Separating Algorithmic Thinking and Programming

Maurice Chandoo*

Abstract. We show how the first steps of algorithmic thinking and programming can be trained separately. The learner is not assumed to have any prior experience. A general framework and a sequence of training tasks is described and tested in practice with two 10th graders. Both were able to write relatively complex programs using only pen & paper within two days.

To train algorithmic thinking, computational problems are presented as games to the learner. Roughly speaking, a winning strategy corresponds to an algorithm which solves the problem. Thus, if the learner consistently wins the game for various instances then this indicates that they have found an algorithm. We describe a general mechanism to translate a computational problem into such a game. For the programming part, the learner is shown how a program can be constructed from traces. Programs are specified in a language which depends on the underlying model of computation (think of Turing machines, pushdown automata or instruction set architectures); such a model can be seen as a notional machine. The language itself is very simple yet broadly applicable due to the generality of our definition of model of computation.

Keywords: machine-computer game, trace-based programming, model of computation, notional machine, programming education

1 Introduction

Complex tasks are usually taught by breaking them into smaller ones which can be trained separately. Writing a computer program requires to devise an algorithm for a given task and then implement it in a programming language. We shall call the ability to devise an algorithm *algorithmic thinking* and the task of implementing it *programming* (some prefer to call the latter coding).

To effectively train something the learner requires feedback to assess their success or performance. This enables them to appropriately adjust their course of action. Ideally, the feedback is immediate, informative and easy to obtain. To train algorithmic thinking separately, the learner needs to be able to verify a potential solution without having to implement it as a program. One way to do this is to give an informal description of the algorithm to someone else who verifies it. However, coming up with a description can be difficult by itself, especially for learners with limited means to express themselves verbally, which makes feedback neither immediate nor easy to obtain. We describe a game which enables the learner to quickly and easily verify their solution. Playing this game essentially amounts to executing the algorithm and a recording of the game corresponds to a trace (therefore playing it is also useful because the implicitly produced traces can be utilized to construct a program which implements the algorithm).

*Leibniz Universität Hannover, Institut für Theoretische Informatik, Appelstr. 4, 30167 Hannover, Germany; E-Mail: chandoo@thi.uni-hannover.de

A notional machine is described as "the general properties of the machine that one is learning to control" by du Boulay who introduced the concept [Bou86]. In the context of a programming language a notional machine can be seen as an execution model for the language which describes the semantics of its various constructs. Forming a mental representation of such a machine is one of the difficulties that novice programmers face. Sorva has given an extensive survey on notional machines and their relation to different research threads in programming education [Sor13]. One of the conclusions of the survey is that it is recommended to have notional machines as an explicit learning goal, which we follow in our approach.

Models of computation are a central part of our framework and form a subclass of notional machines. A model of computation can be seen as an abstract description of a machine: it consists of buttons (operations) and indicator lamps (predicates), and it resides in a particular state (machine state). The indicator lamps (partially) describe its current state. When a button is pressed the machine changes its state depending only on its current state and which button has been pressed. For example, a Turing machine with tape alphabet $\{0, 1, \square\}$ has 5 buttons and 3 indicator lamps. For each character x from the alphabet it has an indicator lamp which is on iff the current cell contains x and a button which writes x to the current cell. It also has buttons to move the head one cell to the left and right. Its state consists of the tape contents and the position of the head. This concept has also been described in [Sco67] to uniformly describe various kinds of automata.

Besides automata, it can also be used to describe computing agents such as a drawing turtle [KK18] or a lumberjack [RZD18]. For example, in a dimly lit maze the agent's actions (operations) could be to turn left by 90 degrees or move a step forward and the agent can see whether there is a wall in front and whether the exit is reached (predicates). The machine state describes the maze's layout and the agent's position. A limitation of models of computation is that the machine state cannot change unless an operation is executed. For example, an agent in a dynamic real-time environment cannot be described as such a model. Our framework (game, programming language & method) is applicable to anything that can be framed as a model of computation.

Our motivation for finding a way to teach algorithmic thinking and programming separately comes from the experience that students who struggle the most with programming tasks tend to think about algorithms in terms of a programming language ([Lam18] explains why this is problematic). A reason for this might be that they equate algorithms with programs. Since it is usually more difficult to liberate someone from a misconception than to prevent it from happening, we tried to find a way of letting the learner experience that an algorithm can be constructed without even being aware of a programming language. Futschek defines algorithmic thinking as a set of abilities related to constructing and understanding algorithms and also argues that it can be taught independently of programming [Fut06]. He and Moschitz also propose playing algorithms as means to verify them [FM10]. Moreover, we also wanted to emphasize that programming requires to take a myriad of details into account and even if an algorithm is easy to conceive, implementing it can be surprisingly difficult.

In general, the complexity of translating a mental representation of an algorithm into a program seems to be underestimated. In [Cha19a] we show how to deal with this translation process systematically. The programming method presented here is a simpler variant thereof (the difference is that operations and predicates are already given). Hilton et al. describe a similar approach in [HLR19]. Both have in common that one starts with executing the algorithm by hand for a concrete input and then generalizes the steps to build a program incrementally. A related approach called direct-manipulation programming is tested in [ADF19]. The idea is that the learner can manipulate a program in terms of its traces using special software which provides a visual link between the concrete values in the traces and their abstract representations in the code. These works emphasize the value of traces as means to bridge the gap between the mental representation of an algorithm and its implementation.

In our approach a trace for some input is written first (or generated by playing the game) and

used as specification of the program's expected behavior on that input. Then the learner performs two tasks on the trace to add additional information. One task is to assign a program state to each row of the trace. From this enriched trace a (partial) program is derived that is consistent with the trace. This process is repeated for additional traces until the program is complete. When adding additional traces the program is always modified such that it remains consistent with the previous traces, otherwise a contradiction occurs. Compared to ad hoc programming where the learner is left alone with figuring out how to construct programs, this approach imposes a structure on this process with clearly defined steps, which facilitates detecting errors.

In practice, the two 10th graders quickly understood the method and were able to intuitively apply it to construct complex programs. The method can be carried out using only pen & paper. An advantage of working away from computers can be that it eliminates the temptation of blindly guessing parts of a program and verifying its correctness by running it on some arbitrary input [Cut+19].

2 Machine-Computer Game

A model of computation consists of a set of machine states S, a set of operations (total functions from S to S) and a set of predicates (total functions from S to $\{0, 1\}$). For example, a counter machine with k registers can hold a non-negative integer in each register; a machine state is a sequence of k such numbers. For each register it has the operations to increment and decrement by one (decrementing a register that contains 0 has no effect) and a predicate which holds iff the register contains 0.

A simple task on the counter machine with 3 registers is to copy the contents of register 1 to register 2. More specifically, the machine is initialized with an unknown state and a (human) computer has to operate the machine such that register 1 and 2 contain the value that register 1 had in the beginning. The difficulty is that the indicator lamps (predicates) only reveal partial information about the machine state, i.e. whether a register contains 0 or not.

This can be played as a 2-player game where one person assumes the role of the machine and the other is the computer. The machine starts by choosing the initial machine state and writes it down on a piece of paper which the computer cannot see. Moreover, the machine has a card for each predicate with 'on' written on one side and 'off' on the other. These cards are used to simulate the indicator lamps for the computer. The computer can tell the machine to execute an operation. The machine writes down the new machine state after applying this operation and updates the indicator lamps by possibly flipping the cards. When the computer says 'end' the machine shows the hidden paper to the computer. The computer wins if the final contents of register 1 and 2 contain the value which was in register 1 in the beginning.

A trivial winning strategy for the computer is to determine the contents of register 1 by counting (decrement until 0) and then set register 1 and 2 to this value. This strategy can be generalized to solve arbitrary tasks: determine the initial machine state, solve the problem without the machine, enter the result into the machine. The issue is that no algorithm is executed on the machine itself. To prevent the learner from resorting to this strategy the additional requirement must be made that it should be possible to teach the winning strategy to a person who cannot count (e.g. a preschool child). More generally, the computer is only allowed to remember a constant amount of information independent of the input.

An actual computer can be used to simulate the machine which makes the game much faster and less error-prone. However, initially playing the game with another person can facilitate relating computation to 'physical reality'. For instance, in the case of the counter machine the machine-player can use a stack of beer mats to represent each register's content (number of beet mats = register value) instead of writing it down on a paper. This allows the learner to associate the abstract

quantities in the registers with something tangible. By writing characters from a fixed alphabet on the beer mats this also extends to stack machines (defined below) and makes it intuitively clear why counter machines are isomorphic to stack machines over a unary alphabet. In [FWR19] it is found that programming a physical object versus programming a simulation thereof does not significantly affect learning gains. Assuming this generalizes, using a computer to simulate the game with the beer mats might just be as effective; since the computer cannot see the beer mats until the end, the learner should also be able to play the machine in this simulation.

The learner can train the first steps of algorithmic thinking by trying to find algorithms for simple models of computation and tasks. If the learner consistently wins the game as computer (without resorting to the trivial winning strategy described above) then this indicates that they found a correct algorithm. The feedback for the learner is relatively quick (provided the inputs are not too large) and easy to obtain since there is no need to convert the mental representation of the algorithm into a program or an informal description.

A stack machine with k registers over an alphabet Σ can hold a string over Σ (including the empty string ε) in each register; a machine state is a sequence of k such strings. For each register it has an operation to remove the last character or append a given character from Σ; removing the last character from an empty string has no effect. For each register and each character x from Σ it has a predicate which holds iff x is the last character of the string in that register.

In the following, sequences of training tasks for the counter and stack machine are given. The number in the parenthesis indicates the number of registers. We write Ri and Ri' for the initial and final value of the i-th register, respectively. For a string x let $|x|$ denote its length. The right column contains tasks for counter machines and the left one for stack machines over the alphabet $\{\mathrm{A}, \mathrm{B}, \mathrm{C}\}$.

- Reverse (2): $\mathrm{R2}' =$ the reverse of R1
- Move (3): $\mathrm{R2}' = \mathrm{R1}$
- Copy (3): $\mathrm{R1}' = \mathrm{R2}' = \mathrm{R1}$
- Concat (3): $\mathrm{R3}' =$ R1 concatenated with R2
- Repeat (4): $\mathrm{R3}' =$ R1 concatenated with itself $|\mathrm{R2}|$ times
- Substr (4): $\mathrm{R4}' =$ substring of R1 starting from the $(|\mathrm{R2}| + 1)$-th character with length $|\mathrm{R3}|$
- Binary (4): $\mathrm{R2}' = x$ times A where x is the value of the binary number in R1 (A=1, B=0, last character is LSB)
- Move (2): $\mathrm{R2}' = \mathrm{R1}$
- Copy (3): $\mathrm{R1}' = \mathrm{R2}' = \mathrm{R1}$
- Add (3): $\mathrm{R3}' = \mathrm{R1} + \mathrm{R2}$
- Mult (4): $\mathrm{R3}' = \mathrm{R1} \cdot \mathrm{R2}$
- Parity (1): $\mathrm{R1}' = \mathrm{R1} \bmod 2$
- Prime (6): $\mathrm{R2}' = 1$ if R1 is prime, otherwise 0

For the task Substr the initial machine state must be chosen such that it describes a valid substring, i.e. $|\mathrm{R2}| + |\mathrm{R3}| \leq |\mathrm{R1}|$. The game for these tasks can be played at [Cha19b].

This game can be used for any computational problem which can be framed as the task of reaching a particular target machine state from an unknown initial machine state in some model of computation.

Here is an example. In a gummy bear factory there are two containers: one with sugar and one with food dye. Beneath these two containers is a mixer. A gummy bear consists of three pieces of sugar and two pieces of food dye. There is a button which drops a piece of sugar in the mixer and another one which drops a piece of food dye in the mixer, provided the respective container is not empty. Moreover, there are two indicator lamps which show whether the sugar or food dye container

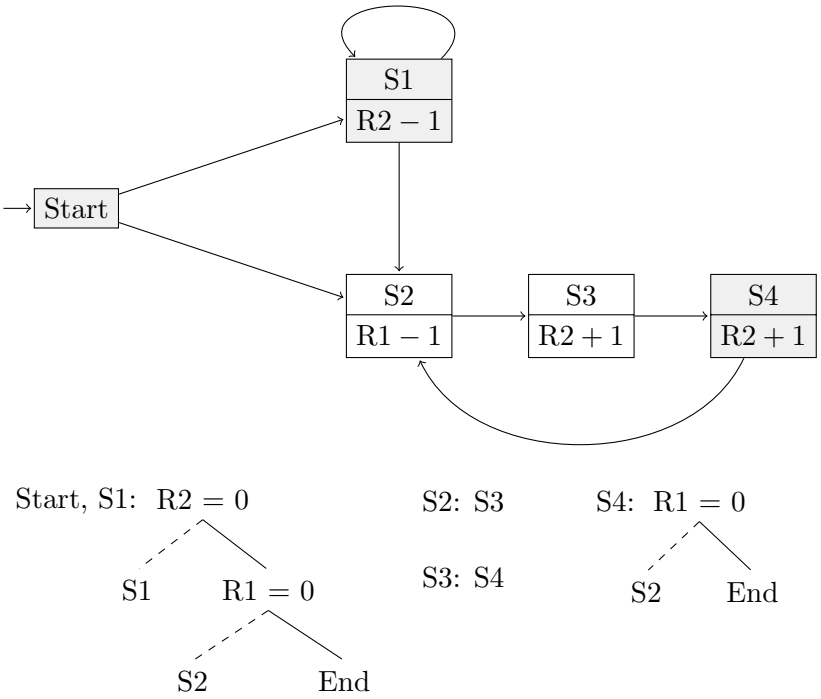

Figure 1: Example program for a counter machine

is empty. The task is to put as many piece of sugar and food dye as possible into the mixer in the correct ratio (3:2) to produce gummy bears. For example, if there were 60 pieces of sugar and 45 pieces of food dye in the beginning then there should be 60 pieces of sugar and 40 pieces of food dye in the mixer in the end. The challenge is that one does not know how many pieces of sugar and food dye are in the containers in the beginning. In certain cases it is not possible to achieve a correct ratio (e.g. 5 pc. sugar, 4 pc. food dye); how can the indicator lamps be modified to solve this issue?

3 Programming with Traces

We define (machine) programs w.r.t. a model of computation. A program is represented as control flow graph (CFG) with a designated start node. Nodes of the CFG are called program states. Each program state is associated with an operation (except the start node) and with a binary decision tree (BDT) which specifies what program state to visit next or whether the program should terminate. The inner nodes of such a tree are labeled with predicates and the leaves with program states (except the start node) or 'End'.

Figure 1 shows a program for a counter machine with two registers. If a predicate does not hold then the left child (dashed line) is taken. A program state is gray if the program can terminate at that state. The program doubles the value of register 1 and writes it to register 2, clearing register 1 in the process. Table 1 shows the trace produced when executing this program on input (2,1). An online interpreter for counter and stack machines is available at [Cha19b].

Observe that the CFG is redundant since it can be reconstructed from the BDTs, i.e. a mapping from program states to operations along with the BDTs is a complete description of a program. However, the CFG provides an alternative perspective on the program which we deem valuable for the learner.

Since this definition works for any model of computation it can be used to describe programs for

Table 1: Execution trace of program from Fig. 1

Program State	Operation	R1	R2
Start		2	1
S1	$R2 - 1$		0
S2	$R1 - 1$	1	
S3	$R2 + 1$		1
S4	$R2 + 1$		2
S2	$R1 - 1$	0	
S3	$R2 + 1$		3
S4	$R2 + 1$		4
End			

various kinds of automata, computing agents and instruction set architectures to a certain extent. In contrast to assembler programs the control flow is explicit and jump/branching operations are replaced by BDTs. The lack of abstraction mechanisms leads to a compact and simple definition which can be quickly understood.

A program can be derived from traces as follows. When a trace is generated from playing the machine-computer game each row only contains the operation and machine state. The first step is to assign program states to each row. This requires deciding which rows correspond to the same program state. If two rows have different operations then they cannot correspond to the same program state. The converse is not necessarily true, i.e. the same operation does not imply identical program states. The next step is to add an additional column to the trace which contains the sequence of predicates that were considered to determine what state to visit next. This information can be used to systematically construct the desired program.

To illustrate this, let us consider how the program from Fig. 1 can be partially reconstructed from Table 1. The trace already contains the program states. The next step is to add a column for the predicates. In row 1 (Start) only $R2 = 0$ is considered. Since it does not hold we go to S1. In row 2 $R2 = 0$ and then $R1 = 0$ is considered. Since the first holds and the second does not, we go to S2. In row 3 no predicate is considered and we immediately go to S3. The sequence of predicates in a row describes a path through the BDT of the respective program state. For example, row 2 uncovers the path $R2 = 0 \longrightarrow R1 = 0 \dashrightarrow$ S2 in the BDT of S1.

The machine-computer game can be modified to reveal these predicate sequences as well. The machine does not use cards to reveal the truth of the predicates anymore. Instead, the computer has to ask the machine whether a certain predicate holds until it is clear with what operation to continue. The sequence of queries before the computer decides the next operation describes the predicate sequence for that particular row of the trace.

If one strictly follows this approach then programming reduces to selecting a set of inputs whose traces reveal the complete program, computing those traces, adding program states and predicate sequences and mechanically deriving the program from this information. The program can be built incrementally by following the above steps for each input. The partial programs constructed during this process can be helpful to determine what other inputs to add.

This programming method comes with a correctness guarantee for the partial programs. Suppose that P is the target program that one wants to construct. A mental representation of P is assumed to be available, which is used to generate the required information. Let P' be a partial program which is synthesized from the traces T. Notice that P' is not necessarily a complete program because there might be inner nodes in a BDT with only one child. To resolve this, add a node labeled $\oslash$ for each missing child. The program state $\oslash$ represents undefined behavior; its BDT is 'End' and

its operation arbitrary. Executing this modified program on input x will reach $\oslash$ as final state whenever the behavior of P' on x is undefined. Now, if the traces T are consistent with P then P and P' produce the same trace for every input x for which the behavior of P' is defined. For example, the behavior of the partial program constructed from Table 1 is defined for all inputs (x, y) with $x \geq 1$ and $y = 1$. Therefore it will output $(0, 2x)$ for all such (x, y).

4 Experience Report

We tested our framework and the previous tasks with two 10th graders A and B who had no prior experience with programming. They attended a 5-day workshop for ca. 3 hours per day. They were given work sheets with multiple problems. We engaged only if we noticed a conceptual misunderstanding, the interpretation of a problem was unclear or to verify solutions. They were allowed to interact, but only did so to compare their solutions for the first worksheet.

Worksheets. Worksheet 1 had two problems similar to the task MOVE for the counter machine but with a real-world context. The first one involved trading apples against pears before counting was invented (give one apple, receive one pear and so forth until one party runs out). The second one was the gummy bear factory from section 2.

Worksheet 2 explained the machine-computer game for counter machines with beer mats. The game was described such that the connection to counter machines was not as obvious. Instead of predicate cards the machine-player has to notify the computer as soon as a stack becomes empty. The problem was to find winning strategies for MOVE and COPY without counting. Worksheet 3 contained a description of counter machines and the 6 tasks for them. The problem was to find algorithms for each of them.

Worksheet 4 contained the program from Fig. 1 and two execution traces when executing it (similar to Table 1). The first problem was to write a trace when executing this program for a given input. The second problem was to find out how the program can be partially reconstructed from the given traces; no further hint was given. The purpose of this problem was to make them aware of the connection between traces and programs. The last problem was to write programs for the 6 previous tasks.

Worksheet 5 described the machine-computer game for stack machines with beer mats. The problem was to find winning strategies for the first three tasks for stack machines without memorizing the stack contents. Worksheet 6 described stack machines and the seven tasks for it. The first problem was to find algorithms for each one and the second problem was to write programs implementing them. For the task BINARY there was an explanation how a binary number can be converted into decimal by summing the appropriate powers of two.

To demonstrate the algorithms that they found they had to write traces for given inputs. We provided printed sheets with trace tables that they could fill in. For the more difficult tasks we also asked them to play the game on the computer.

Observation. Both of them found correct algorithms for all tasks up to and including PARITY after just one hour. They did not play the game at all (either on the computer or with beer mats). Both failed to find an algorithm for PRIME. When we asked them to play the game for the more difficult tasks, they usually needed two or three tries until they were able to reliably execute their algorithm.

After less than 20 minutes both had written the execution trace for the program on worksheet 4 and figured out that the sequence of program states in a trace describes a path through the CFG. Next, A decided to write a program for MULT first and B decided to write a program for ADD. We gave both a laptop with an interpreter where they could type in their programs to run them and

explained the syntax of the interpreter. After 1 hour B claimed to have a correct program. We found an error during testing, which was resolved by B after another half an hour. A had a correct program for MULT after 2 hours. This concluded the first day.

On day 2 we demonstrated the programming method. During the demonstration we asked them how to proceed at certain points to check whether they could follow. This took 15 minutes. Then we gave them trace tables with an additional column for the predicate sequences to facilitate the use of the method. We asked them to write a program for MULT using only pen & paper. Initially, A was not sure how to fill out the predicate sequence column: instead of writing down the sequence of predicates considered for that particular input, A seemed to write down the subset of predicates that has to be considered at the corresponding program state over all inputs.

After less than 45 minutes A was done. We typed it into the computer and it worked correctly immediately without further modifications. After 90 minutes B was done. We typed it in but it returned incorrect results in certain cases. It took B only a few minutes of looking at the trace produced by the program to identify the cause of the problem (in two places the wrong register was referenced). After this correction the program worked correctly. B's program for multiplication had 15 program states and implemented a more efficient algorithm than A's whose program had only 9; for certain inputs B's program was nearly twice as fast.

It took A and B another 30 minutes to write programs for the other tasks on the counter machine using only pen & paper. All of their programs worked immediately when typed in with the exception of B's program for ADD (a predicate was missing). A said that the method was useful for MULT but not so much for the other tasks since the programs were quite simple.

Both solved worksheet 5 before the end of day 2. It took both 20 minutes to understand the game and find correct winning strategies for all problems with the following exception. B's strategy for COPY was incorrect because it involved implicitly counting the length of a string. This concluded day 2.

The remaining 3 days they were working on worksheet 6. A found algorithms for all of the tasks. B found algorithms for all tasks but BINARY. We gave B a trace produced by an algorithm which solves it. After 12 minutes B was able to understand the algorithm from the trace alone and execute it by playing the game. B wrote correct programs for all tasks but CONCAT and BINARY before the workshop ended. A wrote correct programs for all tasks but REPEAT. While B used the method for all problems, A tried to write programs for REPEAT and SUBSTR on the computer. In both cases A failed to write correct programs without the method. After two hours A gave up on the program for SUBSTR because it had multiple errors. A tried again on paper using the method which lead to a correct program for SUBSTR after 90 minutes. A expressed surprise that programming on the computer can be more difficult than on paper.

In general, A and B's programs were either immediately correct or had one error which they resolved after one correction. If they had an incorrect program we gave them the choice to continue with the next task and return to the current one later on.

We noticed that after MULT both only used the method to construct the CFG and did not write predicate sequences. Instead, they recognized certain patterns in the BDTs and that some subtrees occured multiple times in a program. They would draw a colored circle around the first occurrence of such a subtree and then used the color to reference it.

Interpretation. Both had a relatively high level of abstraction as exhibited by the fact that they did not need to play the game at all to come up with the algorithms/winning strategies. Also, the connection between the beer mat game and the counter and stack machines were obvious to them. It took them only a few minutes to come up with algorithms for a task. We suppose that their performance during the workshop was above average with respect to their age group.

We claimed that the game can help the learner to train the first steps of algorithmic thinking.

However, the learners did not play the game for themselves at all. Why is that? Clearly, playing the game itself does not help in finding an algorithm. The learner needs to come up with an algorithmic idea beforehand. The game can only act as an external source which reinforces the belief in the correctness of the algorithm. If a learner is already confident in their algorithm's correctness then playing the game provides no additional value in that regard. In particular, we believe that the game can help a learner to build confidence in their ability to reason (about the correctness of algorithms) until the point where it is not needed anymore. A and B were already beyond that point and thus found no use in playing the game to check their algorithms. Nonetheless, we found it useful to let them play the game to confirm that they had found a correct algorithm.

The programs that they had to write were quite large. For example, both their programs for Substr had 27 program states and over 60 nodes in the BDTs. Writing programs of such complexity is not trivial. We let them try the ad hoc approach on the computer to see how it compares to the method. For instance, A's ad hoc program for Mult had 10 program states and 44 nodes in the BDTs whereas the one constructed with the method had only 9 program states and 31 nodes. A required less than half of the time to write the program with the method. An even stronger example of the method's usefulness is that A failed to write a program for Substr without it. We would expect that a weaker learner can profit even more from the method because they would already fail to write simpler programs ad hoc.

However, programming with traces alone is not a silver bullet. It order to write programs effectively it has to be combined with other techniques. A good example is the task Concat for which B failed to write a correct program using the method. It can be solved using only 4 calls to a subprogram for Reverse. Knowing how to compile a program with subprogram calls makes it trivial to write a program for Concat if a program for Reverse is already known. In contrast, trying to write a program for it using only the method is tedious and error-prone because it requires a lot of needless repetition.

Similarly, neither A nor B used predicate sequences for the tasks on the stack machine. Instead they recognized recurring subtrees and used them to efficiently construct the BDTs after having derived the control flow and program states with the method. Deriving BDTs strictly from traces in these cases would have been much more laborious.

The learner should be taught how the method can be used more flexibly and how techniques such as subprogram calls can be integrated into its workflow; something that we did not do. For example, subprogram calls can be incorporated by adding them as operation to the model of computation. In this case the learners figured out how to modify the method by themselves in order to accommodate it to their needs; something that cannot be expected of every learner.

We conjecture that writing the more complex programs using only pen & paper would have been infeasible for the two without the method. Since it is impractical to use the method directly on the computer without additional software support, it was easier for them to program away from it. However, we think that pen & paper are not essential in this regard. Roughly speaking, letting the learner work with pen & paper has the advantage that is forces them to think whereas when programming on the computer there is the temptation of guessing part of the program and running it to check. A software which enforces a rigid application of the method (the program cannot be manipulated directly) would eliminate that temptation.

5 Conclusion & Future Work

We described a theoretical framework consisting of models of computation and machine programs, a game which allows the learner to test and execute arbitrary algorithms within this framework, a method whereby a program can be constructed from execution traces and a sequence of training tasks.

The presented framework is simple yet general. Our notion of program is simple enough that a single example can be already sufficient for a learner without prior knowledge of any programming language to fully comprehend its syntax and semantics. Despite this simplicity such programs can be used to describe algorithms on the same level of abstraction as modern programming languages [Cha19a]. During the workshop the two 10th graders quickly understood the framework just from the description and problems on the worksheets without further instructions.

The game was not played by the 10th graders to verify their algorithms. From this one might infer that it provided no value. However, it also serves the purpose of implicitly describing what constitutes an algorithm (a certain kind of winning strategy). Before a learner can be asked to find algorithms they need to know what algorithms are. An abstract definition is likely too vague in order for the learner to understand what they are looking for. The alternative is to directly introduce programs. But this can cause the conflation of programming and algorithmic thinking since the learner might feel obliged to represent their solution as program. A learner which struggles to find a correct implementation might not understand whether the algorithm or the implementation is wrong. Instead of writing a program, the learner can represent their algorithm in terms of traces. During the workshop we found traces to be quite effective as means to communicate algorithms, especially when paired with a description of the idea behind the algorithm. Writing such traces implies the ability to mentally simulate the game. Thus, while the 10th graders did not play the game they must have simulated it mentally.

A disadvantage of the game compared to programs is that not every winning strategy corresponds to an algorithm. More specifically, any strategy which requires the computer to remember a non-constant amount information does not correspond to a program. In general, it might not be easy for the learner to understand what constitutes a valid strategy.

The method to derive a program from traces worked well in practice. The two 10th graders were able to understand and use it surprisingly quickly. But the workshop also showed that in certain instances a rigid application of the method can be exhausting and error-prone. Therefore the learner should be taught basic forms of abstraction early on to complement the method. Also, applying the method by hand requires a lot of concentration.

We plan to conduct the algorithmic part of this workshop for a larger group of younger students. The notion of a program will not be presented to the students in order to evaluate whether teaching algorithmic thinking using only the machine-computer game is feasible. They should learn the following abilities: recognizing whether a winning strategy constitutes an algorithm, describing an algorithm using traces and verbal descriptions, designing algorithms from scratch and translating a given idea into an algorithm.

We also intend to develop a software which facilitates programming with traces. More specifically, a modified version of the game can be played which also yields predicate sequences. Then the user can add state labels to such a trace and automatically synthesize a (partial) program from it. When another trace is added, the software also checks whether it is consistent with the current program. Such a software could simplify the use of this method by taking care of menial tasks.

References

[ADF19] Michel Adam, Moncef Daoud, and Patrice Frison. “Direct Manipulation versus Text-based Programming: An experiment report”. In: *Proceedings of the 2019 ACM Conference on Innovation and Technology in Computer Science Education, Aberdeen, Scotland, UK, July 15-17, 2019.* 2019, pp. 353–359. DOI: 10.1145/3304221.3319738. URL: https://doi.org/10.1145/3304221.3319738.

[Bou86] Benedict du Boulay. “Some Difficulties of Learning to Program”. In: *J. Educational Computing Research* 2.1 (1986).

[Cha19a] Maurice Chandoo. “A Systematic Approach to Programming”. In: *CoRR* abs/1808.08989 (2019). arXiv: 1808.08989.

[Cha19b] Maurice Chandoo. *Games and Interpreter for Counter and Stack Machines*. 2019. URL: https://upsl.uber.space/aws19/info.txt.

[Cut+19] Quintin I. Cutts, Matthew Barr, Mireilla Bikanga Ada, Peter Donaldson, Steve Draper, Jack Parkinson, Jeremy Singer, and Lovisa Sundin. “Experience Report: Thinkathon - Countering an “I Got It Working” Mentality with Pencil-and-Paper Exercises”. In: *Proceedings of the 2019 ACM Conference on Innovation and Technology in Computer Science Education, Aberdeen, Scotland, UK, July 15-17, 2019*. 2019, pp. 203–209. DOI: 10.1145/3304221.3319785.

[FM10] Gerald Futschek and Julia Moschitz. “Developing Algorithmic Thinking by Inventing and Playing Algorithms”. In: *Constructionism, Paris*. 2010.

[Fut06] Gerald Futschek. “Algorithmic Thinking: The Key for Understanding Computer Science”. In: *Informatics Education - The Bridge between Using and Understanding Computers, International Conference in Informatics in Secondary Schools - Evolution and Perspectives, ISSEP 2006, Vilnius, Lithuania, November 7-11, 2006, Proceedings*. 2006, pp. 159–168. DOI: 10.1007/11915355_15.

[FWR19] Grégoire Fessard, Patrick Wang, and Ilaria Renna. “Are There Differences in Learning Gains When Programming a Tangible Object or a Simulation?” In: *Proceedings of the 2019 ACM Conference on Innovation and Technology in Computer Science Education, Aberdeen, Scotland, UK, July 15-17, 2019*. 2019, pp. 78–84. DOI: 10.1145/3304221.3319747.

[HLR19] Andrew D. Hilton, Genevieve M. Lipp, and Susan H. Rodger. “Translation from Problem to Code in Seven Steps”. In: *Proceedings of the ACM Conference on Global Computing Education*. CompEd ’19. Chengdu,Sichuan, China: ACM, 2019, pp. 78–84. ISBN: 978-1-4503-6259-7. DOI: 10.1145/3300115.3309508. URL: http://doi.acm.org/10.1145/3300115.3309508.

[KK18] Tobias Kohn and Dennis Komm. “Teaching Programming and Algorithmic Complexity with Tangible Machines”. In: *Informatics in Schools. Fundamentals of Computer Science and Software Engineering - 11th International Conference on Informatics in Schools: Situation, Evolution, and Perspectives, ISSEP 2018, St. Petersburg, Russia, October 10-12, 2018, Proceedings*. 2018, pp. 68–83. DOI: 10.1007/978-3-030-02750-6_6. URL: https://doi.org/10.1007/978-3-030-02750-6_6.

[Lam18] Leslie Lamport. “If You’re Not Writing a Program, Don’t Use a Programming Language”. In: *Bulletin of the EATCS* 125 (2018). URL: http://eatcs.org/beatcs/index.php/beatcs/article/view/539.

[RZD18] Sylvia da Rosa Zipitria and Andres Aguirre Dorelo. “Students Teach a Computer How to Play a Game”. In: *Informatics in Schools. Fundamentals of Computer Science and Software Engineering - 11th International Conference on Informatics in Schools: Situation, Evolution, and Perspectives, ISSEP 2018, St. Petersburg, Russia, October 10-12, 2018, Proceedings*. 2018, pp. 55–67. DOI: 10.1007/978-3-030-02750-6_5. URL: https://doi.org/10.1007/978-3-030-02750-6_5.

[Sco67] Dana S. Scott. “Some Definitional Suggestions for Automata Theory”. In: *J. Comput. Syst. Sci.* 1.2 (1967), pp. 187–212. DOI: 10.1016/S0022-0000(67)80014-X.

[Sor13] Juha Sorva. “Notional machines and introductory programming education”. In: *TOCE* 13.2 (2013), 8:1–8:31. DOI: 10.1145/2483710.2483713. URL: https://doi.org/10.1145/2483710.2483713.