



Automated programming, symbolic computation, machine learning: my personal view

Bruno Buchberger¹

Accepted: 27 March 2023 / Published online: 10 October 2023
© The Author(s) 2023

Abstract

In this note, I present my personal view on the interaction of the three areas Automated Programming, Symbolic Computation, and Machine Learning. Programming is the activity of finding a (hopefully) correct program (algorithm) for a given problem. Programming is central to automation in all areas and is considered one of the most creative human activities. However, already very early in the history of programming, people started to “jump to the meta-level” of programming, i.e., started to develop procedures that automate, or semi-automate, (various aspects or parts of) the process of programming. This area has various names like “Automated Programming”, “Automated Algorithm Synthesis”, etc. Developing compilers can be considered an early example of a problem in automated programming. Automated reasoners for proving the correctness of programs with respect to a specification is an advanced example of a topic in automated programming. ChatGPT producing (amazingly good) programs from problem specifications in natural language is a recent example of automated programming. Programming tends to become the most important activity as the level of technological sophistication increases. Therefore, automating programming is maybe the most exciting and relevant technological endeavor today. It also will have enormous impact on the global job market in the software industry. Roughly, I see two main approaches to automated programming:

- symbolic computation
- and machine learning.

In this note, I explain how the two approaches work and that they are fundamentally different because they address two completely different ways of how problems are specified. Together, the two approaches constitute (part of) what some people like to call “artificial intelligence”. In my analysis, both approaches are just part of (algorithmic) mathematics. The approaches, like all non-trivial mathematical methods, need quite some intelligence on the side of the human inventors of the methods. However, applying the methods is just “machine execution” of algorithms. It is misleading to call the application “machine intelligence” or “artificial intelligence”. The analysis of the two approaches to automated programming also suggests that the two approaches, in the future, should be combined to achieve even higher levels of sophistication. At the end of this note, I propose some research questions for this new direction.

✉ Bruno Buchberger
bruno.buchberger@jku.at; buchberger.bruno@gmail.com

¹ Research Institute for Symbolic Computation (RISC), Johannes Kepler University of Linz, Linz, Austria

Keywords Automated programming · Symbolic computation · Automated reasoning · Machine learning · Artificial intelligence · Artificial general intelligence · Pretrained large language models · Software industry · Programming assistant.

Mathematics Subject Classification (2010) 68T01 · 68W30 · 68N15

1 Automated programming

The process of programming starts from problems in some reality and creates programs that solve the problems in appropriate finitary models of this reality (i.e., models that can be stored in a computer). From the model, the solutions can be interpreted back in the problem reality. Programming has matured from being a kind of magic at the beginning to a decent engineering discipline called “software engineering”. Programming is at the center of automating all kinds of processes. In a natural jump to the meta-level, we can try to automate the process of programming itself (or at least part of it) to make the process more reliable, provably correct, faster, more flexible, more economic, and ultimately less (brain) labor intensive.

More formally, for a given programming language, we write:

$$p[d] := \text{the result (a data value) of applying program (algorithm) } p \\ \text{(of the language) to input data } d.$$

Data are numbers and other elementary objects and compositions of objects (like matrices, trees, etc.).

For a given logic language (like predicate logic), we write:

$$q[d] := \text{the result (a truth value) of evaluating formula } q \\ \text{(of the language) on input data } d.$$

Roughly, a *problem* can be described (“specified”) by a formula q (in some logic language) that explains, for all data d and e , whether or not e is an admissible answer to the problem input (“problem instance”) d . One often presents problems using the following wording: Given d , find e such that $q[d, e]$.

A program p (in a given programming language) *solves* problem q (described in a given logic language) iff

$$\text{for all } d, q[d, p[d]]. \text{ (The “correctness statement” for program } p \\ \text{with respect to problem } q.)$$

Problem specifications of the above form are called “explicit” (i.e. they are not in mutual dependence with other problems) and “general” (i.e. there is the “for all” quantifier at the beginning of the correctness statement). In practice, problem specifications are not formulated in formal logic languages but in natural languages. A formal language is a language whose syntax and semantics is defined in a meta-language. A natural language is a language whose syntax and semantics is learned by using the language in contexts that are shared by the people who speak the language. When problem specifications are formulated in natural language, the universal quantifier in the correctness statement may be hidden. (Problem specifications in natural language are also the starting point of automated programming by pretrained large language models, see the last sections of this note.)

In most cases, problem specifications are explicit. (However, there are important classes of algorithmic problems that cannot be described in explicit form. For example, a canonical simplifier A for an equivalence relation P cannot be described in this form. More generally, for example, the specification of operations on data structures by axioms or the construction of algorithmic isomorphic representations of mathematical domains are not explicit. Here, we cannot go into more details about this.) Later, in the section on machine learning, we will also consider problem specification that are not “general” but “by example”.

Programming can be easy for some problems. (*Example*: The problem of deciding whether a given object is in a given list.) Programming may need some well-trained mind. (*Example*: The problem of finding shortest paths in graphs. A famous algorithm for this is Dijkstra’s algorithm.) For some problems, finding an algorithm has been open for many years but, finally an algorithm was found. (*Example*: The problem of deciding whether a non-linear multivariate polynomial system over the complex numbers has a complex solution. Solved 1965 by the author.) For some problems, finding an algorithm is still open. (*Example*: Collatz’ halting problem: $f[n] := \text{if}[\text{even}[n], n/2, 3n + 1]$; $f[1] := 1$.) For some problems, finding a “good” algorithm is still open. (*Example*: The problem of deciding whether a Boolean system has a solution, called the SAT (satisfiability) problem). For some problems, finding an algorithm was proved to be impossible. (*Example*: The problem of deciding whether a non-linear multivariate polynomial system over the integer numbers has an integer solution. Proved impossible in 1971 by Yu. Matiyasevich.)

By the work of millions of “programmers” (algorithm inventors, mathematicians) over the past centuries and decades: we have thousands of algorithms for fundamental problems (made available in well-organized algorithm libraries); millions/billions of complex programs for easy and more and more difficult problems are composed from the fundamental algorithms and are made available in “software systems” for thousands of “applications”; these programs are executed on billions of computers (that become faster, bigger, smaller, cheaper, ... at an impressive rate); a growing flood of data (in/out) is processed and produced; by the internet, these computers, the software systems, and the data are globally interconnected forming kind of a “global computer”; the global computer interconnects the billions of users and becomes the digital memory and the digital processing power of a “global society” in a digital model of the world; the global society is the global programmer and the global user of the global computer.

One may argue that “programming” (finding provably correct algorithms for problems) is the most intelligent human activity: Programming is the abstract form of all problem solving (= composing solutions for problems from available solutions for sub-problems), and programming is the essence of mathematics and is essentially mathematics. However, in a certain sense, by “automated programming”, low-level programming will be trivialized and millions of programmers are in danger of losing their jobs. On the other hand, programming will remain essential and probably the most essential activity on higher and higher levels of abstraction. (In my view, this is a practical formulation of the incompleteness theorem of Kurt Gödel, who proved this insight 1931, i.e., ten years before the first universal computer was built!)

On the way from a problem specification or a collection of problem specifications to a program or software system that solves the problem(s) there are many creative steps. Each of these steps can be handled ad hoc for the particular problem(s) at hand by a mathematician, computer scientist, programmer. Each of these steps, however, can also be considered a problem on “data” like software requirements, programs, algorithm schemata, verification conditions, etc. Algorithms for these steps on the meta-level constitute the field of “automated programming”.

A simple example: In a calendar software system, typically, we will need one unit to check whether a proposed new appointment collides with one already stored.

In such a simple example of a problem specification, one normally expects that the natural language specification is sufficient for a moderately experienced programmer to come up with a program that solves the problem (hopefully correctly). Recently, I used this simple problem for some experiments on the attitudes and capabilities of programmers. I found that most programmers immediately understood that the input data in this problem are the stored appointments and the new appointment and that the output is a message like “collision” or “no collision”. They also naturally decomposed the approach into a loop (of some kind) through all the stored appointments and a collision test between the new appointment and the appointment considered inside the loop. Furthermore, it was immediate that the collision test should be done by formulating a condition between the four time moments s_1, e_1, s_2, e_2 , i.e., the start time and the end time of the first and the second appointments, respectively, where we may assume that $s_1 < e_1$ and $s_2 < e_2$.

“After some thinking”, the problem was then described by most programmers by a sentence like this: “The two appointments characterized by s_1, e_1, s_2, e_2 collide iff $s_2 \leq s_1 \leq e_2$ or $s_1 \leq s_2 \leq e_1$.” Now it is clear that this “specification” of the problem is, basically, already the solution algorithm. Only some transformation into the syntax of the programming language used is necessary. No powerful algorithm verification method or algorithm synthesis method seems to be necessary in such a simple case.

As simple as the example is, it is not simple enough to guarantee the avoidance of serious programming mistakes. In my systematic experiments with programmers, quite a few came up with the following specification/code: “The two calendar entries characterized by s_1, e_1, s_2, e_2 collide iff either $s_2 \leq s_1 \leq e_2$ or $s_2 \leq e_1 \leq e_2$.” This specification is “incorrect” because it does not include the case $s_1 < s_2 < e_2 < e_1$, which of course “everybody” would also consider as a collision, even a “particularly heavy one”. I put “incorrect” in quotation marks because, at the very first stage of uttering a requirement, the “customer is always right”. Maybe, he really wants what he tells, or he believes that what the programmer tells him is what he wants! Alternatively, one may consider the following discussion as a way to find out what the customer “really wants”. For this, we start a little “earlier” and just say: “The two calendar entries characterized by s_1, e_1, s_2, e_2 collide iff the time interval $[s_1, e_1]$ intersects with the time interval $[s_2, e_2]$.” Now we should discuss the notion “intersects” and might agree on the following: “The two calendar entries characterized by s_1, e_1, s_2, e_2 collide iff there is a time moment x such that $s_1 \leq x \leq e_1$ and $s_2 \leq x \leq e_2$.” (Note that there is an existential quantifier in the definition!) In this form, we can send the condition as input to a quantifier elimination algorithm and we will get the result “ $s_2 \leq s_1 \leq e_2$ or $s_1 \leq s_2 \leq e_1$ ” or something equivalent. (Please try this out using a symbolic computation system like Mathematica! It is worthwhile!) Hence, this simple example shows that already in the very early stage of discussing and clarifying even seemingly simple requirements some non-trivial systematic/formal reasoning is involved. It is the main question of automated programming how much of this reasoning can be automated.

The second task in writing a program for the above collision problem is an appropriate organization of the loop over the stored appointments. It turned out that, of course, all programmers immediately proposed a trivial loop from the first to the last appointment. Some, not all, in a second thought, proposed that the stored appointments should be kept sorted according to the start (or end) times and then an efficient binary search could replace the trivial loop. I will discuss this part of the programming task below.

In more complicated examples of problems, the task of finding an algorithm that solves the problem becomes much more challenging and the possibility for committing (not only

“sloppiness” but) logical mistakes gets drastically bigger and more dangerous. Therefore, “automated programming” is a worthwhile undertaking: For a particular class of problems \mathcal{C} , instead of

finding a program p_1 for problem q_1 in \mathcal{C} such that, for all d , $q_1[d, p_1[d]]$,
 finding a program p_2 for problem q_2 in \mathcal{C} such that, for all d , $q_2[d, p_2[d]]$,
 ...

try to find a “general program synthesis algorithm” \mathcal{G} such that, for all q in \mathcal{C} ,
 for all d , $q[d, \mathcal{G}[q][d]]$.

Note: the general algorithm \mathcal{G} has the problem q as an input parameter! (One also says: “ \mathcal{G} is on the meta-level of the q in \mathcal{C} ”.)

In the subsequent sections, I discuss the two main approaches to automated programming: Automated programming by “symbolic computation” and automated programming by “machine learning”. Together, the two approaches form (part of) what people like to call “artificial intelligence” or “machine intelligence”. This terminology, however, is the source of a big misunderstanding: The algorithms on the meta-level of problem-solving, like any other algorithms, are the creation of human intelligence. Since some of the problems q on the object level might have been quite hard and needed quite some “human intelligence” and the execution of the algorithm \mathcal{G} on the meta-level on q does not need any intelligence, one is tempted to say that “the machine \mathcal{G} is intelligent”. However, the only thing one should or could say is that “the human who invented \mathcal{G} is intelligent”.

In other words: There is no race between “human intelligence” and “machine intelligence” but, rather, there is always a race between “human intelligence spent for finding programs for individual problems in a class \mathcal{C} of problems” and “human intelligence spent for finding a meta-program \mathcal{G} for all the problems in the class \mathcal{C} ”. In other words: Human intelligence on the higher level aims at making human intelligence on a lower level superfluous!

Again in other words: It is the goal of mathematics (computer science, ...) to trivialize itself! Thus, personally, I like to expand the abbreviation “AI” as “algorithmic intelligence”, i.e. the human intelligence for finding programs on higher and higher levels!

The individual brain power for problem-solving may vary quite significantly. However, in my opinion, the following “law” holds:

Brain Power Constancy Hypothesis (“The Reflexion Principle”): The human brain power for problem-solving did not change over the past, say, ten thousands years and it will not drastically change over the next ten thousand years. The spectacular increase in the problem-solving capacity and the dramatic acceleration in the increase is a consequence of applying the constant brain power in higher and higher rounds of going from the object level to the meta-level. In one round, the objects of the previous round become the actors on the next round. (I call this transition “reflexion”).

I discuss the reflexion principle in more detail and more generality in the first part of [1].

2 Automated programming by symbolic computation

In 1984, Academic Press London issued a call for designing a new journal for a new field that had emerged approximately since 1960. Various names were used for this field: computer algebra, symbolic and algebraic manipulation, analytic computation, formula manipulation,

computation in finite terms, computation in closed form, symbolic computation, and others. In response to this call, I submitted a proposal to Academic Press for a “Journal of Symbolic Computation” (JSC). My proposal was selected and my clarification of the scope of “symbolic computation” also formed the Editorial of the journal, see [2]:

I defined “*symbolic computation*” as the area that deals with algorithms on symbolic objects, and I proposed “*symbolic objects*” to be defined as finitary representations of infinite mathematical entities (or, in other words, finitary objects with infinite semantics). Here, “*finitary*” means “storable in a computer memory”. For example, finitely many generators with finitely many relations between words formed from the generators form a finitary object that may represent an infinite group (or a “large” finite group, i.e., a group whose number of elements is much larger than the size of the finitary representation). Algorithms can only work on finitary objects, and the flavor of “symbolic” is exactly the point that we want to solve problems on infinite (or “large”) mathematical entities by finding algorithms that work on finitary (small), “symbolic”, representations of these entities.

In essence, the editorial of the JSC still characterizes its scope. Pragmatically, in the editorial of the Journal of Symbolic Computation, I identified three main areas for symbolic computation: *computer algebra*, *automated reasoning*, and “*automatic programming*”. I also emphasized that all aspects of these areas should be in the scope of the Journal of Symbolic Computation: mathematical theory on which symbolic algorithms can be based, the algorithms with their correctness proofs and complexity analysis, the details of the implementation of the algorithms, languages and software systems for symbolic computation, and applications. The three main branches of symbolic computation consider three important classes of “symbolic objects”:

- *computer algebra*: symbolic objects that represent algebraic entities like terms that represent functions, differential operators, etc., or finite relations that represent residue class structures;
- *automated reasoning*: symbolic objects containing (quantified) variables that are considered as statements on (infinite) domains;
- *automatic programming*: symbolic objects containing variables that are considered programs that define processes on potentially infinitely many inputs.

(Of course, these three sub-areas of symbolic computation are intimately connected and, in some precise way, even embedded in each other. The distinction between the three areas is only a matter of “flavor”.)

In other words, symbolic objects are finitary objects that have “semantics” attached to them where, typically, the semantics is “large”, even infinite, not tangible by computers, whereas the symbolic objects are “small”, finitary, tangible by algorithms. Any field of mathematics can be studied under the “symbolic” aspect: In any field of mathematics, if we want to solve problems by algorithms, we have to find finitary representations for the objects in the field. Finding suitable finitary representations, by itself, may be a difficult – sometimes provably impossible – mathematical problem: Before embarking on deeper algorithmic questions in some area of mathematics, deciding whether two symbolic objects represent the same abstract mathematical object and finding “canonical” finitary representatives for symbolic objects may already be very difficult (sometimes provably impossible). By finding representations of mathematical objects in any field of mathematics, the field becomes “algebraic”, and problems in the algebraic disguise of the field, essentially, become combinatorial problems. Thus, very sketchy, one may say: *symbolic computation, ultimately, is the “combinatorization” of all of mathematics via finitary representations of infinite mathematical entities.*

It is a common misunderstanding that symbolic computation is the trivial side of mathematics, i.e., some people believe that, whereas “pure” mathematics lives in complex spaces needing deep and difficult thinking, algorithmic mathematics (which must be “symbolic” in the above sense) “just puts everything to the computer” and presses the start button. The truth is that the “just” needs more and deeper mathematics than a version of mathematics that allows non-algorithmic constructions for problem-solving like the unlimited set quantifier, infinite summation, infinite unions, transition to residua class domains, etc. (A trivial example: In “pure” mathematics, a Gröbner basis for given ideal generators can be “easily” found by “just” taking the ideal generated by the generators. However, the definition of the ideal generated by generators involves an infinite set construction!) Hence, with some provocation, in my view, mathematics only *starts* at the moment when we try to solve problems by “symbolic computation”.

In this section, we assume that all the descriptions on the way from a problem specification/requirements to an algorithm/software are expressions (i.e., finitary objects) that describe or at least try to describe something “in general terms”, i.e., they describe infinitely many possible cases using variables ranging over infinite domains, i.e., they are symbolic objects. Hence, the entire programming process and various parts and variants of the process are symbolic computation problems: program synthesis; program verification; analysis of termination, complexity and other properties of programs; specification extraction from programs; generation of inductive assertions for programs; program transformation; program repair; program optimization; model-based programming; component-based programming; etc.

The literature on symbolic methods in automated programming is huge, see for example the review paper [3]. As an example of a symbolic automated programming method I describe here briefly my “Lazy Thinking” method for general algorithm synthesis. The method is implemented in the *Theorema* system, [4]. Roughly, the Lazy Thinking method is an algorithmic implementation of my personal advice to humans how to invent an algorithm p for a given problem q . The overall strategy of Lazy Thinking is:

- Preparatory step (i.e., a step that is not specific for the particular problem q but refers to the environment of all notions occurring in the specification q of the problem): Understand the problem “completely”, i.e. collect and prove all knowledge that you can possibly get on the notions that occur in the specification q . This includes also a “library” of programs which are already proven to be correct with respect to certain specifications.
- Program schemes: Consider known fundamental ideas of how to structure programs in terms of sub-programs. Try one scheme P after the other.
- For the chosen scheme P , try to prove (automatically): for all d , $q[d, P[d]]$. (The first proof attempt will normally fail because nothing is known about the sub-programs in P .)
- From the failing proof, construct (automatically) such specifications for the sub-programs B, \dots occurring in P that will turn the failing proof into a successful proof.
- Iterate inductively over the sub-programs B, \dots and their specifications until you arrive at specifications for which a program is available in the program library.

This strategy can be turned into an algorithm if the necessary (inductive) proofs and the constructions of the specifications for the sub-programs can be automated. The automation of inductive proving is far advanced, and for the construction of specifications for sub-programs from failing proofs I developed an algorithm in [5].

Example of a Program Scheme (“Divide and Conquer”):

$$\left(P[d] = \begin{cases} S[d] & \Leftarrow \text{isBasic}[d] \\ M[P[L[d]], P[R[[d]]] & \Leftarrow \text{otherwise} \end{cases} \right)$$

P is the unknown program. S, M, L, R are unknown sub-programs.

Example of a program which we want to synthesize: Synthesize a program “sorted” for the problem “isSortedVersion”, i.e. a program that satisfies:

$$\text{for all } d, \text{ isSortedVersion}[d, \text{sorted}[d]].$$

Preparatory step: Collect knowledge on the problem. Of course, we start with the definition:

$$\text{isSortedVersionOf}[list1, list2] := \text{isSorted}[list2] \wedge \text{isPermutation}[list1, list2]$$

$$\text{isSorted}[list] := \text{forAll}[\{i, \text{length}[list] - 1\}, list_i \leq list_{i+1}]$$

...

etc. (approx. 20 formulae are needed here.)

We now start an attempt to prove the correctness theorem, i.e. the assertion that the program scheme P solves the problem isSortedVersion. The proof will stop at a certain stage indicating that it would need some knowledge on the (unknown) sub-programs S, M, L, R.

The following rule produces knowledge for an unknown sub-program m that is sufficient to continue with the proof:

Collect the temporary assumptions $T[x0, \dots, P[...], \dots]$ and

the temporary goals $G[x0, \dots, m[P[...]]]$ and produce the specification:

$$\forall_{x, \dots, y, \dots} \left(T[X, \dots, Y, \dots] \implies G[X, \dots, m[Y]] \right)$$

Here, x0 symbolizes the Skolem constants that are generated during the proof. Note that the specification generated does not anymore contain the program name P, i.e., the specification is a specification (a requirement) for the sub-program m!

This rule is the essence of my Lazy Thinking method. Lazy Thinking, automatically (in approx. 1 minute on a laptop using the *Theorema* system), finds the following specifications for the sub-algorithms that guarantee the correctness of the above algorithm (scheme):

$$\begin{aligned} &\forall_x (\text{isBasic}[x] \implies P[x] = x) \\ &\forall_{y,z} \left(\begin{array}{l} \text{isSorted}[y] \\ \text{isSorted}[z] \end{array} \implies \begin{array}{l} \text{isSorted}[M[y, z]] \\ M[y, z] \approx (y \times z) \end{array} \right) \\ &\forall_x (L[x] \times R[x] \approx x) \end{aligned}$$

(Here, \times denotes concatenation of lists and \approx denotes that two lists contain the same elements equally often.) The specification expresses the fact that any sub-programs that satisfy this specification turn the above divide-and-conquer program scheme into a correct program for the sorting problem. Note that the specifications generated are not only sufficient, but they are also natural! They specify merge algorithms M and pairing algorithms L and R! What we got automatically is a problem reduction! We either have suitable M, L, R in our program library, or we can apply the Lazy Thinking principle again for M, L, R and their specifications until we arrive at programs in our library. (Library: Programs with their specifications and, maybe, other knowledge on the programs.)

Thus, by the Lazy Thinking Method, entire hierarchies of provenly correct algorithms can be generated in arbitrary domains. Now the question is: Can we automatically synthesize

algorithms for non-trivial problems? I answered this question positively by showing, in [6], that with the Lazy Thinking method, it is possible to invent the essential idea of the algorithm which I invented as a Ph.D. student in 1965 for solving the canonical simplification problem for non-linear multivariate polynomial ideals. In the conclusion of this note, I will propose how this research could be continued to develop a combined symbolic/machine-learning approach to automated programming. It should be clear that, with Lazy Thinking, we could generate for example various kinds of loops for the above calendar program.

3 Automated programming by machine learning

Undoubtedly, in the past two decades, artificial intelligence has produced impressive results. This is due to the fact that, by the drastically increased computational power of current computer systems and the availability of huge databases of “labeled” data, a couple of difficult, long-standing and important problems have been solved impressively well, for example, machine translation of natural languages.

Mentioning artificial intelligence, for me, raises some nostalgia because, when I founded the Research Institute for Symbolic Computation (in 1985), for some time, I was torn between using “symbolic computation” or “artificial intelligence” as the main notion in the name of the new institute. At that time, bringing symbolic computation under the umbrella of artificial intelligence was quite tempting and quite common: For example, finding symbolic integrals was considered an “artificial intelligence” task like playing chess, with lots of heuristics. Correspondingly, the most comprehensive symbolic computation software system at that time, MACSYMA, had “MAC” (= Machine Aided Cognition) in its name! And, of course, implementing heuristics is still an important approach for improving the practical efficiency of methods for symbolic computation problems. However, in 1985, I deliberately decided against having “artificial intelligence” in the name of my institute since I wanted to emphasize the logical, mathematical, formal approach to problem-solving over the psychological, experimental aspect, which some people (then and now) believe goes “beyond mathematics”. I will go deeper into analyzing the relationship between symbolic computation and artificial intelligence later in this note.

Amazingly, there is still a lot of mystery, nebulosity, and misunderstanding around what artificial intelligence (AI) is and why it is (is believed to be) essentially different from all other computational approaches. This nebulosity is all over the place: in politics, the media, even science, and, of course, with the man on the street. At times, I have the impression that even quite some researchers in the AI area do not have a very clear picture of the distinctive characteristics of AI when compared with other computational approaches. Also, labeling a project with AI may have a beneficial effect when it comes to funding, societal respect, political influence, etc. Thus, it is tempting to keep the notion ambiguous. What amazes me even more is that the nebulosity about the essence of AI did not disappear since the field started in the middle of the fifties. I remember talks of AI evangelists around 1980, i.e. in the “first wave of AI research”, who believed and spread that “AI can solve hard problems that cannot be solved by mathematics”. And still, when I participate in political discussions about the importance of mathematical education (in the sense of training mathematical thinking), I hear the argument that, actually, the ability to do mathematics will be less and less important because “tedious” mathematical thinking, in the presence of “artificial intelligences” (plural!), will not be necessary anymore and that we should teach the youngsters more “creative” things than mathematics.

Now all such statements may be true or false according to which notion of artificial intelligence one has in mind. To clarify this notion, I want to distinguish three possible characterizations of AI:

Hard Problems:

Artificial Intelligence may be described as the field that tries to solve problems that, at a certain historic moment, are considered to be “hard” in the sense that they apparently need a decent amount of (human) “intelligence” to solve them. For example, playing chess or finding symbolic integrals, at some historic moment, were considered as needing human intelligence. Algorithms (invented by humans!) that finally were able to solve these problems were then (and still are) considered to be the result of “AI research”.

Now, in my opinion, this definition of the notion of AI is quite shallow. It is the natural flow of science and technology that we can solve harder and harder problems automatically, i.e. by algorithms. However, from some point on, people think that now “algorithms are taking over”, “artificial intelligence is replacing humans” etc. forgetting that this happened and happens already since centuries and that this is the very goal of science and technology. And, of course, whatever the methods behind automation were and are, we humans should stay in control and decide how far we let problems be solved and decisions be taken by algorithms. Anyway, the notion of a “hard” problem is relative and “hard” problems for which an algorithmic solution was finally found very soon are considered to be “easy” by the consumer. For example, car drivers nowadays take the functionality of a navigation system for granted. Some thirty years ago, the current functionality of navigation systems would have been considered unbelievably intelligent. In fact, the stack of scientific findings and algorithmic techniques involved in a navigation system for guiding a driver from *A* to *B* is quite deep.

In my opinion, one should not use the notion of “artificial intelligence” for “finding algorithms for hard problems” but rather continue to call this just “mathematical, algorithmic solution of hard problems”. Attaching the label “AI” to algorithms depending on whether they solve hard or easy problems is more a question of marketing rather than a logically sound distinction.

Simulate the Brain:

A completely different view (and branch) of artificial intelligence is artificial intelligence as the science of understanding and simulating biological structures that show “intelligence”, notably the human brain. Historically, this type of AI research was one of the origins of the field of AI that started, maybe 1943, with the investigations of W. McCulloch and W. Pitts, who introduced a simple mathematical model of the functionality of a neuron. Of course, understanding and simulating the most complex biological systems, commonly considered to display “intelligence”, is a fascinating and relevant undertaking. Well, why not call this type of research “artificial intelligence” in the same way as a technical realization of the phenomenon of flying could be called “artificial flying”. Also, studying biological structures (like the brain, swarms of animals, and the evolution of life on earth) motivated some of the algorithmic methods that today are called “AI methods”, see the next paragraphs.

“Intelligent” Methods:

The third approach to characterizing artificial intelligence is by specifying certain algorithmic methods as “intelligent”. These algorithms would constitute the area of “artificial intelligence”. I hope I do not overlook something important, but my impression is that, essentially, “machine learning” is the only such method or, better, class of methods that has not already been around before the term “artificial intelligence” was coined. The many

other algorithmic methods that are often labeled as “AI methods”, like automated reasoning, semantic networks, graph search, expert systems, etc., in my view, are algorithmic methods that are not specific to AI and, in fact, are all “symbolic computation” methods. They are, so to speak, usual algorithmic methods and were also applied to “hard” problems that, for some reason, got the label “AI”, see above.

In my view, machine learning methods cannot be characterized by the way how they work but, rather, by the way how problems are specified. As we have seen in the previous sections, the fundamental part of algorithm and software development is the transition from a given problem specification q to an algorithm (program, system) P that solves the problem for any admissible input. If the steps for going from a problem specification to a solution algorithm are done by a human, this is just the “usual business” of mathematics/informatics. If the problem specification and the intermediate stages from the problem specification to the program are expressed by symbolic expressions that characterize the stage “in general terms” and finding the intermediate steps is (partially) supported by algorithms this is what I call “symbolic computation” methods for “automated programming”. How and when does “machine learning” come in and why, if at all, is this different from “usual” mathematics and “usual” (maybe quite sophisticated) symbolic computation for automated programming?

The point is that, in many situations, when we want to specify a problem, we do not have a specification “in general terms” available (i.e. an expression with variables that explains the problem for all infinitely possible input and output cases). For example, let’s consider the seemingly simple problem of deciding whether a given English sentence contains information of the type “somebody cooperates with somebody else”. (In the literature, such problems are called the “relation extraction problems”.) An algorithm for this problem should produce the answer “YES” and the two “somebodies” if such information is contained in the input sentence and “NO” otherwise. Now, of course, before trying to invent such an algorithm, we will ask: What exactly do you mean by “cooperate”? Among the English-speaking community, under the natural assumption of a long experience of using English in thousands of different situations, it would be natural the start to explain “cooperate” in terms of a couple of other notions like “working together”, “having a common goal”, ... Oh, “having a common goal” may not always be sufficient for speaking about “cooperation”! One may have a common goal but fight against each other. Thus, “supporting each other”, etc., should be added. Some more subtle details should be explained, some other things excluded. A long list of sentences explaining the meaning of “cooperate” would be necessary. Then one could, in the attempt to find an algorithm for this little problem, try to put these numerous explanations into algorithmic rules (assuming that we already have access to a powerful grammar parsing algorithm for all of English). As a result, we would hope that this rule system will be able to do the job. For example, if we now would input “Peter and Ann found a way to help each other to pass the exam”, the algorithm should answer (“YES”, “Peter”, “Ann”). If we would input “Peter and Ann passed the exam on the same day”, it should answer “NO”. Should it really answer “NO”? Shouldn’t it instead answer “DON’T KNOW” or “COULD BE” or “COULD BE BUT NOT EXPLICITLY MENTIONED”?

I now want to explain what, in my view, is the essence of the machine learning approach. For this, we need not at all bother about what “learning” is. I just consider those methods that, over the years, have been named “machine learning” methods. The common feature of these methods is not how they proceed but the type of specification of the problems to which they are applied: Namely, they all are applied to problems of the kind above where a spelled-out complete specification is not possible or, at least, not feasible. Now, the fundamental idea of machine learning for solving such problems is:

- Instead of spending (“endless”!) time trying to specify the problem by a huge number of general definitions, cases, rules, etc., one spends the time giving a huge number of examples of input instances together with the answers. (In this paper, we consider only “supervised learning”.) In this context, the answers are called “labels”.
- One sets up an algorithm (program) from a certain class of relatively simply structured programs (like the class of neural networks, the class of hyperplanes in a high-dimensional space, the class of nested if-then-else expressions, etc.) with some constants c_1, \dots, c_n (for example the weights at the inputs of neurons in neural networks) in the algorithm left unspecified. For each choice of numerical values for the c_1, \dots, c_n , the algorithm would produce an answer for each admissible input for the problem, i.e. the algorithm would be an attempt to solve the problem. (One can view this also as another application of the above idea of “program schemes”.)
- One uses techniques of mathematical optimization (or other experimental techniques, for example, techniques that mimic biological evolution) to change the initial values for c_1, \dots, c_n iteratively until the answer of the algorithm to more and more inputs from the set of labeled data would give the answer specified by the label (or an answer “close enough” to the label). In the jargon of machine learning, this iteration is called “training a model”.
- One stops the iteration on the c_1, \dots, c_n when sufficiently many answers are identical (or close) to the labels. Practically, at the beginning of the whole operation, one partitions the set of labeled input into a “training set” that is used for the iterations and a “test set” on which the algorithm with the current values for the c_1, \dots, c_n – which in the jargon of machine learning is called the “trained model” – is tested. (Some people like to call a trained model an “artificial intelligence” so that one can have various “artificial intelligences” in one’s problem-solving arsenal. In my opinion, this is one of the reasons for all the “philosophical and political” confusion in the area of “artificial intelligence”.)
- The trained model (which is a program) is then used for all future possible inputs with the hope (!) that the outputs are what the person who described the problem (by examples) would probably expect. (If some “general” knowledge is available on the problem domain, this “hope” may be replaced by “good reasons” or even by a provable statement.)

The impressive success of this approach in the past two decades hinges on three ingredients:

- a huge amount of *mathematical* research on good and, partly provably convergent, techniques for improving the algorithm parameters c_1, \dots, c_n ; such research was already available in the first phase of AI between 1960 and 1980, but it did not convince because of the next two ingredients were not available,
- huge corpora of labeled data; for example, in the spectacular application of machine translation, a huge amount of “labeled data” is now available in the form of files that contain original texts and their translation - by humans - to some other language,
- high-performance computing; in successful applications, the number of iterations of the machine learning steps for determining suitable c_1, \dots, c_n , the number n , and the computational effort in each step is huge and is only manageable by recent computers.

More formally, we consider problems q , for which no general specification (by a formula with variables ranging over the infinitely many possible input and output values) can be given but for which the following is possible:

- one can give a (huge) number of examples of input/output pairs $(d_1, e_1), \dots, (d_n, e_n)$ for which $q[d_1, e_1], \dots, q[d_n, e_n]$ holds;

- for any additional d and e , one would be able to decide whether $q[d, e]$.

Can one still find a program that solves that problem, i.e., a P such that, for all infinitely many possible d , $q[d, p[d]]$? The answer is: In principle, “no”. Of course, one can give an algorithm P (e.g., “table-look/up”) that gives output e_i for input d_i , for $i = 1, \dots, n$. However, how can we be sure that $q[d, p[d]]$ will also be satisfied for all other d , since q is actually not specified for all d .

Only under some additional assumptions, can one find a suitable program:

- One may know or “one wants to believe” that the algorithm should have some simple form (the form of a “program scheme”). For example, one may believe that the program should be a linear function, a polynomial function, or a nested if-then-else expression, or . . .
- One may be satisfied if the program produces outputs that are close to the expected result within some “tolerance”.

In principle, this approach is not at all new. Examples of historical “learning from examples” problems are: Given points in the plane, find the coefficients c_1, \dots, c_n of a polynomial that goes through the given points (the interpolation problem). Given a function with some properties on differentiability, an interval, and a distance, find the coefficients c_1, \dots, c_n of a polynomial that is closer to the function than a given distance everywhere on the interval (approximation problem). Given points in the plane, find the coefficients of a straight line that minimizes the distance to all points (regression problem). Given a function with certain differentiability properties, find the coefficients c_1, \dots, c_n of a finite Fourier (Taylor, Walsh, . . .) approximate of the function, etc. Finding programs for many classes of such problems, many such “program schemes” and many such notions of “tolerance”, is an old and rich area of mathematics with many different variants.

Seen in this way, machine learning falls neatly into the “automated programming” view: It is the method of choice in cases where the problem is not specified by general (formal or natural language) statements but, instead, is specified (or can be specified) only by a (huge) number of examples of admissible input and desired output. In the case of general specifications of problems, the transition from the problem to a solving algorithm, in principle, is a reasoning process that is executed by humans or, in the symbolic computation approach, is a reasoning process (partly) supportable by symbolic computation methods. In the case of problems that are specified only by examples, this process can still be automated by the machine learning approach.

From the simple summary of the machine learning approach I gave above, one important deficiency of the machine learning approach should be clear: The program we get for a given problem just does the job of delivering (in sufficiently many cases) desired answers. However, in general, no reason can be given why, for example, the particular neural network that decides whether or not two sentences express cooperation between two persons mimics certain fundamental insights about the environment both persons share as their semantics. This is, in fact, similar to the situation in the historical predecessors of “learning from examples”: The Fourier analysis just does the job of finding an optimal Fourier sum. In the example, where the function to be represented is the frequency spectrum of a musical tone, the representation by a finite Fourier sum has a reasonable “explanatory” power: The tone is composed of tones and overtones that occur in the physical “music” world (for example, when picking the strings of a guitar). However, if a Fourier representation of some arbitrary other function is executed, there will be some outcome, but there may not be any reasonable interpretation of what this representation means in the reality from which the function is taken.

The problem of weak explanatory power in the “models” (programs) created in machine learning is well known. Lots of research was recently started to extract “meaning” and “reasons” from such models. This research area is called “explainable AI”.

The spectacular success of machine learning is particularly impressive in natural language problems like machine translation or composition of texts on given topics. Very recently, ChatGPT of OpenAI, [7], demonstrates the dramatic leap in sophistication that has been achieved in the past few years by machine learning for processing knowledge expressed in natural languages. The pretrained language model GPT-3 on which ChatGPT is based is a neural model with 175 billion parameters, trained on a very large corpus consisting of publicly available datasets such as CommonCrawl 1, WebRText dataset, two internet-based books corpora, and English Wikipedia. For the topic of this note, ChatGPT adds another dimension: ChatGPT can produce programs (in a wide range of programming languages) specified “in general terms” in (a wide range of) natural languages.

Example 1 Enter a natural language specification of the above collision test problem to ChatGPT: “Write a program for the following task: Given a set of appointments in a calendar and a new appointment, check whether the new appointment collides with any of the appointments in the calendar. An appointment consists of a start time, an end time, and a text describing the topic of the appointment. Code in your favorite programming language.” In a few seconds, ChatGPT will answer something like the following. (I say “something like the following” because, next time ChatGPT will be asked the same question, the answer may be different since it may have learned from your reaction to the current answer.)

Sure, here is a solution in Python:

```
class Appointment:
    def __init__(self, start, end, text):
        self.start = start
        self.end = end
        self.text = text

def check_collision(appointments, new_appointment):
    for appointment in appointments:
        if (appointment.start < new_appointment.end and
            appointment.end > new_appointment.start):
            return True
    return False

# Example usage
appointments = [Appointment(10, 11, "Doctor's appointment"),
                Appointment(12, 13, "Lunch with John")]
new_appointment = Appointment(11, 12, "Meet with Jane")

if check_collision(appointments, new_appointment):
    print("Collision detected")
else:
    print("No collision detected")
```

This solution creates an `Appointment` class to represent an appointment, with `start`, `end`, and `text` as its attributes. It then defines the `check_collision`

function which takes in a list of appointments and a new appointment and checks whether there is a collision by iterating through the list of appointments and checking if the start or end time of any of the appointments fall within the start and end time of the new appointment. If a collision is detected, the function returns `True`, otherwise it returns `False`.

I hope this helps! Let me know if you have any questions.

The answer is not only correct, it is also documented. (Note, however, that the explanation of the collision condition in the documentation is logically wrong whereas the formulation of the condition in the program is correct! The logical mistake in the documentation is exactly the one which I encountered a couple of times in my experiment with human programmers, see above. I find this really somewhat hilarious: ChatGPT gets a vague natural language specification of a problem as input which is logically correct since it avoids to go into any detail about what “collision” means, but relies on common sense. ChatGPT produces a logically correct program for the problem, which is really an impressive achievement. However, it then produces a natural language documentation that is logically incorrect! Please compare this with the potential of quantifier elimination (a powerful symbolic computation method), which is able to start from the “common sense” explanation of the notion of collision (which involves a hidden existential quantifier, as I have explained above) and, from there, produces the correct condition on the start and end times of the two appointments.

Anyway, ChatGPT is aware of it and speaks modestly about its own automated programming capabilities: If one asks ChatGPT about this, it answers:

It’s worth noting that GPT-3’s code generation capabilities are not perfect, and the generated code may require some manual editing or tweaking to be fully functional. However, it can still be a powerful tool for developers and non-technical users alike, allowing them to quickly prototype and test ideas without having to write code from scratch.

In my experiment with ChatGPT, see [8], I continued with the question: “Have you got any idea how the computation time of the program `check_collision` in your previous answer could be improved significantly?” In a dialogue, ChatGPT came up with the idea of keeping the stored appointments sorted and using a binary search approach. In fact, in [8], I challenged ChatGPT by asking all the questions I normally ask informatics master’s candidates for a fellowship and, impressively, ChatGPT did much better than the average students and ranked among the top candidates. Only if it came to proving simple things about, for example, the O -notation, it became clear that ChatGPT just tried to imitate usual proving patterns even in cases where the formula to be proved was actually wrong. For the details, see [8]. This can be easily explained because the approach to programming (and proving) of ChatGPT is machine learning from millions of examples. In the case of programming, this means: learning from millions of examples of problem specifications (in natural language) labeled by programs written by human programmers for this specification. Millions of training data for this are available, for example, in GitHub, where programs are stored together with their documentation in natural language. In other words, ChatGPT is training the automated programming capability as a translation process between two languages: A natural language for specifying problems and a programming language for formulating the program.

The automated programming capability of ChatGPT is also methodologically interesting for the following reason that shows the power of the Brain Power Constancy Hypothesis

(“The Reflexion Principle”), i.e. the power of always trying to go to the meta-level as soon as sufficiently many instances on the object level can be mastered:

For a problem like the sorting problem, it would not make sense (at least not at first sight) to use machine learning for solving it because it can be specified “in general terms” (i.e. by a “symbolic object”), see the problem specification above. Why should one give, say, thousand examples of input/output pairs to the sorting problem and then set on a neural network (i.e., mathematically, a nested in-then-else expression with some unknown numerical weights) and train it with the thousand input/output pairs to get the weights fixed to some values that will make the neural network produce the desired outputs in thousand examples hoping that, from now on, it would produce the sorted version of an input list in all infinitely many possible cases? Instead, a human programmer or a symbolic reasoning method (for example, my Lazy Thinking approach) will produce a sorting program (a symbolic object) together with a proof (another symbolic object) that the program will meet the specification for all infinitely many possible input lists.

As explained above, there are millions of problems, however, for which we do not have a specification in general terms, like the relation extraction problem discussed above or, for example, the problem of predicting the time of a malfunction of a brake in a car from the values of some sensors in millions of past time moments. For any such problem q , in order to apply a machine learning method (like neural networks) for composing a program, we need a (huge) number of examples of input/output pairs (“labeled inputs”) $(d_1, e_1), \dots, (d_n, e_n)$ for which $q[d_1, e_1], \dots, q[d_n, e_n]$ holds.

When using ChatGPT for automated programming, the machine learning method implemented in ChatGPT was also trained on finitely many (billions) of labeled inputs. However the labeled inputs were of the kind $(q_1, p_1), \dots, (q_n, p_n)$ where both the q_i and the p_i are symbolic (!) objects, namely (natural language) problem specifications q_i “in general terms” and programs p_i (in a formal programming language) that appear together as specification and corresponding program in some program library. Note that, among the q_i , there may appear, for example, the specification (or various specifications) of the sorting problem. Note also that the specifications of problems by huge number of examples and their corresponding program (“model”) generated by machine learning do *not* appear among the q_i ! (As a “gedankenexperiment”, think about what would happen if ChatGPT trained its automated programming power by using in its training set also problems that are not given in general terms but by huge labeled training sets. We would get something like a very general “universal model”. I do not know whether this gedankenexperiment, or at least an attempt at it, has already been carried out. It could be considered a path toward AGI (artificial general AI)).

In other words, the automated programming power of ChatGPT does not come from particularly deep insights into the peculiarities of the programming process (insights into the semantic or reasoning aspects of programming) but, instead, from applying a successful machine learning method for generating programs from finite labeled input sets to training sets on the meta-level consisting of specifications labeled by programs!

In the frame of the analysis of this paper, I summarize: The machine learning approach can be well subsumed under the general target of (semi-) automating the software development process (“automated programming”). It can be viewed as a numerical, rather than a symbolic, approach to automatic programming, for example, as an interpolation approach. What is new is that some new “program schemes” are used, e.g., neural networks, which mathematically can just be seen as nested if-then-else expressions (maybe with some “recurrence links” in richer versions of neural networks). Since the choice of these program schemes was motivated by biological neural networks, some people believe/wish/think that these networks and the algorithms (“models”) produced by them contain/mimic/grasp/“are” intelligence or

“intelligences”. However, the algorithmic process based on these terms works (well or badly) independently of whether or not one considers these networks and the models produced as being intelligent or not. Also, the nested if-then-else expressions describe “real” biological neural networks only in a very superficial way. In fact, what happens in real neural networks and, in particular, the human brain and, actually, in the entire human “system” is much more complicated: Current biological research is far from “understanding the brain”. (Anyway, biological research is much more advanced than, say, thirty years ago and advances at high speed.) The people who invented the neural network approach to “machine learning” and are refining it permanently, though, are quite intelligent. Thus, neural networks based “machine learning” is another example of “human algorithmic intelligence”. In contrast, the machines (computers) that execute the approach are as dull as ever.

Implicitly, I expressed this view already in the early days of AI, see [9]. At the “Spring School on AI” in Teisendorf (Germany), 1982, I contributed a long article summarizing the most important “symbolic” methods for automating the algorithm/software development process that were known at that time. And we had long, intensive, and quite controversial discussions at this conference on the question of whether AI is something that goes beyond mathematics. As you may guess, my answer then was “no” with essentially the arguments which I gave above. In my hectic years of researching methods for “algorithmic intelligence” and research management, I never found the time and occasion to spell out these arguments in a paper. Thus, I am grateful that I am given the opportunity here.

Personally, when speaking to people who do not (want to) understand the timeless, universal, always new, creative power of mathematics, I like to use the term “*algorithmic intelligence*” for what we are doing: Algorithmic intelligence is the *human* intelligence that produces algorithms for more and more challenging problems in all areas of human activity. By an algorithm, an infinite class of individual problem instances can then be treated by a completely *unintelligent* machine. People who do not really understand what is going on may *believe that these machines display “intelligence”*. The algorithmic intelligence - by reflection, i.e. jumps to higher and higher meta-levels - also provides more and more sophisticated algorithms for producing algorithms. Symbolic computation is a jump to the meta-level of programming (that can be iterated), and so is machine learning (which we saw can also be iterated!) The incompleteness theorem of Kurt Gödel (1931), in a somewhat liberal interpretation, shows that this tour through higher and higher levels of algorithmization has no upper bound.

4 Conclusion: automated programming by symbolic computation and machine learning

In this note, I explained that automated programming is the algorithmic production of programs that solve specified problems. Then I identified symbolic computation and machine learning as (the) two approaches to automated programming and explained why they are fundamentally different: Symbolic methods are applied to problems specified “in general terms”, i.e., by formal expressions that involve input and output variables that range over infinitely many possible values. In contrast, machine learning methods are applied to problems specified by a finite (typically large) number of input/output pairs (also called “labeled inputs”). Note that both the formal specifications and the labeled inputs are “finitary” objects that are hence amenable to programs (on the meta-level).

Both approaches to automated programming have their pros and cons. For discussing the pros and cons, it may be useful to distinguish between the following quite different situations:

- Finding algorithms (programs) for fundamental, non-trivial, algorithmic problems (or hierarchies of such problems inside a well-defined formal theory): In this case, the problem specification and the solution algorithm and everything that happens between problem and solution could, in principle be formulated by completely formal and general expressions (i.e., by “symbolic objects”). Hence, the process from the problem specifications to the solution algorithms is, in principle, amenable to algorithmic treatment on the meta-level, i.e. to symbolic computation methods. Examples of such problems: the problem of sorting; the problem of finding shortest paths; the problem of finding symbolic integrals; the problem of finding Gröbner bases; etc.
- Developing software for *a typical every-day application* (like a calendar application, a search engine, an office administration system, etc.): In this case, the individual parts of the system (called “units”) should implement a (big) number of functionalities, most of which are not really difficult. Only some of the functionalities may involve the algorithmic solution of fundamental problems. The algorithms for these advanced functionalities, typically, are known and can be taken from reliable libraries (with some syntactical changes). Writing programs for the other units is not a big logical challenge but, still, error-prone. The complexity of such systems originates from the huge number of units, the many possibilities to commit small errors, and the various (desired and undesired) interactions of the units.

This distinction is important for the following reason: The application of formal methods for establishing the steps between problem and algorithm (e.g. the proof of the correctness of algorithms) only makes sense if we consider non-trivial algorithmic problems. In contrast, for most of the millions of units to be developed in large software systems a formal specification of the problem to be solved by the unit is often essentially identical to the code to be developed. In other words, a proposal for the code of a unit, in the case of “easy” problems, is a way for describing the problem to be solved.

In fact, I think it is fair to say that symbolic computation has not played any role in automating programming for every-day applications. In history, symbolic computation did not play a big role in establishing new algorithms for fundamental algorithmic problems either because the reasoning process for finding these algorithms was “brain-crafted” for each of these individual problems. Anyway, the power of symbolic computation methods for automated programming became evident in some attempts to build up coherent algorithmic mathematical theories consisting of definitions, problem specifications, theorems, algorithms, and the proofs of theorems and algorithms in the context of the entire theory, see for example the Coq proof assistant by Thierry Coquand, Gérard Huet, et al. [10], and the Isabelle proof assistant by Tobias Nipkov, Lawrence Paulson, et al. [11]. There were also some spectacular examples about how formal methods may guarantee the correctness of programs, for example the use of Gröbner bases for proving the correctness the switching circuits in multipliers, see [12].

PTLMs (Pre-Trained Language Models) like GPT by OpenAI shows that obtaining programs from problem specifications in natural language is feasible and this is a real game changer. Even if a lot of criticism is being voiced about the fact that many of the programs generated have flaws, it is a big step forward. Playing with ChatGPT and watching the demo video [13] is a real thrill. The criticism goes mainly into the direction that the objects going into the machine learning approach of automated programming (as input and output) do not contain any “semantics”, i.e., they do not say anything about a common model world,

see for example the interview [14]. In contrast, symbolic expressions in a symbolic computation approach “contain semantics”. (Note that the semantics of symbolic expressions is contained, implicitly, in the applicable reasoning rules.) Most applications of machine learning are semantic-free in the input, in the intermediate process, and in the output. However, in the application of machine learning to automated programming, the input is a problem specification formulated in natural language and the output is a program. Both have semantics. Although the semantics is not used in the automated programming process, the semantics of the input and the output are available in exactly the same sense as if a human programmer reads a specification and finally delivers a program. Thus, one could and should apply the same (practical and theoretical) methods for verifying the correctness of the programs generated in the frame of PTLMs as with programs generated by human programmers.

My main message in this note is: I think the next big step forward could and should be to combine the machine learning approach to generating (maybe not completely perfect) programs from natural language specifications and the symbolic computation approach to handling various intermediate steps in the programming process. This could be done, for example, in the following three stages:

- The translation of problem specifications in natural language to formal specifications. It is not clear how good the current PTLMs are in the translation of natural language text to a formal logic language (like predicate logic). My experiments in [8] (towards the end of the report) indicate that this translation is not very reliable. Actually, the translation of individual natural language statements to, say, predicate logic is not really meaningful because both natural language statements, and formal statements need contexts for obtaining semantics. Thus, translation from natural language to a logic language should be undertaken in a systematic way that considers entire corpora of text on certain subject areas like, for example, “handling calendars”.
- As soon as formal expressions are available, symbolic methods could be used for getting over the individual steps of program development. (I gave a tiny example of the power of symbolic methods above in the calendar program: The application of a quantifier elimination algorithm to a formal statement of the collision problem that involves an existential quantifier in a natural way and yields, automatically, a condition on the start and end times of two intervals to be collision-free.)
- If, instead, machine learning methods are used for program development from (natural language or formal) specifications, at the end one could try to apply symbolic methods to improve the proposed programs or, at least, to check correctness. A first step in this direction is, for example, described in [15]. The “symbolic” method developed in this paper is just a bunch of rules of how to eliminate frequent semantic errors in the programs generated by ChatGPT.

Summarizing, we can consider the entire development of methods for supporting, improving, semi-automating or automating the programming process as a path to establish an “assistant programmer” for programmers that takes more and more of the work from the programmer. As we know “by Gödel”, at a certain historical moment of our technological evolution, the human programmer will be necessary - and is fundamentally desirable - on the current top level of programming; however, this level will grow higher and higher in the next stage. At the moment, the “assistant programmer” consists of various successful techniques and tools in “software engineering”, powerful stacks and libraries, powerful testing strategies and tools, networks of programmers that act as a big programming consulting machines, etc. We propose and envisage that symbolic computation and machine learning methods should and could be combined in order to increase the level of sophistication which we can make

available in the “assistant programmer”. It can be predicted that the efficiency of programming will increase drastically within the next few years with dramatic consequences on the software industry, in particular the job market in this area.

Funding Open access funding provided by Johannes Kepler University Linz.

Data availability statement Data sharing not applicable to this article as no datasets were generated or analysed during the current study.

Declarations

Conflict of Interest The author declares that he has no conflict of interest.

Remark This note is a significant extension of [16], which was written for a different purpose and at a time when ChatGPT was not yet available.

Open Access This article is licensed under a Creative Commons Attribution 4.0 International License, which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons licence, and indicate if changes were made. The images or other third party material in this article are included in the article’s Creative Commons licence, unless indicated otherwise in a credit line to the material. If material is not included in the article’s Creative Commons licence and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder. To view a copy of this licence, visit <http://creativecommons.org/licenses/by/4.0/>.

References

1. Buchberger, B.: Meditation in Today’s World. Part I: Science, Technology, Economy, Welfare: The Reflexion Principle. (2022). Video on Youtube: https://youtu.be/rNhRn7WMT_U
2. Buchberger, B.: Symbolic computation (an editorial). *J. Symb. Comput.* **1**(1), 1–6 (1985). [https://doi.org/10.1016/S0747-7171\(85\)80025-0](https://doi.org/10.1016/S0747-7171(85)80025-0)
3. Grover, A., Thornton, C., Singh, P.: Automated algorithm synthesis: A survey. *ACM Comput. Surv.* **54**(2), 1–36 (2021)
4. Buchberger, B., Jebelean, T., Kutsia, T., Maletzky, A., Windsteiger, W.: Theorema 2.0: Computer-assisted natural-style mathematics. *J. Formaliz. Reason.* **9**(1), 149–185 (2016). <https://doi.org/10.6092/issn.1972-5787/4568>
5. Buchberger, B.: Algorithm invention and verification by lazy thinking. In: Petcu, D., Negru, V., Zaharie, D., Jebelean, T. (eds.) *Proceedings of SYNASC 2003 (5th International Workshop on Symbolic and Numeric Algorithms for Scientific Computing)*, Timișoara, Romania, October 1–4, 2003, pp. 2–26. Mirton Publisher, Timișoara (2003). ISBN: 973-661-104-3
6. Buchberger, B., Craciun, A.: Algorithm synthesis by lazy thinking: Examples and implementation in Theorema. In: Kamareddine, F., Proc. of the Mathematical Knowledge Management Workshop, Edinburgh, November 25, 2003, *EPTCS*, vol. 93, pp. 24–59. (2003). ISBN: 044451290X
7. ChatGPT: Optimizing Language Models for Dialogue. (2022). <https://openai.com/blog/chatgpt/>
8. Buchberger, B.: Is ChatGPT Smarter Than Master’s Applicants? RISC Report Series 23-04, Research Institute for Symbolic Computation (RISC), Johannes Kepler University Linz, Altenberger Straße 69, 4040 Linz, Austria (2023)
9. Buchberger, B.: Computer-unterstützter Algorithmenentwurf (computer-aided algorithm design). In: Bibel, W., Siekmann, J.H. (eds.) *Proceedings of the “Frühjahrsschule Künstliche Intelligenz” (Spring School in Artificial Intelligence)*, Teisendorf, Germany, March 15–24, 1982. *Informatik-Fachberichte*, vol. 59, pp. 141–202. Springer, (1982)
10. Coquand, T., Huet, G.: The Coq Proof Assistant. (2022). <https://coq.inria.fr/>
11. Nipkov, T., Paulson, L.: The Isabelle Proof Assistant. (2022). <https://isabelle.in.tum.de/>
12. Kaufmann, D., Biere, A., Kauers, M.: Verifying large multipliers by combining SAT and computer algebra. In: Barrett, C.W., Yang, J. (eds.) *2019 Formal Methods in Computer Aided Design, FMCAD 2019, San*

- Jose, CA, USA, October 22–25, 2019, pp. 28–36. IEEE, (2019) <https://doi.org/10.23919/FMCAD.2019.8894250>
13. Codex.: (2022). <https://openai.com/blog/openai-codex>
 14. Marcus, G.: Transcript of an Interview with Ezra Klein. (2023). <https://www.nytimes.com/2023/01/06/podcasts/transcript-ezra-klein-interviews-gary-marcus.html>
 15. Jain, N., Vaidyanath, S., Iyer, A.S., Natarajan, N., Parthasarathy, S., Rajamani, S.K., Sharma, R.: Jigsaw: Large language models meet program synthesis. In: 44th IEEE/ACM 44th International Conference on Software Engineering, ICSE 2022, Pittsburgh, PA, USA, May 25-27, 2022, pp. 1219–1231. ACM, (2022). <https://doi.org/10.1145/3510003.3510203>
 16. Buchberger, B.: Symbolic computation in software science: My personal view. In: Kutsia, T. (ed.) Proceedings of the 9th International Symposium on Symbolic Computation in Software Science, SCSS 2021, Hagenberg, Austria, September 8-10, 2021. EPTCS, vol. 342, pp. 1–13 (2021). <https://doi.org/10.4204/EPTCS.342.1>

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.