Research Article

Channeling the Flow – A Metaphor for Computer Programs

Attila Egri-Nagy[1]

1. Akita International University, Akita, Japan

What do we do when we program computers? A standard general answer could be that by writing a program we create a process. Here, we entertain a different metaphor: *programming as shaping an existing process by constraints*. By emphasizing constraining over construction, we liken writing code to other activities like engineering and playing board games. Our goal is to enrich the experience of computer programming and deepen its understanding by a possibly challenging analogy. A new appreciation might be of use at the age when humans are about to stop doing this activity.

1. Introduction

We believe that writing computer programs is an efficient way of learning. Programming stands high in the hierarchy of understanding something. It is at the same level, if not higher, as being able to explain the idea to someone. As a form of learning, it will remain to be an important and rewarding human activity, even if most coding will be automated in the future.

Programming is part of a network of activities involved in solving real world problems. In software engineering, we analyze the requirements, design the software architecture, implement and test the functions, we release them as a software package, integrate it into other systems and keep maintaining it. There are other use cases, e.g., creative coding and smaller scale instances like scripting. The common underlying activity in all these is the writing of *code*, i.e., text in a formal language. This view offers a good everyday definition: computer programs are texts written in programming languages to carry out information-processing tasks. The program code initiates a computational *process*, which produces the expected behaviour or the desired result.

First, this short description draws our attention to the running program. Here, we are interested in this dynamic aspect, not in the software artifact, which is a static entity. Second, we can consider

 doi.org/10.32388/VU0ZGo.2

programming as the *construction* of the computational process. This is the view we want to challenge, or at least to make it more subtle.

Programming can also be viewed as an art form [1]. This reinforces the view that writing computer programs is about creating something new. However, the novel entity comes about by restricting what can happen. Much like sculpting is removing the unnecessary part, according to the famous Michelangelo quotes.

2. The main metaphor

By thinking about what happens when we run the code, we will shift the perspective from creating a computational process to shaping an existing one. In a digital computer, the running code traverses through the state space of all possible machine states. The path can zig-zag and loop back. This movement can be generalized to analog computers [2], where the settings of input knobs may represent the program. The path is in a continuous space. In both cases, by writing the program, we *create* paths in state space.

Once we have the state space view, we can talk about constraints. *Constraining* is limiting the possibilities, reducing the degrees of freedom. We can visualize it is as trying to steer something already in motion. We create the path, by not letting it to go anywhere else.

As we argued elsewhere, computation, in general, is about forcing physical systems through some dynamics that are meaningful to us [3]. In this paper, we want to elaborate on this thesis but on all levels of computation, not just on the lowest machine code level. We state our definition upfront of the underlying metaphor.

> *A program is a set of constraints that acts as guardrails for a pre-existing directionless process.*

Therefore, programming is about cleverly placing obstacles. Then, a blind force bumps into those, and thus, it gets stopped and redirected by them. When the plan works out, it ends up doing valuable work for us. What is valuable is defined by us, the functional requirements of the program we write.

This definition is so abstract that it will encompass many things, including those very far from the software packages on digital computers. Accordingly, we will need to treat the definition of a computer very liberally. This we consider as a feature, not a bug.

2.1. The lead example: the canals

Canals are the visual and physical examples for our metaphorical thinking about programming. Water itself is willing to flow in any direction as long as it is downwards as it is pulled by gravity. We can stop the flow by building a dam. By digging out a new riverbed, we can redirect water. By making reservoirs, we can store it later, sort of delaying the flow. Constructing these engineered structures is a hard work. We dig artificial waterways for good reasons. We may want better navigation than wild rivers or flood control. In agriculture, we build a network of irrigation channels for higher crop yields. Paddy fields are elaborate structures for the controlled flooding of rice seedlings. To maximize the yield, the right amount of water needs to be there at the right time to kill the weeds and supply minerals.

The engineered structures are constraints controlling the flow. To spell it out, water and gravity form the underlying hardware, and the channel structure is the software. We can say that we program water by construction work to ensure that the plants will not dry out.

2.2. Notes on the methodology

Conceptual metaphors organize our everyday human experience seamlessly [4]. Here, we use these cognitive tools explicitly as knowledge transfer tools between different domains [5]. After finding an initial link, we explore how and to what extent does the internal structure and dynamics of one domain correspond to the other domain. For the mathematically minded, we try to find a functor between two categories. Deliberate application of metaphors has been suggested even in software engineering [6].

We are not trying to find the *differentia specifica* of programming here. We aim for the most abstract idea, which has a more extensive reach by its definition.

2.3. Logic and relational programming

The metaphor becomes literal in *constraint programming*, which covers a range of techniques for combinatorial problem solving. Most notably, logic and relational programming [7][8].

In these paradigms, there is an underlying combinatorial search algorithm guided by the given set of constraints. The search itself is directionless as without constraints it would go into all directions by enumerating all combinatorial possibilities. We argue that this mechanism extends metaphorically to other programming paradigms, down to how computers work and even to engineering in the physical world.

3. Computational Power as Flow

Now we consider a digital computer, a desktop PC or laptop device. It has a processor, a memory architecture, input-output and storage devices. How does this digital computing device have a directionless process inside? We have two levels for this interpretation: a physical and an abstract. On the physical level we have the flow of electricity. On the abstract level we have the execution and data flow, the universality of the computing machine.

3.1. Physical level - Electricity

In a simplified view, electricity is the flow of electrons. Once freed from atoms, electrons can go anywhere by their nature. This random wiggling is the underlying directionless process. A power source providing voltage can then move electrons in one direction in a wire. A wire is an artificial river for electron flow. In this sense, an integrated circuit is an elaborate pipework. The basic building block of logic circuits, the transistor, is like a lock, or a sluice gate for electricity.

The analogy with the canals has its limits. Intricate connections on the silicon chip are lot more complex. Having an irrigation network with that many channels would not fit into our everyday world. The purpose is different too. In agriculture, the presence of water is an objective, so its distribution is the goal. In computing, the flow of electricity represents something else: a bit in a binary digit or in a Unicode character, or any other digitally represented piece of information.

The wonder of digital computing is that we can choreograph those wandering-wiggling electrons into meaningful patterns tracing the abstract logic of our data transformation.

3.2. Abstract level - Execution flow in state space

On the abstract level, we have computational power defined by the ability to perform data transformation operations by executing instructions. It is an everyday experience that computers can do different things depending on the applications we use. Computers are *universal* [9]. If a computer can do anything computable, then there is no directionality in computation. Universal implies directionless. Therefore, we need a program to constrain the flow of execution.

Program execution is also a form of movement: a working processor traces a walk in the state-space of the computer, defined by the totality of all possible configurations of the memory and the processor's internal registers. Technically, the state-space would be a graph with cycles in it, but we can model it with

a tree for input-to-output computations that make a steady progress towards the results. This tree structure with an immensely high branching factor can represent all the processor's possible state transition dynamics starting from a given reset state.

The computer can potentially go in many directions, meaning that it can go through arbitrary state sequences. Thus, we have to limit these choices. This limitation is what a computer program does: choosing a specific execution line from an immense set of possibilities. A computer program corresponds to a subtree, where the branches within the subtree represent the executions of the same program with different input data. This view is not limited to assembly code. The same argument can be made for any virtual machine or higher level notional machines [10].

One could argue that an 'idle' computer is a counterexample. However, idling is the continuous execution of a simple wait cycle. The computer is still running a program.

There is also a very particular interpretation of the constraining the flow idea: the UNIX pipeline, where simple programs are composed to process the data flow, following the operating system's design philosophy [11].

3.3. Board games as computation

The tree picture of program execution is similar to game trees in artificial intelligence [12] for two person complete information board games. A particular game is a path in the tree, while the whole tree is the totality of all possible games. The interaction of the two players 'computes' the result, the final board position.

For a complex enough game, like Chess or Go, every match is a creation of something unique that can be appreciated and analyzed later. Still, the process can be viewed as a sequence of constraints. Each move is a choice from all available legal moves and reduce the set of future possibilities. In expert play, each move contributes to the final result. Similarly, the subtree corresponding to a computer program should contain states that somehow encode the computation result.

4. Programming the physical world a.k.a. engineering

We used canals, engineering artifacts, as a metaphor for programming. However, it is possible to turn around the cognitive metaphor. Civil engineering predates software engineering, but nothing prevents us using the latter to understand the first, upending the chronological order. It is possible that someone

trained as a programmer to use her understanding of computers to project it onto the physical world to understand engineering. Thus we have the metaphor of *engineering as programming the physical world*.

We can look at the shared principle of *efficiency*. For example, trying to improve the steam engine we want to have the molecules in steam to hit the piston, to make it move. However, the pressure of steam is a blind force. The molecules also hit the wall of the cylinder, not just the piston. Though the efficiency is limited, the engine works. The constraints are defined by how the piston is allowed to move. Without these constraints, the cylinder would simply heat up, the speedy but random movement of the molecules would not do any useful work.

In programming, optimization aims to remove unnecessary computation or trying to reduce the amount data to be transferred. On the hardware level, the abstract computational meets the physical substrate. The efficiency is in trying to maximize the electron movement with symbolic meaning versus their free movement producing waste heat.

The question arises: What is the essential difference between building something physical, like a DIY solar system (solar panels wired to a charge controller, battery bank, inverter), and developing a software application, for instance, a program that monitors, records, and visualizes the performance of those solar panels? We claim that there is not much difference.

This view has a connection to constructor theory [13][14]. It is a new way of formulating physical laws in terms of counterfactuals: what can and cannot happen. For instance, a physical system's state can only be used as computer memory if the system could be in another state. We can (mis)use this grand theoretical framework for our purposes. An engineering artifact comes with its own laws that constrain the possibilities.

4.1. Software bugs

The ground experience of a programmer is often the quest for finding errors. In software engineering parlance, bug hunting. These frequently come from the unintended consequences in the workings of the code. Or, in our terminology, the unexpected implications of the constraints. Transferring this into the realm of engineering, into the programming the physical world, we can find further examples. Maybe the biggest and most painful is the use of fossil fuels. These are unintended consequences of rearranging materials for our purposes.

4.2. Sandbox style video games

For the similarity of engineering in the physical world and software development in the abstract realm, there is the interesting middle ground of virtual worlds. Minecraft-like sandbox style video games define a virtual world with its own 'physical" laws. Players leverage these laws by placing blocks in the discrete world. For instance, putting light sources stops the monsters spawning, which is a basic mechanism of this game world.

On the other hand, we see clearly that everything happening in the game is a runnig programn. The player's actions in the game correspond to changes in the underlying database. Those updates can be done by a computer program. Therefore, there is very little difference between the world altering activity inside the game and the execution of a computer program.

Writing 'mods', that are short programs, scripts, to modify the game world's behavior is very much part of the gaming culture. Perhaps, for regular gamers, expressing engineering as programming the real world is a fairly obvious idea.

5. Conclusion

Here we elaborated on the subtle difference between *'I tell the computer what to do.'* and *'I setup the computer in a way that will do something useful.'* The latter is closer to a semantic definition of computation: computation is a constrained dynamical system, such that its trajectory or its final state is meaningful for us. Consequently, the only difference between engineering and programming is whether we are interested in the state of the physical system itself, or it is a model of something else. In both cases we want to purposefully limit the set of possible states of the physical system.

When engaged in computer programming, do we *construct* or *constrain*? We argued that the second interpretation is also possible and the two can coexist. The constraining view can lead to a unified understanding of engineering in the physical world and programming in the abstract realm.

Acknowledgements

The paper benefited from the discussion at the 2nd Akita Philosophy Seminar.

References

1. ^ *Donald E. Knuth. Computer programming as an art. Commun. ACM, 17(12):667–673, 1974.*
2. ^ *B. Ulmann. Analog and Hybrid Computer Programming. De Gruyter Textbook. De Gruyter, 2020.*
3. ^ *Attila Egri-Nagy. The algebraic view of computation: Implementation, interpretation and time. Philosophies, 3(2), 2018.*
4. ^ *George Lakoff and Mark Johnson. Metaphors we Live by. University of Chicago Press, Chicago, 1980.*
5. ^ *G. Fauconnier and M. Turner. The Way We Think: Conceptual Blending and the Mind's Hidden Complexities. Basic Books, 2002.*
6. ^ *Alvaro Videla. Metaphors we compute by. ACM Queue, 15(3):40:52–40:62, 2017.*
7. ^ *L. Sterling and E.Y. Shapiro. The Art of Prolog: Advanced Programming Techniques. Logic programming. MIT Press, 1994.*
8. ^ *William E. Byrd. Relational programming in minikanren: techniques, applications, and implementations. PhD thesis, USA, 2009. AAI3380156.*
9. ^ *Martin Davis. The Universal Computer: The Road from Leibniz to Turing. A. K. Peters, Ltd., 2011.*
10. ^ *Juha Sorva. Notional machines and introductory programming education. ACM Transactions on Computing Education, 13(2):8:1–8:31, 2013.*
11. ^ *M. Gancarz. Linux and the Unix Philosophy. Operating Systems Series. Elsevier Science, 2003.*
12. ^ *Stuart Russell and Peter Norvig. Artificial Intelligence: A Modern Approach. Prentice Hall Press, 3rd edition, 2009.*
13. ^ *David Deutsch. Constructor theory. Synthese, 190(18):4331–4359, 2013.*
14. ^ *C. Marletto. The Science of Can and Can't: A Physicist's Journey Through the Land of Counterfactuals. Penguin Books, Limited, 2021.*

Declarations

Funding: No specific funding was received for this work.

Potential competing interests: No potential competing interests to declare.