ACTA DIDACTICA NAPOCENSIA

Volume 10, Number 4, 2017

THE COGNITIVE TOOLKIT OF PROGRAMMING – ALGORITHMIC ABSTRACTION, DECOMPOSITION-SUPERPOSITION

Péter Szlávi, László Zsakó

Abstract: As a programmer when solving a problem, a number of conscious and unconscious cognitive operations are being performed. Problem-solving is a gradual and cyclic activity; as the mind is adjusting the problem to its schemas formed by its previous experiences, the programmer gets closer and closer to understanding and defining the problem. The primary cognitive operations the programmer uses to set up refining models are: language abstraction, analogy, algorithmic abstraction, decomposition-superposition, conversion, intuition, and variation.
In our paper we are shedding new light on algorithmic abstraction, while explaining the es-sence of decomposition-superposition, a key element-pair in the cognitive toolkit of program-ming.

Key words: programming [~68N01], didactics [~00A35], systemic thinking [~97C30,~97C70], cognitive toolkit [~97Q30], algorithmic abstraction, decomposition, superposition

1. The Cognitive Toolkit of Programming

As a programmer when solving a problem, a number of conscious and unconscious cognitive operations are being performed. Problem-solving is a gradual and cyclic activity; as the mind is adjusting the problem to its schemas formed by its previous experiences, the programmer gets closer and closer to understanding and defining the problem. We can say that programming is the series of the problem's models, based on finer and finer schemas, which need to be elaborated until the basis of the model is the vocabulary, or the set of instruction schemas, of the given programming language (Figure 1).

The primary cognitive operations the programmer uses to set up refining models are the following:

- language abstraction [1],
- analogy [1],
- algorithmic abstraction,
- decomposition-superposition,
- conversion,
- intuition,
- and variation.

The programmer applies these tools a number of times, for a number of reasons. This is why we will have to mention the given tools more than once.

In our paper we are shedding new light on the well-known notion of **algorithmic abstraction**, while explaining the essence of **decomposition**–**superposition**, a key element-pair in the cognitive toolkit of programming. Our article closely connects to the argumentation started in [1].

Received October 2017.

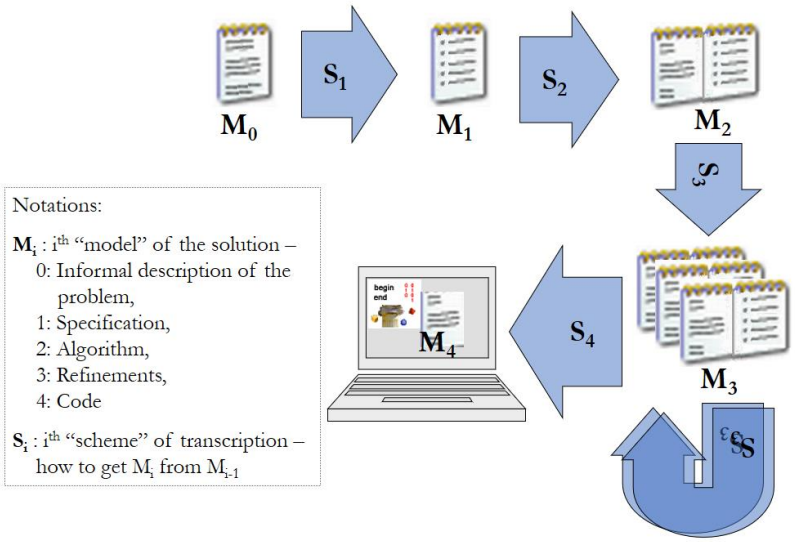

Figure 1. *The abstract model of programming*

2. Algorithmic abstraction

The programmer makes use of the cognitive tool of **generalization** many times and in manifold ways while programming. One of these ways is **analogy-based thinking**, which we have examined extensively in [1,2] in connection with problem schemas. We have found that analogy-based thinking can be facilitated if we elaborate and introduce well chosen schemas. Since schemas are the results of abstraction, we need to analyze them from this aspect as well.

If we want to save memory, we can create the starting set of analogies by generalizing and abstracting numerous **specific** problems that are similar, together with their solutions. That is, we highlight their similarity while eliminating their difference, for the sake of finding the "common ancestor," the **abstract** core. [1]

The elements of the analogy base are an abstract problem and the optimal solution. This thought leads us to the notion of **programming theorems**: abstract problem + abstract algorithm + verification. We do not need to keep verification in mind all the time; it is enough to perform it only upon introducing the theorem. Just as scientists do when using certain theorems: they check first the conditions and the goal of the research, then accept the truthfulness of the theorems to use without further examination. Needless to say, in public education verification is performed without any sort of formalism, just by common sense; however, it needs to take place, as it facilitates the understanding of how the algorithm and the problem are related. Table 1 provides an example to the definition of a programming theorem. [2/p.22]

We are focusing on the programming theorems which are based on abstract problem types. Other methodologies might build similar but less programming theorems on abstract solution types [6].

[1] Note that by this we are offering a remedy for one of the critical claims of PISA, namely, an overly lexical-based education.

Table 1. *The structure of programming theorems – the Theorem of "Decision."*

Part of the programming theorem	Example
Abstract problem – specification:	**Decision**(H^*,*F*(H,**L**)):**L** **Input:** N∈**N**, X∈H^*, T:H→**L** **Output:** YES∈**L** **Precondition**: N=Length(X) **Postcondition:** YES = ∃*i*∈[1..N] : T(X_i)
Abstract solution:	**Constant** MaxN:Integer(???) **Type** THk=**Array**(1..MaxN:TH) **Procedure** Decision(**Constant** N:Integer, X:THk, **Variable** YES:Logical): **Variable** i:Integer i:=1 **Loop while** i≤N **and not** T(X(i)) i:+1 **Loop end** Yes:=i≤N **Procedure end.**
Verification:	Due to its lengthiness we ignore for now. [3/pp.19-21]

Comments for the Table 1:

- The headline of the specification is given as a function. The function's domain is constituted by the sets in the parentheses. (The direct product of these sets is the domain.) The set following the colon is the codomain. The description comes in handy mainly with the combination of theorems, because it helps to exclude some of the theorems that could be combined, and this way we can reduce the number of "analogies" to consider. Ultimately, it makes the thinking process more efficient.
- The specification contains subtleties which can be used when going further. [2/pp.25-27] For example, the series "X∈$\mathbf{H}^*$" could have been expressed as: "X∈$\mathbf{H}^N$", which creates a clear connection between two parameters (N and X), but in the next phase of programming it would, falsely, lead us to define "**Type** THk=**Array**(1..N:TH)". Regarding the missing N-X relation, it can be corrected in the precondition, as we have done so.
- Programming theorems are well known in the methodology of programming. The 17 most important ones (and some special versions) are defined with their specific formalism in the literature: [4, 5]. The language that describes the problem, as shown by the above example, is based on the elements of **graph theory** and **first-order predicate calculus**. Let us add to the formalism that in public education students only need to read, not write in this language. That is, it is only its conscious use, not the formalism itself that matters at start.
- When we teach specification in advanced programming, the focus is on the abstract mathematical concept of **series**. A number of other linguistic versions have been created for making specifications. For example [7] uses **graphs** instead of series. We have decided for the central role of series for didactic reasons: the concept of the **series is closer to the notion of algorithmic array, often used by novice programmers**.

When using programming theorems, the programmer **abstracts**, naming and describing the subproblem, which implies a refinement, then **looks for a theorem analogy**, which could be used for the subproblem. Most often the programmer applies abstractions more basic than theorems. Let us see them.

When the programmer uses the strategy of **designing from top to bottom**, introducing a new procedure or function into the algorithmic language (refinement), the language itself is extended

(compare with language abstraction [1]). Enriching the vocabulary of a language entails two things. For one, it requires matching the "form" or **syntax**, and the expectation or **specification** of the concept, which is abstraction itself; for two, it calls for defining the meaning or semantics of the concept, which is its representation-implementation. In programming this form of building is traditionally **algorithmic abstraction**; that is, it has a narrower interpretation. Most often, the process advances gradually; that is, the solution needs to be **refined step by step**, until it reaches the level of the programming language (see decomposition). This is how we create the abstract **concept hierarchy**, quoting Dijkstra that fits the problem.

In our experience, understanding a concept occurs on two levels: first, the novice programmer realizes to have acquired the tool to shorten algorithms, then when applying it, the programmer wants to start parameterizing. It is interesting that students notice its benefit to shorten algorithms faster than its main advantage: that it enables the practice of the "divide and conquer" principle, thus allowing a better use of their mental energy. Some further difficulties regarding the concept of parametered refinement are the distinction and the proper interpretation of the formal and current parameters. Thus, the second level of understanding, as mentioned before, is in reality to overcome the abstraction challenges posed by parameters.

It is worth to mention another algorithmic thought related to abstraction. Keeping our method-ological principle of "generalizing the problem" [3/p.97] is an abstraction step in the design. Its essence is to make a program, fitting not just a specific problem but a wider circle of problems, by generalizing the concepts (primarily the constants) of the problem.

In the simplest case, we can define the solution by replacing the constants with symbols. The symbolic data can become additional input parameters but they can also be symbolic constants (which we call internal or latent parameters). Even in the latter case, we have the benefit that changing the value of the constant can be done with one move and with complete safety.

For example, in the problem where we need to "list basketball team players taller than 210 cm," we might want to generalize

- using the constant or additional input parameter `Height` (with 210 as initial value), instead of using just 210, and
- even though the problem mentions a basketball team, setting a specific quantity, we can declare the number of players by a constant or a variable called Number.

We can do the same with types "induced" by the problem. Namely, our program will work with more general basic sets. Naturally, generalization can trigger obvious efficiency questions as well. For example, regarding the above problem the generalizations will affect these types:

- we can work with integers, or scalars instead of expressing height in cm,
- we can use diverse series-types, not just arrays, to define the data of the players.

The third case of problem generalization is weakening the precondition. Perhaps the simplest case is when we trace back a selection problem to a programming theorem, annulling the precondition which would require the element type in search to be part of the input series.

Therefore, the generalization examined above challenges the programmer less in abstracting, and much more in finding the ideal balance between **problem-level generalization** and **design-level efficiency**.

We can blend problem-abstraction with refinement-abstraction. Let us see the above problem. Instead of the relation of height constraint, we can use a logical function that stands for the generalized property (that is, which generalizes the relation). This way, our solution can be applied, with only minor changes, in any other problem where it is just the player's "property" that differs from the above.

In this subchapter we have covered the following fields of generalization:

- **problem generalization** – problem parameters, types
- **abstraction of solution schemas** – programming theorems

- more strictly defined algorithmic abstraction – refinements and parameterization

Through algorithmic abstraction we can recognize the general operation of cognitive abstraction. Based on [8/p.18]:

- **specific** program solution to **specific** problems →
- **instruction and data abstraction** – abstract instruction types: sequence, branch, loop; data types: basic types; complex types: record, array, file →
- their "**re-specification**" – specified instruction and data (types) →
- their **abstraction** – programming patterns: programming theorems; **type constructions**: modules, parametered types →
- their "**re-specification**" – "specific" theorems →
- their **abstraction** – algorithm level: **parametered mappings**; data level: **generalized series** or set from array →
- their "**re-specification**" – algorithm level: specific parametered theorem-functions, applications → **theorem combination = function combination** → code level: the template is created, class pattern, generic notion

The same applies to the metamorphosis of the data concept according to [9].

3. Decomposition-superposition

Decomposition means to **break down** a complex problem into the unity of more basic problems. When we reverse the relation, we talk about **superposition**: in this case, we **build up** a complex problem from basic problems. The essence, however, is the same: to relate the complex problem and the basic ones.

About the well-known psychological limits of the human mind, psychologist László Mérő claims: the short-term memory of the human mind can store maximum 7±2 schemas or cognitive structures per person. [10/p.12] Simply speaking, this means that when designing we cannot handle plans whose complexity activates more schemas than the above number. Keeping this in mind, we have to realize that the principle of **building from top to bottom**, or "**d**ivide and conquer" in mundane words, applied by structured programming, addresses this very human weakness, only without numbers. Consequently, it is a principle inherent to programming.

The essence of this cognitive operation is to repeatedly redefine the problem with a refining op-eration set until it reaches the instructions of the algorithmic language. Each level offers a complete solution to the problem, but the schemas (or as we have used it so far: the refinements) of the given level break down into 5-9 more basic micro-schemas (or: refinements, or defined operations). We can see this in the third schema transformation of the programming model in Figure 1, Chapter 1.

Comen et al. [11/p.10] approach the issue like this:

> "*The divide-and-conquer paradigm involves three steps at each level of the recursion:*
> *Divide the problem into a number of subproblems.*
> *Conquer the subproblems by solving them recursively. If the subproblem sizes are small enough, however, just solve the subproblems in a straightforward manner.*
> *Combine the solutions to the subproblems into the solution for the original problem.*"

Two comments need to be added here:

On the one hand, merging the subproblems (or, in psychological terms, the schemas) means se-lecting the appropriate instruction management. The subproblems need to be ordered on the relevant level of recursion. Their relations need to be determined, for which one of the following – or their combinations – are available:

- **sequence** – several subproblems in a coordinate relation, each of which need to be executed;

- **branch** – several subproblems with independent conditions, of which only those with true conditions need to be executed; they are in subordinate relations compared to the whole branch;
- **loop** – the subproblems, comprising the core of the loop, need to be executed "depending on the condition"; they are in subordinate relations compared to the whole branch.

On the other hand, subordination and coordination are relevant for many reasons. First, the relation influences how the algorithm is described. Second, it also affects its complexity. When we extend the program by coordinating instructions, we increase its complexity in a linear or additive way, but when we subordinate them, like in a complex structure, complexity is multiplied exponentially. It is nicely expressed by a well-known complexity measure, depth complexity, which expresses the complexity of a substructure of a program by assigning to it, in the index of the power of two, the number of higher order structures the specific substructure is embedded into [12/p.103], and the overall complexity of the program will be their sum. Thus, **by applying refinements** the **complexity** of the program **will decrease**, which is not just what we expect but what numbers will show.

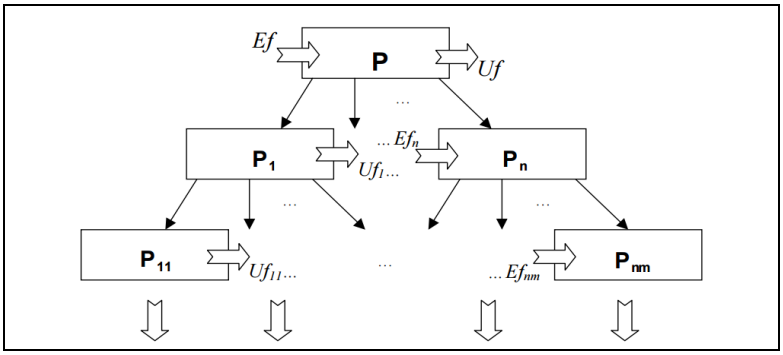

Figure 2. *Program design following the "top-down" approach and specifications.*

To choose from the three problem-construction modes, novice programmers can turn to the principle of "process according to structure" [13/p.46] for help. This principle creates a tight bond between the data structure to process and the algorithm-structure managing the process.

Table 2. *The principle of "process according to structure."*

Data structure		Algorithm-structure
scalar	→	function or procedure
direct product	→	sequence
union	→	branch
data collection	→	loop

Applying the principle means that

1. we determine the input and output data of the problem or the subproblem, using the terms from [4], the input and output superstructure belonging to the problem,
2. we select one of them as "guiding" data structure,
3. we assign it to one of the data structures offered by the principle, and
4. we select the corresponding algorithm-structure based on the principle.

Seemingly, using the principle only assists in drafting the algorithm. Occasionally, however, some further information regarding the specific problems can also be built into the assigned algorithm. In Chapter 2 of the above quoted [13] literature, we scrutinize the benefits of the principle.

Many times **superposition** is conceived of as **building from bottom up**. Programmers engage in such activities when they are creating a "programming environment" for a new problem-family, or a greater

problem world. The aim is to define a higher level "language" which helps solve special problems more easily. Typically, it means defining a new type-construction (like graphs, trees, or data sets with special structures), or specifying some routine library, and their representation-implementation. In such a case, the goal is not to solve a problem with a program, but to create the toolset of the program. If the aim of a program is traditional (decompositional) programming, such as creating problem-solving tools, then the superpositional approach aims to provide the toolkit necessary for these problem-solving tools. In other words, they are not interchangeable. Especially considering that we use decomposition even when creating certain more complex "tools."

4. Summary

In our paper we have examined two of the 7 cognitive operations (methods) as identified by us; we wrote about algorithmic abstraction and the complement pair of decomposition-superposition. We use algorithmic abstraction in a broader sense than traditionally, while decomposition and superposition are understood as two strategies that complement, not replace, each other when designing more complex systems.

The cognitive tools we have covered do not have clear-cut boundaries but can extend into the domain of the others. Nevertheless, it is useful to distinguish them for the sake of **independent** scrutiny so we can explore each **cognitive mechanism**. An important consequence of this is that we can develop **methods** that **facilitate the use of the given cognitive tools**. Primarily, they will make programming more efficient, but very often they will **have a positive impact on problem–solving thinking** as well.

References

[1] P. Szlávi, L. Zsakó, G. Törley (2016), The Thinking Toolkit of Programming, *in Proceedings of XXIXth DidMatTech 2016, "New methods and technologies in education and practice" Conference, Budapest (2016)*, pp.55-62. http://didmattech.inf.elte.hu/wp-content/uploads/2016/08/SzP_TG_ZsL_The-Thinking-Toolkit.pdf (last retrieved: 11/06/2016)

[2] Péter Szlávi (2005), *A programkészítés didaktikai kérdései* (*The didactic questions of programming*), ELTE. http://www.inf.elte.hu/karunkrol/szolgaltatasok/konyvtar/lists/doktori%20disszertcik%20adatbzisa/attachments/32/szlavi_peter_tezisek_hu.pdf (last retrieved: 11/06/2016)

[3] Péter Szlávi (1999), Programok, programspecifikációk (Programs, program specifications), *in Informatika a felsőoktatásban'99, pp. 576-582.* http://people.inf.elte.hu/szlavi/ProgModsz/Progspec.pdf (last retrieved: 11/06/2016)

[4] Péter Szlávi, László Zsakó (2002), *Módszeres programozás: Programozási bevezető* (*Methodological programming: Introduction.*), Mikrológia, ELTE.

[5] Péter Szlávi (2001), *Programozási tételek – összefoglaló (Programming theorems – summary).* http://people.inf.elte.hu/szlavi/ProgModsz/Prtetel.pdf (last retrieved: 11/06/2016)

[6] Tibor Gregorics (2013), *Programozás 1. kötet Tervezés* (*Programming, Volume 1: Planning*), ELTE Eötvös Kiadó.

[7] Ákos Fóthi (1983), *Bevezetés a programozáshoz* (*Intro to programming*) Tankönyvkiadó.

[8] Imre Fényes (1980), *A fizika eredete* (*The origin of physics*), Kossuth Kiadó.

[9] Péter Szlávi, László Zsakó (2004), *Programozási nyelvek: Alapfogalmak* (*Programming languages: Basic notions*), Mikrológia, ELTE.

[10] László Mérő (1989), *A mesterséges intelligencia és a kognitív pszichológia kapcsolata* (*The connection between artificial intelligence and cogni-tive psychology*), Tankönyvkiadó.

[11] T. Cormen, Ch. Leiserson, R. Rivest (1997), *Introduction to Algorithms, Volume 1,* Műszaki Könyvkiadó.

[12] László Zsakó (2003), *Módszeres programozás: Hatékonyág* (*Methodological programming: Efficiancy*), Mikrológia, ELTE.

[13] Péter Szlávi, László Zsakó (2004), *Módszeres programozás: Adatfeldolgozás* (*Methodological programming: Data processing*), Mikrológia, ELTE.

Authors

Péter Szlávi, PhD, Faculty of Informatics, Department of Media- and Educational Informatics, Eötvös Lorand University, Budapest, Pázmány Péter sétány 1/C., Hungary, e-mail: szlavip@elte.hu

László Zsakó, PhD, Faculty of Informatics, Department of Media- and Educational Informatics, Eötvös Lorand University, Budapest, Pázmány Péter sétány 1/C., Hungary, e-mail: zsako@caesar.elte.hu