Our aim is to show that the result of our badly implemented program is equal to the correct product $x \cdot y$ for all integer x and y if there is no overflow in the representation of x, y, or $x \cdot y$. First of all, two lemmas must be proven.

***Lemma*:** Let a and b be arbitrary integers.

i. $[a + n \cdot 2^k]_k = [a]_k$ for all $n \in \mathbb{Z}$.

ii. $[\,[a]_k + [b]_k]_k = [a + b]_k$

***Proof*:** Since the last k bits of the binary code of 2^k are zero, if $n \cdot 2^k$ (where n is an arbitrary integer) is added to an arbitrary integer a, the last k bits of its binary code will not change. It follows that the first statement is true. In the second statement, we must take the form $a = [a]_k + n \cdot 2^k$ and $b = [b]_k + m \cdot 2^k$ where $n, m \in \mathbb{Z}$ and

$$[a + b]_k = [\,[a]_k + n \cdot 2^k + [b]_k + m \cdot 2^k\,]_k = [\,[a]_k + [b]_k + (n+m) \cdot 2^k\,]_k = [\,[a]_k + [b]_k\,]_k. \square$$

Denote $res_1(x, y)$, the result of the first badly implemented program, and $res_2(x, y)$, the result of the second badly implemented program, if the input data are x and y.

***Theorem*:** For all integers x and y where $[x]_k = x$ and $[y]_k = y$, $res_1(x, y) = [x \cdot y]_k$.

***Proof*:** The first program counts a multiple summation of y where overflow may occur:

$$res_1(x, y) = [\, \ldots [[y + y]_k + y]_k + \ldots \,]_k.$$

Because of the lemma *ii*, this formula can be written as

$$res_1(x, y) = [\, y + y + \ldots \,]_k.$$

If $x \geq 0$ (this case was verified in the framework of planning), the number of the iteration is x, so

$$res_1(x, y) = [\textstyle\sum_{i=1 \ldots x} y]_k = [x \cdot y]_k.$$

If $x < 0$, the number of the iteration is $2^k - |x|$, and using lemma *i* where $a = x \cdot y$ and $n = y$, furthermore $x = -|x|$ because $x < 0$, we get

$$res_1(x, y) = [\textstyle\sum_{i=1 \ldots 2^k - |x|} y]_k = [(2^k - |x|) \cdot y]_k = [(2^k + x) \cdot y]_k = [2^k \cdot y + x \cdot y]_k = [x \cdot y]_k. \square$$

***Theorem*:** For all integers x and y where $[x]_k = x$ and $[y]_k = y$, $res_2(x, y) = [x \cdot y]_k$.

***Proof*:** The second program counts a multiple summation of $x+y-1$ where overflow may occur:

$$res_2(x, y) = [\, \ldots [[\, (x+y-1) + ([x-1]_k + [y-1]_k - 1]_k) + ([x-2]_k + [y-2]_k - 1]_k) + \ldots \,]_k.$$

Because of the lemma *ii*, this formula can be written as

$$res_2(x, y) = [\, (x+y-1) + (x-1+y-1-1) + (x-2+y-2-1) + \ldots \,]_k.$$

Without loss of generality, we can assume that $x < y$.

If $x \geq 0$ (this case was verified in the framework of planning), the number of the iteration is x, so

$$res_2(x, y) = [\textstyle\sum_{i=0 \ldots x-1} x - i + y - i - 1]_k = [-2 \cdot (\sum_{i=0 \ldots x-1} i) + (\sum_{i=0 \ldots x-1} x + y - 1)]_k =$$

$$= [-x \cdot (x-1) + x \cdot (x+y-1)]_k = [x \cdot y]_k.$$

If $x < 0$, the number of the iteration is $2^k - |x|$, so

$$res_2(x, y) = [(\sum_{i=0 \dots 2^k-|x|-1} x-i+y-i-1)]_k = [-2\cdot(\sum_{i=0 \dots 2^k-|x|-1} i)+(\sum_{i=0 \dots 2^k-|x|-1} x+y-1)]_k =$$

$$= [-(2^k-|x|)\cdot(2^k-|x|-1)+ (2^k-|x|)\cdot(x+y-1)]_k = [\ (|x|-2^k)\cdot(2^k-|x|-x-y)]_k =$$

$$= [|x|\cdot(-|x|-x-y) - 2^k\cdot(-|x|+2^k+|x|-x-y)]_k =$$

Using lemma *i* ($a = |x|\cdot(-|x|-x-y)$ and $n = -(-|x|+2^k+|x|-x-y)$), furthermore $x = -|x|$ because $x<0$, we get

$$= [|x|\cdot(-|x|-x-y)]_k = [(-x)\cdot(-y)]_k = [x \cdot y]_k. \square$$

5. Conclusions

The above case study highlights the importance of the partition of the programming process. Due to this, it was easy to detect that the incorrect implementation caused the anomaly observed. It is also very important to separate the methods, standards and rules of thumb according to the appropriate phase of the programming process which uses them. Different competences are needed for planning and different for testing. If the programming language of the implementation is changed, most of the rules of thumb of the implementation and all testing strategies remain the same. Programmers must know which knowledge may be renewed, and which one must be saved.

Our case study gave an example of a case when ignoring some rules of thumb could improve the program code: remember that it could also work on negative numbers. In this case, two rules of thumb were violated but their effects neutralized each other. The conjunction of the abandonment of data checking and the disregard of the boundary of the machine representation result in an extraordinary phenomenon. However, this example was exceptional. In general, we are not so lucky. In spite of this case, methods, standards and rules of thumb must be applied and taught.

An important conclusion of our case study is that the role of the boundary of the machine representation is more important in simple programming tasks than what I have thought earlier. The replacement of an abstract data type with a concrete one deserves more attention.

Perhaps the most interesting conclusion was that I have discovered an enigma in a course for first-year students after thirty years of practice. Identifying and solving this riddle do not only cause felicity to us but it can also complement our expertise in informatics. Well, you live and learn.

References

1. Hoare, C. A. R.: An axiomatic basis for computer programming. Comm. ACM 12 (10), pp. 576-580, 1969.
2. Dijkstra, E.W.: A Discipline of Programming, Prentice-Hall, 1976.
3. Gries, D: The Science of Programming, Springer-Verlag, 1981.
4. Gregorics, T.: Concept of abstract program, Acta Universitatis Sapientiae, Informatica, 4, 1 (2012), 7-16
5. Gregorics, T.: Programozás 1.kötet Tervezés. ELTE Eötvös Kiadó, 2013. (in Hungarian)
6. Gregorics, T.: Programozás 2.kötet Megvalósítás. ELTE Eötvös Kiadó, 2013. (in Hungarian)