Programming Theorems Have the Same Origin

SZLÁVI Péter, TÖRLEY Gábor, ZSAKÓ László

Abstract: One of the classical ways of learning programming is to divide programming tasks into large groups, so-called programming theorems, and then to trace the specific tasks back to the programming theorems. Each teaching method introduces a different amount of programming theorems into the learning process, occasionally even combining them. In this article we will show that the basic and complex programming theorems have the same origin; consequently, it would be enough to present one theorem and trace everything back to it. At the end of the article, then, we will explain the practical use of still introducing more theorems.

Keywords: programming theorem, programming methodology, methodical programming, algorithm, specification

1. Introduction

In order to be able to solve a programming task, we need to identify its essence: programming tasks can be categorized into groups according to their type, which is useful because for each group we can create an algorithm rule that solves all of the tasks in that specific group. These task types are called programming theorems because their solutions are justifiably the correct solutions. Their number can vary from teaching method to teaching method. [1,2,3,4]

If we are familiar with the programming theorems, all we have to do to solve most tasks is to recognize the suitable programming theorem, use the specific data of the general task type, and in the general algorithm substitute them with the task-specific data. Applying this method is supposed to lead us to the correct solution. [5,6]

In these tasks we usually have to assign a certain result to one (or more) data collection(s), which, for simplicity's sake, we will handle as some sort of sequences. The essence of a sequence is the processing order of the given elements. Most of the times, it is sufficient to deal with sequences whose elements can be processed – one by one – one after another. In the input sequence, this requires an operation that is capable of giving the elements of the sequence one by one, while in the output sequence elements can be followed by a new element. In simple cases sequences can be illustrated as arrays.

The basic programming theorems are those that assign one value to one sequence: [7]:

- sequential computing
- counting
- maximum selection
- decision
- selection
- search

2. The first programming theorem

The first programming theorem comes from a simple task, that of calculating the sum of numbers. In the following, we will provide the specification and the algorithm of the task, which are the two pillars of the programming theorem. We are applying the marking system from [5,6].

Input: $N \in \mathbb{N}$, $X \in H^N$, $\Sigma: H^* \to H$, $+: H \times H \to H$ (where $H=\mathbb{N}$ or $H=\mathbb{Z}$ or $H=\mathbb{R}$)
$\Sigma(X_1,...,X_N) = \Sigma(X_1,...,X_{N-1}) + X_N$, $F()=0$

Output: $S \in H$

Precondition: –

Postcondition: $S = \Sigma(X_1,...,X_N)$

```
Sum(N,X,S):
  S:=0
  For i:=1 to N do
    S:=S+X[i]
  End For
End.
```

Generalizing this task, we will get the so-**called sequential computin**g programming theorem. We generalize the operation +:

1) the generalized, binary (f) operation should have a (left-side) zero element (F0);

2) when we apply the operations one after another, the result should not depend on the execution order; that is, the operations should be associative.

The operation, based on binary operation f and zero element F0, and interpreted at H*, is indicated with F.

Input: $N \in \mathbb{N}$, $X \in H^N$, $F: H^* \to H$, $f: H \times H \to H$, $F_0 \in H$
$F(X_1,...,X_N) = f(F(X_1,...,X_{N-1}), X_N)$, $F() = F_0$

Output: $S \in H$

Precondition: –

Postcondition: $S = F(X_1,...,X_N)$

Note: many times F is Σ, average, deviation, scalar multiplication, Π, $\cup$, $\cap$, $\forall$, $\exists$, writing one after another, Max, Min.

```
Computing(N,X,S):
  S:=F₀
  For i:=1 to N do
    S:=f(S,X[i])
  End For
End.
```

Note: we are indicating changes from the previous algorithms with **bold**, both here and from now on.

3. Counting

Sequential computing can be formulated so that its result is the sum of all elements with T feature; thus, leading us to the **counting** programming theorem.

Function F will be a **conditional sum**. This means that we define function f as a **conditional function**, whose value is the first parameter+1 if the second parameter is of T feature, otherwise it is the value of the first parameter.

Input: $N \in \mathbb{N}$, $X \in H^N$, $F:H^* \rightarrow \mathbb{N}$, $f:\mathbb{N} \times H \rightarrow \mathbb{N}$, $F_0 \in \mathbb{N}$

$$F(X_1,\dots,X_N)=\begin{cases} F(X_1,\dots,X_{N-1})+1 & \text{if } T(X_N) \\ F(X_1,\dots,X_{N-1}) & \text{otherwise} \end{cases},$$

$F()=F_0$

$$f(a,b):=\begin{cases} a+1 & \text{if } T(b) \\ a & \text{otherwise} \end{cases},$$

$F_0=0$

Output: $Count \in \mathbb{N}$

Precondition: –

Postcondition: $Count=F(X_1,\dots,X_N) \rightarrow Count = \sum_{\substack{i=1 \\ T(X_i)}}^{N} 1$

```
Counting(N,X,S):
  Count:=0
  For i:=1 to N do
    Count:=f(Count,X[i])  →  [If T(X[i]) then Count:=Count+1*]
  End For
End.
```

Note: the change marked with * is the normalized algorithmic transformation of the conditional expression.

4. Maximum selection

If we replace function **f** with function **max**, we will get to one of the versions of the **maximum selection** theorem.

Input: $N \in \mathbb{N}$, $X \in H^N$, $F:H^* \rightarrow H$, $max:H \times H \rightarrow H$, $F_0 \in H$, (H ordered set)
$F(X_1,\dots,X_N)=max(F(X_1,\dots,X_{N-1}),X_N)$,
$F()=F_0$

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

$F_0=-\infty$

Output: $MaxVal \in H$

Precondition: $N>0$

Postcondition: $MaxVal=F(X_1,\dots,X_N)$

```
Maximum(N,X,MaxVal):
  MaxVal:=-∞
  For i:=1 to N do
    MaxVal:=max(MaxVal,X[i]) → [If X[i]>MaxVal then MaxVal:=X[i]]
  End For
End.
```

With a very simple modification, we can even define the version that provides the maximum index as well. The output, the post-condition and the algorithm change as follows:

Output: $MaxVal \in H$, $MaxInd \in \mathbb{N}$

Postcondition: $MaxVal=F(X_1,...,X_N)$ and $1 \leq MaxInd \leq N$ and $X_{MaxInd}=MaxVal$

```
Maximum(N,X,MaxVal,MaxInd):
  MaxVal:=-∞
  For i:=1 to N do
    If X[i]>MaxVal then MaxVal:=X[i]; MaxInd:=i
  End For
End.
```

If we make use of the precondition, we can leave out the check of the first element from the iteration; in fact, in the classical version we do not need the maximum value either. Hence, the final version is like this:

```
Maximum(N,X,MaxInd):
  MaxInd:=1
  For i:=2 to N do
    If X[i]>X[MaxInd] then MaxInd:=i
  End For
End.
```

5. Decision

In sequential computing, function **F** can be operator ∃ too, which leads us to the **decision** theorem. The operator 'or' is associative, its zero element is 'false'; thus, we can include it in the basic version.

Input: $N \in \mathbb{N}$, $X \in H^N$, $\exists: H^* \to L$, $T: H \to L$, $or: L \times L \to L$, $F_0 \in L$
$F(X_1,...,X_N)=\exists i(1 \leq i \leq N): T(X_i)$
$\exists i(1 \leq i \leq N): T(X_i) \equiv \exists i(1 \leq i \leq N\text{-}1): T(X_i)$ or $T(X_N)$,
$f(a,b)=a$ or b, $F()=F_0=$false

Output: $Exists \in L$

Precondition: –

Postcondition: $Exists=\exists i(1 \leq i \leq N)\ T(X_i)$

```
Decision(N,X,Exists):
  Exists:=false
  For i:=1 to N do
    Exists:=Exists or T(X[i])
  End For
End.
```

Note: in the iteration the variable **Exists** can change in the following way:

- `false, ..., false, true, ..., true`
- `false, ..., false`

that is, either it remains to be false all through, or it becomes true and stays like that. Consequently, once it becomes true, the iteration can stop. Before it does, it is constantly false, so we do not need to change value. The decision programming theorem derives from this: by the end of the iteration we will be able to distinguish which of the two cases we are dealing with.

Input: $N \in \mathbb{N}, X \in H^N, T:H \rightarrow L$

Output: $Exists \in L$

Precondition: —

Postcondition: $Exists = \exists i(1 \leq i \leq N)\ T(X_i)$

```
Decision(N,X,Exists):
  i:=1
  While i≤N and not T(X[i])
    i:=i+1
  End While
  Exists:=(i≤N)
End.
```

6. Selection, Search

At the end of the decision iteration, the value of variable i is the ordinal number of the item of feature T, provided that we know such an item exists (that is, the iteration will stop once an item is found) → **Selection** theorem

Input: $N \in \mathbb{N}, X \in H^N, T:H \rightarrow L$

Output: $S \in \mathbb{N}$

Precondition: $\exists i(1 \leq i \leq N)\ T(X_i)$

Postcondition: $1 \leq S \leq N$ and $T(X_S)$

```
Selection(N,X,S):
  i:=1
  While ~~i≤N and~~ not T(X[i])
    i:=i+1
  End While
  S:=i
End.
```

The ordinal number of an element of T quality is N+1 if it has not gone beyond the end of the sequence. If it has → **Search** theorem

Input: $N \in \mathbb{N}, X \in H^N, T:H \rightarrow L$

Output: $Exists \in L, S \in \mathbb{N}$

Precondition: —

Postcondition: $Exists = \exists i(1 \leq i \leq N)\ T(X_i)$ and $Exists \rightarrow 1 \leq S \leq N$ and $T(X_S)$

```
Search(N,X,Exists,S):
  i:=1
  While i≤N and not T(X[i])
    i:=i+1
  End While
  Exists:=(i≤N)
  If Exists then S:=i
End.
```

7. Complex programming theorems

Complex programming theorems belong to problems that assign sequence(s) to sequence(s) [8]:

- copying,
- multiple item selection,
- partitioning,
- intersection,
- union.

8. Copying – function calculation

We can also define the theorem of sequence computing by applying function (**g**) to each element of the input array and placing the element (**f**) at the end of the output array (**Y**). This way, the null element (**F_0**) of Y will be the empty array, and function **F** will add the elements of **$g(X_i)$** to the elements of **Y.** Practically, function **F** will be an addition operation interpreted for the array (which is what have signaled below with operation „push back").

Input: $N\in\mathbb{N}$, $X\in H^N$, $g:H\to G$, $F: G^N\to G$, $f: G^*\times G\to G$, f – push back

Output: $Y\in G^N$

Precondition: –

Postcondition: $\forall i(1\le i\le N)$: $Y=F(g(X_1),\ldots, g(X_N))$

```
Copying(N,X,Y):
  Y:=empty
  For i:=1 to N do
    Y:=push_back(Y,g(X[i]))
  End For
End.
```

If we assigned **string** type variables to the input and output arrays, the above algorithm would look like this:

```
Copying(N,X,Y):
  Y:=""
  For i:=1 to N do
    Y:=Y+g(X[i])
  End For
End.
```

This refers even more to sequence computation.

Since each element of **Y** will be **$g(X_i)$**, the theorem can be expressed like this as well (we are indicating function **g** from the previous algorithm with **f** now):

Input: $N\in\mathbb{N}$, $X\in H^N$, $f:H\to G$

Output: $Y\in G^N$

Precondition: –

Postcondition: $\forall i(1\le i\le N)$: $Y_i=f(X_i)$

```
Copying(N,X,Y):
  For i:=1 to N do
    Y[i]:=f(X[i])
  End For
End.
```

Finally, we will arrive to the theorem of conditional copying with a simple modification, namely, that we apply function calculation only to elements with property T.

Input: $N \in \mathbb{N}, X \in H^N, f{:}H \rightarrow G, T{:}H \rightarrow \mathbb{L}$

Output: $Y \in G^N$

Precondition: –

Postcondition: $\forall i(1 \leq i \leq N)\ T(X_i) \rightarrow Y_i = f(X_i)$ and not $T(X_i) \rightarrow Y_i = X_i$

```
Copying(N,X,Y):
  For i:=1 to N do
    If T(X[i]) then Y[i]:=f(X[i]) else Y[i]:=X[i]
  End For
End.
```

9. Multiple item selection

In sequence computing we may substitute function **F** with conditional addition interpreted for the array, which will lead us to the theorem of multiple item selection. $\mathbf{X_i}$ will be added to array **Y** if it has property **T**, otherwise it is not added (that is, we add nothing).

Input: $N \in \mathbb{N}, X \in H^N, T{:}H \rightarrow \mathbb{L}$

Output: $Y \in H^*$

Precondition: –

Postcondition: $\forall i(1 \leq i \leq N){:}\ T(X_i) \rightarrow X_i \in Y$ and not $T(X_i) \rightarrow X_i \notin Y$

```
Multiple_item_selection(N,X,Y):
  Y:=empty
  For i:=1 to N do
    If T(X[i]) then Y:=push_back(Y,X[i]) {else nothing to do}
  End For
End.
```

Starting from counting:

```
Counting(N,X,Count):
  Count:=0
  For i:=1 to N do
    If T(X[i]) then Count:=Count+1
  End For
End.
```

Connecting the two algorithms:

```
Multiple_item_selection(N,X,Count,Y):
  Count:=0
  For i:=1 to N do
    If T(X[i]) then Count:=Count +1; Y[Count]:=X[i]
  End For
End.
```

Or a different approach:

Input: $N \in \mathbb{N}$, $X \in H^N$, $T:H \rightarrow \mathbb{L}$,
$f:H^* \times H \rightarrow H$,
$f(x,e):=\begin{cases} \text{push back}(x,e), & T(e) \\ x, & \text{otherwise} \end{cases}$

Output: $Y \in H^*$

Precondition: –

Postcondition: $\forall i(1 \leq i \leq N)$: $Y_i=f(X,X_i)$

```
Multiple_item_selection(N,X,Y):
  Y:=empty
  For i:=1 to N do
    Y:=f(Y,X[i])
  End For
End.
```

We arrive to the final version after inserting the core of function f:

```
Multiple_item_selection(N,X,Y):
  Y:=empty
  For i:=1 to N do
    If T(X[i]) then Y:=push_back(Y,X[i]) else {nothing to do}
  End For
End.
```

10. Partitioning

The previously mentioned operation of conditional addition interpreted for the array will be applied with two conditions now. If $\mathbf{X_i}$ has property **T,** then we add it to array **Y**, otherwise to array **Z.**

Input: $N \in \mathbb{N}$, $X \in H^N$, $T:H \rightarrow \mathbb{L}$

Output: $Count \in \mathbb{N}$, $Y,Z \in \mathbb{N}^*$

Precondition: –

Postcondition:

$$\text{Count} = \sum_{\substack{i=1 \\ T(X_i)}}^{N} 1$$

and $\forall i(1 \leq i \leq Count)$: $T(X_i)$ and $\forall i(1 \leq i \leq N\text{-}Count)$: not $T(X_i)$ and $Y \subseteq (1,2,\ldots,N)$ and $Z \subseteq (1,2,\ldots,N)$

```
Partitioning(N,X,Count,Y,Z):
  Count:=0
  For i:=1 to N do
    If T(X[i]) then Count:=Count+1; Y[Count]:=i
  End For
  CountZ:=0
  For i:=1 to N do
    If not T(X[i]) then CountZ:=CountZ+1; Z[CountZ]:=i
  End For
End.
```

In the above algorithm, independent loops with the same amount of steps or branches with the same conditions can be handled together. This is how we get to the well-known algorithm of sorting in two.

```
Partitioning(N,X,Count,Y,Z):
  Count:=0; CountZ:=0
  For i:=1 to N do
    If T(X[i]) then Count:=Count+1; Y[Count]:=i
               else CountZ:=CountZ+1; Z[CountZ]:=i
  End For
End.
```

Therefore, sequential computing led to copying, copying led to multiple item selection, and multiple item selection led to partitioning; in short, even partitioning originates from the theorem of computing.

11. Intersection (decision in multiple item selection)

The theorem of intersection is essentially the combination of two theorems, that of multiple item selection and that of decision. We have already proven that both derive from the theorem of sequential computing.

Input: $N,M \in \mathbb{N}$, $X \in H^N$, $Y \in H^M$

Output: $Count \in \mathbb{N}$, $Z \in H^{Count}$

Precondition: isSet(X,N) and isSet(Y,M)

Postcondition:

$$Count = \sum_{\substack{i=1 \\ X_i \in Y}}^{N} 1$$

and $\forall i(1 \le i \le Count)$: $Z_i \in X$ and $Z_i \in Y$ and isSet(Z,Count)

Definition: $isSet: H^* \times \mathbb{N} \rightarrow \mathbb{L}$

$isSet(h,n) = \forall i,j(1 \le i \ne j \le n): i \ne j \rightarrow h_i \ne h_j$

```
Intersection(N,X,M,Y,Count,Z):
  Count:=0
  For i:=1 to N do
    If (X[i] in Y) then Count:=Count+1; Z[Count]:=X[i]
  End For
End.
```

```
operator in(x,Y):
  j:=1
  While j≤M and x≠Y[j]
    j:=j+1
  End While
  in:=(j≤M)
End.
```

After inserting the function:

```
Intersection(N,X,M,Y,Count,Z):
  Count:=0
  For i:=1 to N do
    j:=1
    While j≤M and X[i]≠Y[j]
      j:=j+1
    End While
    If j≤M then Count:=Count+1; Z[Count]:=X[i]
  End For
End.
```

12. Union (copying + decision in multiple item selection)

Practically, the theorem of union is the blend of the copying, multiple item selection, and decision theorems. It has already been demonstrated that all three are based on the theorem of sequential computing.

Input: $N,M \in \mathbb{N}$, $X \in H^N$, $Y \in H^M$

Output: $Count \in \mathbb{N}$, $Z \in H^{Count}$

Precondition: isSet(X,N) and isSet(Y,M)

Postcondition:

$$\mathrm{Count} = \mathrm{N} + \sum_{\substack{j=1 \\ Y_j \in X}}^{M} 1$$

and $\forall i(1 \leq i \leq Count)$: $Z_i \in X$ or $Z_i \in Y$ and isSet(Z,Count)

```
Union(N,X,M,Y,Count,Z):
  For i:=1 to N do
    Z[i]:=X[i]
  End For
  Count:=N
  For j=1 to M do
    If not (Y[j] in X) then Count:=Count +1; Z[Count]:=Y[j]
  End For
End.
```

After inserting the function and assigning value to the arrays:

```
Union(N,X,M,Y,Count,Z):
  Z:=X; Count:=N
  For j=1 to M do
    i:=1
    While i≤N and X[i]≠Y[j]
      i:=i+1
    End While
    If i>N then Count:=Count+1; Z[Count]:=Y[j]
  End For
End.
```

13. Conclusion

From the above analysis, we can see that the first 6 theorems (sequential computing, counting, maximum selection, decision, selection, and search) all originate from one programming theorem, namely sequential computing, which is simple addition.

Furthermore, it was proven that 5 additional complex theorems (copying, multiple item selection, partitioning, intersection, and union) derive from the same theorem, that of computing, as well.

It is worth dedicating some time to uncovering why even complex theorems can be traced back to one of the simplest programming theorems. We can derive copying, multiple item selection, and partitioning back to sequential computing because copying applies addition interpreted for the array (that is, we start out from an empty array and while processing the elements of X, we add them to Y, to which we have referred with the operation "push back" in the theorem of copying). Multiple item selection is different from this only in the fact that we apply conditional addition. (The relation is similar to summing and conditional summing as explained in our previous article.) If multiple item selection can be traced back to computing, so can partitioning, and since intersection and union are based on the above, even those theorems are proven to originate from sequential computing.

Among complex theorems, we have not dealt with sorting. It is so because the specific algorithms of sorting can be traced back to different programming theorems, for example minimum selection sort comes from minimum selection and copying, insertion sort comes from search and (while) copying, counting distribution sort comes from counting and copying, etc.

Despite all the above, we are not saying that it is useless to manage these theorems independently. Beginner programmers will recognize the programming theorems while dealing with the different task types, which, in the case of basic theorems, correspond to the above defined six, while with complex theorems, they are most likely the above five. [1,5] With more advanced programmers, it would be worth to base the theorems not on the task types but on the solution types. In that case, decision, selection and search are three sub-types of the same solution principle; therefore, they can be considered as one programming theorem. [2]

This article was supposed to demonstrate that theoretically it is sufficient to check the validity of the first programming theorem because all the rest can be deducted from it. In sum, if sequential computing is correct, so are the others.

Bibliography

1 Péter Szlávi, László Zsakó: *Módszeres programozás. (Methodical programming)* Műszaki Könyvkiadó, Budapest. 1986.

2 Tibor Gregorics: *Programozás – Tervezés. (Programming – Design)* ELTE-Eötvös Kiadó, 2013.

3 Ákos Fóthi: *Bevezetés a programozáshoz. (Introduction to programming)* Fóthi Ákos, 2012. (http://people.inf.elte.hu/fa/pdf/konyv.pdf – Last Retrieved 05/05/2017)

4 Tibor Gregorics: *Programming theorems on enumerator*. Teaching Mathematics and Computer Science 8/1, 2010. DOI: 10.5485/TMCS.2010.0243 (http://tmcs.math.unideb.hu/load_doc.php?p=186&t=abs – Last retrieved 05/05/2017)

5 Péter Szlávi, László Zsakó at al.: *Programozási alapismeretek. (Programming basics)* online course material, (http://progalap.elte.hu/downloads/seged/etananyag/ - Last retrieved 05/05/2017), ELTE Informatikai Kar, 2012.

6 Péter Szlávi, László Zsakó at al.: *Módszeres programozás: programozási tételek.* (*Methodical programming: programming theorems*) Mikrológia 19. ELTE Informatikai Kar, 2008.

7 Péter Szlávi, Gábor Törley, László Zsakó: *Programming theorems have the same origin*. XXX. Didmattech 2017, Trnava University, Faculty of Education, 2017 (http://real.mtak.hu/55421/1/szp_tg_zsl_programming_theorems_have_the_same_origin.pdf – Last retrieved 05/05/2017)

8 Péter Szlávi, Gábor Törley, László Zsakó: *Az összetett programozási tételek is egy tőről fakadnak. (Complex theorems have the same origin too)* Infodidact2017, Webdidaktika Alapítvány, 2017 (http://people.inf.elte.hu/szlavi/infodidact17/manuscripts/zsltgszp.pdf – Last retrieved 02/18/2018)

Authors

SZLÁVI Péter

Eötvös Loránd University, Faculty of Informatics, Department of Media and Educational Informatics, Hungary, e-mail: szlavip@elte.hu

TÖRLEY Gábor

Eötvös Loránd University, Faculty of Informatics, Department of Media and Educational Informatics, Hungary,
e-mail: pezsgo@inf.elte.hu

ZSAKÓ László

Eötvös Loránd University, Faculty of Informatics, Department of Media and Educational Informatics, Hungary,
e-mail: zsako@caesar.elte.hu

About this document

Published in:

CENTRAL-EUROPEAN JOURNAL OF NEW TECHNOLOGIES IN RESEARCH, EDUCATION AND PRACTICE

Volume 1, Number 1. 2019.

ISSN: 2676-9425 (online)

DOI:

10.36427/CEJNTREP.1.1.380

License