THE THINKING TOOLKIT OF PROGRAMMING

SZLÁVI, Péter – ZSAKÓ, László – TÖRLEY, Gábor, HU

Abstract: When a program is being written, a number of cognitive operations are performed in the programmer's brain. It is important to recognize and consciously use them not only when a program is being written, but they are absolutely vital for developing students' problem-solving thinking, which is the most important goal in education. This paper focuses on two of the most significant cognitive operations – linguistic abstraction and analogical thinking–; in addition, it also aims at discussing their characteristics. To introduce them shortly, let us define these concepts in a nutshell. The starting point of language abstraction: language acquisition requires a great degree of abstraction, because the elements and structures of every language are all abstractions. The essence of analogical thinking: when solving a specific task, you always start off with programming tasks that have already been done and thus exist in your mind.

Key words: programming didactics, cognitive processes, problem solving thinking, algorithmic and language abstraction

1 Introduction

While programmers are working hard to solve a programming task, consciously or unconsciously, they use a wide variety of thinking methods. [1] When starting off from the task, they refine it several times to match them their own schemes existing in their brain and based on their personal experience; and – continually, circularly and more precisely – reformulate the task. It means that programming is a sequence of more and more refined (pattern-based) models where you need to get to a stage where the vocabulary (i.e. the set of instruction patterns) of the programming language chosen serves as a basis for the model. (Fig. 1.)

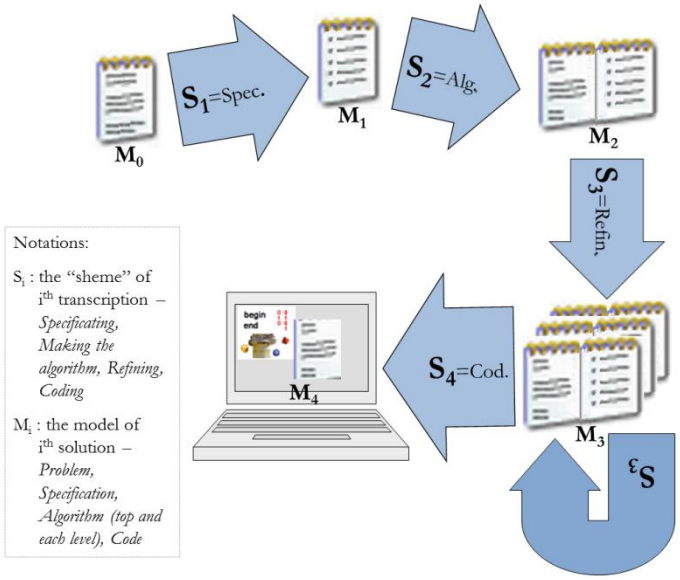

Fig.1: The abstract model of programming

2 Linguistic abstraction

This thinking method is based on three principles: 1) Acquiring a language requires a great degree of abstraction, since the elements (words) of each language and their structures (larger units constructed in accordance with rules like affixed or inflected words and phrases) are all abstractions. Without the knowledge of their syntactic and semantic relationships it is hopeless to correctly use any languages. As we saw 2) programming is the production of a model sequence (from the task to the code); and 3) a unique vocabulary and specific descriptive rules belong to each model.

You should be aware that there are two levels of the syntax: the "concrete" or "written" level (where the spelling and grammatical rules of the language apply), and the abstract level, which is free from the accuracy of description and contains only the essential features (at this level there can be "imperfect or wrong" words and "wrong" phrases; as long as they are clearly decipherable, they are just as good as their perfect synonyms).

We will study three kinds of model languages: 1) wording the task (or specification), 2) design (or algorithmization) and 3) encoding.

It is the formulation of the problem that requires the first abstraction. When wording the problem, first you detach everything irrelevant from the point of view of the task. The important message to be selected from the informally formulated text of the problem is as follows: What are the starting-off data to be defined during the problem solving?; What conditions can be taken into account from the initial data, and how do the resulting data relate to the initial ones? Students' task is similar to solving maths word problems: they must be able to interpret them. The listed above provide guidelines for the interpretation.

Relying on the above mentioned, you can create the vocabulary and structure of the specification language. You can design the abstract syntax with the help of "Pólya's principles":

- What data are available?
- What sort of answer does the task require?
- What do you know about the basic data? Give a few examples!
- What are the relationships between the basic data?
- How do the basic data determine the result data? Give an example for the basic data and the corresponding result data!

This is followed by abstracting the typical attributes of data from anything specific: standardisation i.e. generalisation of the data into a value set, then assigning it to the data. Here you choose one–or when you have complex data, more–suitable set(s) out of some basic sets, and then–if necessary–you create the basic set of complex data using allowed set operations.

We have decided to include this step among language abstraction steps, because a programmer – even without using formal tools [2], in an intuitive way – must be aware of the "linguistic framework" which he is to use when considering data. This linguistic framework as a minimum means the following: the concept of the set, the stock and construction of sets that can be chosen to start with. In Chomsky's sense this language can be regarded as a language, since it can be described with a very simple grammar.

"… I will consider a language to be a set (finite or infinite) of sentences, each finite in length and constructed out of a finite set of elements. … the set of 'sentences' of some formalized system of mathematics can be considered a language." [3]

It is just a "misbelief" that you can write a program without specification! Every programmer either unconsciously specifies, or carries out this activity together with the next step (in the worst case together with encoding); and this is where problems may arise. The knowledge of the "syntax" of this language is just as much required as the knowledge of writing (i.e. the "formal" grammar) in a natural language (cf. specification illiteracy).

The next situation that expects a programmer to perform language abstraction is the process of designing. Then the knowledge of the chosen descriptive tool requires language abstraction skills. The design is done in a "standard" descriptive language. For example:

- block diagram
- structogram

} languages using "drawings"

- pseudocode

languages using "characters"

- …

It is worth mentioning that the above "drawing" algorithm-describing tools are just as languages as the pseudo-code, but their linguistic basic components are graphical, and their grammatical rules are a set of regulated relations of these components.

In data description as well as in algorithmization you must recognise typical and sufficient structures, and it is sensible to develop a clear, but flexible enough language. But what do these attributes mean?

- "**Typical and sufficient**" – the ability to work with it in a natural way, and to foresee all probable problems with its help;
- "**Flexible**" – developers should not spend too much of their energy recalling syntactic constraints that must be respected during the work;
- "**Obvious**" – as time goes by, the idea that has been put on paper must solidly mean the same as it did when it was written.

Many languages meeting these criteria have been developed up to now. Taking into account several criteria, we developed the Hungarian pseudo-coding formalism, which we use to make our algorithms. [9]

It is worth recognizing that programmers apply linguistic abstraction at various levels during the design. The lowest level is the so-called **instruction level**, where the words of the language are the instructions of the algorithmic language, and the structures of the language are defined by the algorithmic language itself. It is common to attach refinements (procedures, functions etc.) to this level, as well. In this case, of course, the vocabulary of the language is dynamically expanded with the names of refinements, and you should include the definition of refinements and the syntax of their application, too. The extension of the previous level is the level of programming theorems. Here belongs the syntax of defining and applying theorems (which conveniently hardly differ from refinements). The increase in level is caused by the higher level of the semantics of the theorems. Theorems are included pre-defined into the language (i.e. they are just as bound as the instruction components of an algorithmic language). The language's becoming more complex is, therefore, negligible, while that of the algorithmic, abstraction content is substantial. The third linguistic level is that of modularization. It also extends the previous level by introducing the concept of the module, giving both the definition and the application syntaxes. In other words, you can say that this is the level of "tool-making". The "speaker" of this language must clearly recognize how a programmer can use the "tool" (i.e. a model to be created) comfortably, efficiently, and safely. Or to put it in another way: this is the

level of producing a "tool-function-universe", during which the module programmer relies on the abstractions of current programmers.

A little diversion to the "drawing algorithmic language":

Herewith, we will mention the graphic "tricks", i.e. graphic language solutions that programmers often use when working. We do not mean the numerous "drawing" tools invented for writing algorithms, such as block diagrams, structograms or Jackson's diagrams, but those figures and "scribbles on the margin" that programmers use to try and catch the essence of an algorithm or map data into the memory. (If you are to face a rather complex linked list, it is almost "indispensable" to make such drawings in order to follow how algorithm parts work.) A graphic sketch – i.e. a "static visualisation" of the problem – is often made in order to better understand simpler tasks, as well. The figure created shows understanding, but it is the activity, the process of drawing that means the real depth of understanding: the order in which the individual items of "artwork" are created is the prefiguration of algorithmic thinking. Its basis is a mental phenomenon similar to the implementation of an algorithm via choreographed dance in Kátai's paper. [8]. What has just been described is also true for programs animating algorithms, which are widespread in education. [10,11] "Scribbles on the margin" have a substantial surplus compared to these, namely, *creating your own thoughts as opposed to recognising other's thoughts*. For example, let us consider the figure pair demonstrating insertion sort, its essence and steps. Just imagine the figure animated correctly; and the algorithm is ready. You should only be formalise it in a traditional algorithmic language!

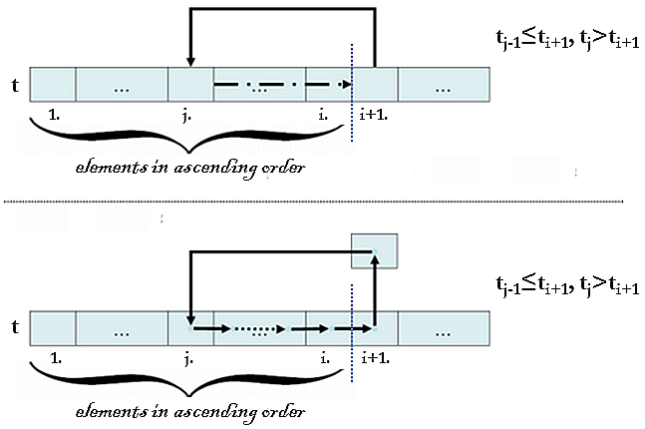

Fig. 2: A "static visualisation" of insertion sort algorithm

The following ideas by György Pólya can easily be adapted to the problem of finding the relevant figure:

"*We start the detailed consideration of such a problem by drawing a figure containing the unknown and the data, all these elements being assembled as it is prescribed by the condition of the problem. In order to understand the problem distinctly, we have to consider each datum and each part of the condition separately; then we reunite all parts and consider the condition as a whole, trying to see simultaneously the various connections required by the problem. We would scarcely be able to handle and separate and recombine all those details without a figure on paper."* [5]

To sum up the above: when designing, in addition to standard algorithm-describing languages, programmers also apply a language that lacks a "regulated syntax" (i.e. they use graphic "help").

The third linguistic challenge is encoding itself. There is no need for serious abstraction here. Of course, only if the programmer has realized that he/she can

mechanically do a substantial part of transplanting into the specific programming language (i.e. encoding), when he/she has a mature algorithm. It is as much a linguistic abstraction activity as the translation activity between related language pairs with a primitive syntax.

This requires that there are elaborate encoding rules available that establish a more or less clear link between the linguistic elements of the algorithm and structures of the programming language. The "more or less" indicates that a certain level of abstraction is definitely involved in the process, although its degree depends on the programmer experience and awareness.

3 Analogy

"There is nothing wrong with thinking according to analogies: analogy has the advantage that it does not bring closure and does not seek a final position; by contrast, induction is disastrous when it has a preconcieved purpose in mind and works toward it, carrying both truth and falsehood along its current"

[Goethe]

This is the thinking method applied most naturally – or you might say – instinctively. Its essence is the following: when the programmer is trying to find a solution to a well-defined (sub)task, first in his/her memory he/she will look for some kind of "related tasks" to start with. Having found one, he/she will recall its "best" solution and then "probe" the matching points between the task to be solved and the model in order that he/she can map them to each other, and thus to apply the known solution. So the scheme of analogic problem-solving is as follows:

- Finding a related task +
- Detecting matching points +
- Recalling the solution of a related task (= related solutions) +
- Updating related solutions.

Let us note that analogical thinking is (also) based on abstraction, e.g. recognizing the parameters of a problem, telling the difference between relevant and irrelevant, or between essential (abstract) and specific.

Regarding the above scheme, two questions arise. On the one hand, "What does *related task* mean?" On the other hand, "In what way are the two solutions related?"

Tasks are related if the way of raising questions is the same regardless of the possibly different attributes of data (e.g. data type). In related tasks data and their relationships play the same role. What tasks a programmer encounter is quite incidental. Thus, the tasks and solution bases can be different, depending on the person. Yet, common experience shows that you can assemble a set of typical problems & questions which will help find one or more analogues to at least 90% of all tasks. [7,9]:

- duplication with item transformation,
- Is there an element with the specific attributes?,
- Give an element with the specific attributes,
- How many elements with the specific attributes are there?,
- Which is the largest element? etc.

The question word can indicate whether you are "raising the same problem" (after Pólya). However, pay attention because they can be misleading! For example: Which? (selection) ≡ "What?" (selection) ↔ "Which?" (Maximum selection) They are not quite the same, are they?!

Having solved a large number of problems, even a self-taught programmer will sooner or later recognize model tasks. However, the detection process can be greatly speeded up if you systematically call your students' attention to these schemes, shortly after introducing them some basic programming vocabulary and some practising. You can do this via examining and analysing some carefully selected tasks, then drawing conclusions from and generalising the experience [7,9], or in a direct, formal way [4]. The former method is primarily used with secondary school students, while the second one is good for university students who have appropriate formal mathematical knowledge.

It is worth quoting György Pólya here and what he stated about the "genetic principle" [6]:

"According to the genetic principle, the learner should retrace the path followed by the original discoverers. According to the principle of active learning, the learner should discover by himself as much as possible. A combination of the two principles suggests that the learner should rediscover what he has to learn."

It means that we adjusted Pólya's genetic principle to programming when – applying programmer-explorers' systematicness – we selected concrete programming tasks that will make students discover the model task.

It is important to notice that the systematic introduction of analogous schemes is absolutely vital because of the limited educational time frame allocated we already mentioned in the Introduction. However, it is a major educational challenge to find the appropriate ratio of "guided" and your own "*aha! insight experience"* based approach. You have to find the ideal for the age group ratio between two extremes: between approaches relying on entire self-discovery, and the one based on the refined, prepared schemes.

The guided or controlled approach has an advantage: it is less time-consuming, whereas the "*aha! insight experience"* approach is known to guarantee deeper learning. The former could be regarded as a lexical approach (and indeed it is if it is badly implemented), while the latter is considered to better develop creativity. Thus time constraints and the development of creativity are confronted antagonistically.

A related solution is a structurally identical algorithm. The necessary components are elementary instructions and instructional construction tools introduced by structured programming.

Instructional constructions :

a) instruction-sequence,
b) branch(es),
c) loop(s),
d) defining refinement

Elementary instructions:

e) assignment (with a formula containing a function call as a possible refinement),
f) call of a refinement considered elementary.

D) and f)–and often e) as well–follow the well-known "top-down planning" principle in the algorithmic language.

It is thought-provoking experience that novice programmers usually come to recognize analogies by noticing similarity in the codes, which means that instead of the substantial, semantic similarity, it is a formal, syntactic one that leads them to abstract thoughts.

Conclusion

In this paper we have had a close look at two thinking toolkits: language abstraction and analogy. When describing the first tool, we wrote about three model languages: formulating

the task (how to get from an informally worded task to a formal specification), the design (where we determined the four main characteristics and the levels of an algorithmic language) and encoding. When discussing the algorithmic language, we highlighted graphical solutions, where programmers "visualize" the process of problem solving for a deeper understanding, and thus reach algorithmic thinking.

The tools of analogy include model tasks (model algorithms) that at least 90% of the problems to be solved can be linked to; thus one can say they are "relatives". It means that searching for related tasks (schemes or models), detecting the connection between the specific task and the model, and then recalling the solution of the model algorithm will lead the programmer to the solution of the specific task.

The thinking tools discussed are concepts that do not have precise boundaries, but mutually expand into each other's "spheres of interest". Yet it is useful to distinguish them. Outlining concepts allows us to study them independently, which enable us to map the mechanism of thinking. An important consequence of this is that you will be able to elaborate methods in order to develop how certain thinking tools can be used. They will mainly bear interest in programmers' performance, but in general they will have a positive effect on problem-solving thinking, as well.

Bibliography

[1] SZLÁVI, P., *A programkészítés didaktikai kérdései.* ELTE, 2005. Available: http://www.inf.elte.hu/karunkrol/szolgaltatasok/konyvtar/lists/doktori%20disszertcik%20adatbzisa/attachments/32/szlavi_peter_tezisek_hu.pdf . [Accessed on: May 28, 2016].

[2] SZLÁVI, P., *Programozási tételek specifikációja*. 1996. Available: https://www.researchgate.net/publication/303582337_Programozasi_tetelek_specifikacioja. [Accessed on: May. 28, 2016].

[3] CHOMSKY, N., *Syntactic structures.* Mouton Publishers, The Hague, Paris, 1957.

[4] FÓTHI, Á., *Bevezetés a programozáshoz.* Tankönyvkiadó, 1983.

[5] POLYA, G., *How to Solve It: A system of thinking which can help you any problem*. Princeton University Press, 1945.

[6] POLYA, G., *Mathematical Discovery on Understanding, Learning and Teaching Problem Solving, Combined Edition.* John Wiley & Sons, 1981.

[7] SZLÁVI, P., ZSAKÓ, L.: *Módszeres programozás: programozási tételek*. Mikrológia19, ELTE TTK Informatikai Tanszékcsoport, 2004.

[8] KÁTAI, Z., TÓTH, L.: *Technologically and artistically enhanced multi-sensory computer programming education.* In Teaching and Teacher Education. 26, 2010, 2, 244–251.

[9] SZLÁVI, P., ZSAKÓ, L., *Módszeres programozás*. Műszaki Könyvkiadó, 1986.

[10] TÖRLEY, G., *Objektum orientált programozás tanítása vizualizációs eszközökkel*. In InfoDidact'2012 Konferencia, 2012. Available: http://people.inf.elte.hu/szlavi/InfoDidact12/Manuscripts/TG.pdf. [Accessed on: May 28, 2016].

[11] TÖRLEY, G., *Algorithm visualization in teaching practice.* In Acta Didactica Napocensia, Vol. 7. No. 1., pp. 1-17., 2014., Babes-Bolyai University, Didactics of Exact Sciences Department and the Pedagogy and Applied Didactics Department, Cluj-Napoca, Romania ISSN 2065-1430

Lectured by: Sándor Király, Dr. Ph.D.

Contact address:

Péter Szlávi, Dr. Ph.D.,
Department of Media and Educational Informatics, Faculty of Informatics, Eötvös Loránd University, H-1117 Budapest, Pázmány P. sétány 1/C, Hungary,
phone: +36-1-372-2500 , e-mail: szlavip@elte.hu

László Zsakó, Doc. dr. hab. Ph.D,
Department of Media and Educational Informatics, Faculty of Informatics, Eötvös Loránd University, H-1117 Budapest, Pázmány P. sétány 1/C, Hungary,
phone: +36-1-372-2500 , e-mail: zsako@caesar.elte.hu

Gábor Törley, Dr. Ph.D.,
Department of Media and Educational Informatics, Faculty of Informatics, Eötvös Loránd University, H-1117 Budapest, Pázmány P. sétány 1/C, Hungary,
phone: +36-1-372-2500 , e-mail: pezsgo@inf.elte.hu