https://doi.org/10.48417/technolang.2022.02.06
Research article

Program and Code

Reiner Hähnle (✉)
Department of Computer Science, Technical University of Darmstadt, Karolinenplatz 5, 64289 Darmstadt, Germany
reiner.haehnle@tu-darmstadt.de

Abstract

The nature of computer programs can be characterized from two different viewpoints: as executable artifacts that create signals on a computing device or as pure mathematical objects with a rigorous, unambiguous semantics. To distinguish both usages I use the word “code” for the first and “program” for the second. This distinction is relevant to avoid confusion when discussing notions such as validity or correctness of software. The point is illustrated by refuting a well-known claim on the impossibility of verification and misleading claims about commercial products. At the same time the distinction “program versus code” is insufficient: I show that a “program” is always accompanied by an implicit or explicit application context which is necessary to scope its semantics. Ultimately, the analysis performed in this paper helps to distinguish relative from mathematical truths when discussing qualities of software.

Keywords: Program; Code; Formal Verification; Semantics; Relativism

Acknowledgment

The reviewers of this papers prompted me to improve a number of formulations and pointed out two factual errors.

Citation: Hähnle, R. (2022). Program and Code. *Technology and Language*, 2022, 3(2), 70-80. https://doi.org/10.48417/technolang.2022.02.06

УДК 004.42
https://doi.org/10.48417/technolang.2022.02.06
Научная статья

Программа и код

Райнер Хенле (✉)
Дармштадский технический университет, Каролиненплац 5, Дармштадт, 64289, Германия
reiner.haehnle@tu-darmstadt.de

Аннотация
Природа компьютерных программ может быть охарактеризована с двух разных точек зрения: как исполняемые артефакты, создающие сигналы на вычислительном устройстве, или как чисто математические объекты со строгой, однозначной семантикой. Чтобы различать оба употребления, я использую слово “код” для первого и “программа” для второго. Это различие уместно, чтобы избежать путаницы при обсуждении таких понятий, как достоверность или правильность программного обеспечения. Данный момент иллюстрируется опровержением известного утверждения о невозможности верификации и вводящих в заблуждение утверждений о коммерческих продуктах. В то же время разграничения “программа против кода ” недостаточно: я показываю, что “программа” всегда сопровождается неявным или явным контекстом приложения, который необходим для охвата ее семантики. В конечном счете, анализ, проведенный в этой статье, помогает отличить относительные от математических истин при обсуждении качеств программного обеспечения.

Ключевые слова: Программа; Код; Формальная верификация; Семантика; Релятивизм

Благодарности
Рецензенты статьи подсказали мне, как улучшить ряд формулировок и указали на две фактические ошибки.

Для цитирования: Hähnle R. Program and Code // Technology and Language. 2022. № 3(1). P. 70-80. https://doi.org/10.48417/technolang.2022.02.06

THE TWOFOLD NATURE OF COMPUTER PROGRAMS

Our starting point is an observation about the twofold nature of computer programs: On one hand, programs are designed to be executed on a machine, a "computer".[1] When connected to suitable hardware – microphones, displays, sensors, and actuators – programs become active, possibly even autonomous agents that are able to influence our reality, including human behavior.[2] On the other hand, programs are formal mathematical objects with a rigorous, unambiguous[3] semantics (Mitchell, 1996). But in contrast to other mathematical structures, for example, equations, functions, algebras, topologies, measures, a program's *executability* posits it directly in the physical world. To be sure, mathematical models also have physical impact, but, unlike programs, they need a dedicated mediator, for example, the bridge or the car that *is being* modeled, or in fact, a computer program that *is based* on a mathematical theory.

Before we continue, let us address an objection against singling out programs in this manner: one can argue that, similar to mathematical models of physical objects, instructions of a program need to be manifest in a physical medium to render them actually executable. In the old days, such media were punched cards or tape, nowadays electrons, soon quantum states. But this difference is inessential for two reasons: first, and most importantly, one can view either punched tape or electrons in a memory cell as a mere physical representation of a program, just as one can view a piece of text such as "$f(x) = x^2$" *as a* representation of a mathematical function. Second, in modern computing the execution tool chain of a program via compilation, loading, and initialization is a fully automatic, transparent process. This is not the case for mathematical models of physical objects – I come back to this observation later.

To sum up, one and the same representation, for example, a piece of program text, can be seen *either* as a mathematical object *or* as the execution of instructions on a machine. It is useful to distinguish these two views terminologically. Henceforth, I use the term *program* when I refer to a mathematical object and I use the expression *code*[4] when I refer to the entity that is actually being executed on a machine. Based on this terminology, in the present article I intend to substantiate two claims and discuss their consequences:

[1] With the understanding that computers can have many physical shapes: laptops, desktops, supercomputers, smart phones, and, most common these days, embedded into another device, such as a car, camera, household appliance, router, etc.
[2] Considerably so, as any parent of a teenager owning a smart phone knows.
[3] This does not at all preclude non-deterministic or probabilistic programs, whose meaning can be rigorously described as well.
[4] This convention is consistent with other usages: a compiler encodes a program into instructions to be executed by a microprocessor, which in turn decodes each instruction before it effectuates the action associated with it.

(A) The distinction between program and code in the above sense is crucial when we talk about the intended meaning of programs and ensuing notions, such as *correctness*, *validity*, etc.

(B) The distinction between program and code is *insufficient*, because it does not make sense to talk about a program in isolation. It is necessary to accompany programs with a *semantic model* that takes the *application context* into account.

MATHEMATICAL PROOFS OF PROGRAM CORRECTNESS

We need a concrete setting to meaningfully discuss the above claims, without getting lost in generalities.

The mathematical nature of programs opens up an intriguing perspective, setting the theory of programs apart from other scientific theories. This is, because a mathematical conjecture can be *proven* by a chain of formal, unambiguous, gapless arguments, which are broken down into instances of elementary steps whose truth is universally accepted and that can be verified by anyone with sufficient training, time, and interest. For example: if an expression e_1 is equal to an expression e_2 and expression e_2 in turn is equal to expression e_3, then also expression e_1 must be equal to expression e_3, and so on. A proven conjecture becomes a *theorem* and can in turn be used in other proofs. So far, so well-known.

Now, since programs are mathematical objects, it is possible to *prove* properties of programs in a rigorous, mathematical manner. This was recognized as early as 1949 by Alan Turing (Morris & Jones, 1984). Let us look at a very concrete example. The following text declares a simple procedure named m in the programming language Java:

```
int m( int i ) {
System. out. println ( i );
return i  +  1;
}
```

This procedure (called a “method” in Java terminology) takes an integer argument i and returns the next largest number (we ignore the print statement for the moment). What might be its formal semantics? A plethora of programming language semantics have been suggested over the years, but perhaps the most straightforward approach is to associate each program with a mathematical relation between its input and output values. One advantage is that merely elementary mathematics is needed to present the essentials of *relational semantics*:

Let in denote the value of i when m is called, and let out be the value that m returns. Then the semantics of m is denoted with the symbol [[m]] and simply given as the set of all pairs of integer numbers (in, out), where in is any integer, in other words, it is a relation over **int** × **int**. Specifically, the semantics of m is the set [[m]] = {. . . , (−1, 0), (0, 1), (1, 2), . . .} of (in, out) pairs.

The obvious, and seemingly trivial, correctness claim one might be tempted to prove about m is that out = *in*+1 always holds, that is, all pairs in [[m]] are of the form (*in*, *in* + 1). The "..." in the above expression suggests this to be the case, but in fact it is misleading, because clearly the claim is wrong! Integers in programming languages are not the mathematical integers (commonly denoted with the symbol $\mathbb{Z}$). To accommodate the finite memory of a computer, only a finite subrange of $\mathbb{Z}$ is represented. In the case of Java integers, four bytes or 32 bits are allocated to store one integer number and the encoding is such that numbers in the interval **int**=[-2^{31}, . . . , $2^{31} - 1$] can be represented. So what happens if in = $2^{31} - 1$? According to Java's semantics[5] the result is *out* = -2^{31}: the part of the result that "overflows" (is greater than $2^{31}-1$) becomes "wrapped around" and is added to $-2^{31} - 1$. Already our tiny example illustrates that it might not be obvious to decide what constitutes a correct program.

Since the late 1960s the demonstration of properties of programs with mathematical rigor has been established as a field of research within Computer Science called *formal verification*. Nowadays, proofs about programs are not carried out by hand, but with the help of other, specialized programs called verification tools (Hähnle & Huisman, 2019).[6] Formal verification is a good scenario to discuss claims (A) and (B), because it relies on the mathematical nature of programs and their formal semantics.

PROGRAM VERSUS CODE:THE CASE OF FORMAL VERIFICATION

We illustrate claim (A) with two papers (DeMillo et al., 1977; Fetzer, 1988) that famously announced the futility of formal verification. They stirred a lot of discussion at the time and provoked angry responses (Dijkstra, 1977) from computer scientists working in formal verification. One of the mistakes (there are several) made in these papers, and this is why we discuss them, is the conflation of the concepts "program" and "code". We focus on Fetzer (1988), where the argument is more explicit.[7]

In the terminology established in the introductory section, paper (Fetzer, 1988) argues that formal verification of *code* is impossible. The central argument is that correct execution of code depends on boundless contingent aspects and assumptions that are impossible to even begin to formalize: the correct functioning of the microprocessor the code is running on, the integrity of memory, the periphery, the connections, etc. Ultimately, one needs to take physics of transistors and other elements into account, down to quantum effects, radiation, and so on. And yes, not only is it infeasible to

[5] Not only Java's: it is the standard approach to integer semantics in most programming languages.
[6] Thus programs become tools to analyze programs. This reflexive stance is typical for Computer Science research (and Literary Studies).
[7] DeMillo et al. (1977) is mostly about validation of mathematical arguments, which they claim to be a purely social process. This is also highly disputable and can be disproven (Hales et al., 2015), but it is not the focus of the present paper.

formalize all of this context, but code as the physical manifestation of a program really is– and literally so–contingent: it cannot be separated from the environment it is executing in.

Fetzer's fallacy is this: because it is impossible to formally verify *code*, he infers that it is impossible to verify programs. In fact, he conflates code with programs. But based on our understanding that programs are precisely specified mathematical objects, it is plainly wrong to claim that programs cannot be verified. Yet it still might be true that program verification is a futile effort, if the gap between program and code turns out to be too substantial. In this case, it would not be useful if a program were verified, because the code derived from it might still be riddled with errors.

I argue that this is not the case for several reasons: (i) The tool chain rendering programs as code is robust–very few, if any, errors are introduced during that process; (ii) error correction and error recovery mechanisms are implemented at any critical juncture: memory, communication, etc.; (iii) scientists working in formal verification are well aware of the gap and tailor their met odology accordingly (Livshits et al., 2015); (iv) different aspects of programs can be isolated and modeled according to the requirements of an application context.

The last two points are closely related and highly relevant for a more detailed understanding of the concept of a *program*. I am now going to discuss them in greater detail. As we will see, this leads to an extension of the concept of what constitutes a program, as stipulated in claim (B).

THE APPLICATION CONTEXT

We come back to the example discussed earlier, where we observed that (at least) two *semantic models* of procedure m are possible:

1. For all integer values *in*, the result of executing m is *out* = *in* + 1.
2. For all values *in* $[-2^{31}, \ldots, 2^{31}-2]$ of *in*, the result of executing m is *out* = *in* + 1 and for *in* = $2^{31} - 1$ it is *out* = -2^{31}.

It is tempting to root for the second model: After all, it is fully precise. Moreover, as we saw, the first model is plainly wrong for input values outside the interval $[-2^{31}, \ldots, 2^{31} - 2]$. But is the second model sufficiently precise?

What is the semantics of the so far ignored print statement? It looks harmless enough, because it does not affect the final value of i. Yet, clearly it has an effect when executing the program, consisting in sending the value of i as a text string to the default system output. Can this be safely ignored, provided that we are only interested in the final value of i? What happens, for example, if no printing device is attached? As it happens, the print statement in Java is always executable: whether printing actually worked can be queried afterwards from status variables. But if we are after precise specifications, should we not be able to specify the print statement anyway? How to do so, without knowing which kind of printing device is attached (if any). The printer

hardware is carefully hidden inside many nested layers of Java’s *application programming interface*. Clearly, it is going to be complicated business to specify the print statement precisely. And not only that: without knowing the application context of our program, it seems impossible.[8]

But suppose we agree we should not worry about print statements–are we happy with semantic model (2.) above? Try the following exercise: specify precisely the outcome of procedure

int mult(**int** x,y)

which computes the multiplication of numbers x, y for values in **int** with “wraparound” semantics. It is surprisingly difficult. And it is not only difficult to specify, but even harder to formally verify.

For this reason, most verification tools offer the option to work with $\mathbb{Z}$ instead of **int** *even though this is generally incorrect*. The justification is that procedures such as “mult” are *intended* to work for input values, where they behave exactly as multiplication $*$ on $\mathbb{Z}$. Put differently, do we really want the correctness of programs to rely on unintuitive properties like mult(2, 2^{30})= -2^{31}? Possibly not, but it certainly depends on the *application context*.

Without such a context, which in the case of procedure m might specify the integer model as well as those aspects of printing (if any) that are relevant, we are doomed to enter an endless series of contingencies. The application context *scopes* the *semantic model* used in formal verification: it defines its boundaries (for example: ignore the print statement or not) and the level of precision (for example, **int** versus $\mathbb{Z}$). Without an application context a given program segues into code and Fetzer’s criticism applies.

RELATIVISM

Perhaps it is no coincidence that paper (Fetzer , 1988) challenging the possibility of verifying *code* mentioned in Section 3 appeared at the zenith of Postmodernism, contemporary with proposals that cast doubt on the possibility of objective scientific truth (for example, Rorty, 1989). Indeed, contingency is inherent to the concept of code and the “application context” coming to the rescue of programs smacks of relativism. It is important to be precise about what is contingent and what is relative.

First of all, much was made in postmodern philosophy about the impossibility to disentangle object and meta language and the consequent loss of an “Archimedean”

[8] Another phenomenon that is hard to specify precisely are side effects or, rather, there absence. Assume the program that procedure m is contained in declares a globally visible variable g. Obviously, m does not change the value of g, but the semantics [[m]] given above does not reflect this fact. To accommodate it, [[m]] would need to include g (as well as all any other variable visible from m) and state that its value is unchanged by m.

point for an objective author or observer. This is not the issue here. Program verification and other formal analyses are based on mathematical logic and set theory. Suitable consistent, formal systems of reasoning that are validated against model theoretic semantics are known since long. Programs and proofs about their properties are unambiguous, rigid, independently verifiable.

Yet it is important not to overstate or to exaggerate what *formally* verified means: the *context* is crucial, because code in our sense is indeed *contingent*. We saw that code can be "lifted" to a program equipped with a specific semantic model determined by an application context. And that context in itself may be economically, socially, or politically motivated. Therefore, scientific truth in Computer Science is indeed *relative*, but not because of flaws in the mathematical arguments or of the language that proofs are expressed in. Rather, it is the choice of the semantic model that is relative to a given purpose. To the extent that this choice is motivated and explained, mechanical correctness proofs are as valid as (in fact more than) any piece of mathematics.

Pragmatism has a stubborn tendency to prevail: happily or, at least, unthinkingly, we entrust our lives to programs running in pacemakers, ventilators, cars, planes and other appliances whose failure has fatal consequences for their users. Some of this software is formally verified, most of it is not. Empirically, the trust seems justified: There are surprisingly few reports about fatal incidences that can be directly traced to software failures. In many cases, a reported incident at closer look exhibits a misunderstanding of the expected application context among different stakeholders rather than a genuine programming error.[9]

There is an important difference between the engineering discipline Computer Science and the Natural/Social Sciences: programs and the languages they are expressed in are designed. All of their aspects can be (and increasingly are) formalized and mechanically checked. Hence, we can place high trust in a formal proof and in at least those aspects of the code represented by a program scoped by an application context. In contrast, outside the Engineering Sciences there are theories about how biological, physical, or societal systems are constructed, but we do not possess the blueprint of those systems.[10] In consequence, these theories are susceptible to relativist criticism (to differing degrees).

Also between Computer Science and the "physical" Engineering Sciences there is a crucial difference: as outlined in Section 3 the gap between program and code is hardly noticeable in practice. In fact, many times the concepts of "program" and "code" are conflated (which compelled me to write this article). Once the program text is written and the application context has been decided, it takes only a mouse click to compile, deploy, and execute the resulting code. In reality this is a highly complex

[9] Typical examples are https://www.bbc.com/news/health-43973652 and https://www.heise.de/downloads/18/2/9/4/3/5/6/9/NTSB_Uber.pdf

[10] Admittedly, at least non-quantum physics is widely considered to be indisputable in absence of relativity effects.

process that not so long ago involved considerable manual steps,[11] but it is lightning fast and fully automatic by now.

Most importantly, the transition from program to code is invisible, a black box, but at the same time highly robust and reliable. In the world of physical engineering the step from a mathematical model to its physical realization is considerably bigger and much more explicit. CAD/CAM technology drastically shortens the path from model to product, but one still has to deal with the physical aspects of production. The salient point is the existence and usage[12] of universal programming languages in the sense of the Church-Turing thesis that can compute any function that is computable at all, where only memory and speed impose practical limits. But there is no universal physical production material and no universal physical production machine that would permit to do the same.[13]

CONCLUDING REMARKS

I argued that it is beneficial to have two different points of view on software: what is situated in a computing device, interacting with its environment, pushing around electrons, I call code. In everyday conversation and popular texts about software, this is often what is meant. But when we attach qualities to software, such as correctness or validity, it is not meaningful or even possible to do so at the level of code. Instead, we look at a fixed number of aspects determined by the application context, represented in a specific semantic model (for example, idealizing **int** to $\mathbb{Z}$).

The distinction between program and code permits to be precise about where scientific truth can be expected in Computer Science and where one should be wary. For example, some vendors actively exploit the lack of distinction between program and code to advertise (or mislead) by insinuating the code running at a customer’s site is inherently secure while, of course, only partial security aspects of some of the *programs* were analyzed.[14]

I used formal verification of programs as an illustration, however, the distinction “code versus program+application context” is useful in all areas of Computer Science, where software is analyzed with mathematical methods, in particular, in IT security.

Truth in verification proofs and other rigorous mathematical arguments is neither subjective nor socially constructed. Formalization and mechanization in Computer Science constitute a very strong argument that formal proofs are indisputable. On the other hand, *what* is proven, the choice of the semantic model, as we called it, is very

[11] I recall punching cards on a typewriter-like contraption and carrying them to the operator room as late as 1982.
[12] All programming languages in wider usage have this property.
[13] In this light it is unsurprising to observe that functionality that used to be realized in hardware is constantly moved to software, whenever possible at all. Another interesting development in this context is 3D printing, which goes some way towards universal physical production–helped by software, obviously.
[14] A representative slogan: “Security. Built right in.” from https://www.apple.com/macos/security

much determined by an application context, which in turn is motivated by subjective, economical, societal, political factors. Therefore, the appropriate question to ask is not *Verum estne*?, but–as ever– *Cui bono*?

REFERENCES

DeMillo, R., Lipton, R., & Perlis., A. (1977). Social Processes and Proofs of Theorems and Programs. In R. M. Graham, M. A. Harrison, and R. Sethi (Eds.), *Conference Record of the Fourth ACM Symposium on Principles of Programming Languages* (pp. 206–214) ACM.

Dijkstra, E., (1977). A Political Pamphlet from the Middle Ages. *ACM SIGSOFT, Software Engineering Notes, 3,* 14. http://www.cs.utexas.edu/users/EWD/ewd06xx/EWD638.PDF

Fetzer, J. (1988). Program Verification: The very idea. *Communications of the ACM, 31*(9), 1048–1063. https://doi.org/10.1145/48529.48530

Hähnle, R. & Huisman, M. (2019). Deductive verification: from pen-and-paper proofs to industrial tools. In B. Steffen, & G. Woeginger (Eds.), *Computing and Software Science. Lecture Notes in Computer Science, vol. 10000* (pp. 345–373). Springer. https://doi.org/10.1007/978-3-319-91908-9_18

Hales, T., Adams, M., Bauer, G., Dang, D., Harrison, J., Hoang, T., Kaliszyk, C., Magron, V., McLaughlin, S., Nguyen, T., Nguyen, T., Nipkow, T., Obua, S., Pleso, J., M. Rute, J., & Solovyev, A. (2015). A formal proof of the Kepler conjecture. CoRR. *Forum of Mathematics, Pi, 5,* E2. https://doi.org/10.1017/fmp.2017.1

Livshits B., Sridharan M., Smaragdakis Y., Lhoták O., Amaral J. N., Chang B. E., Guyer S. Z., Khedker U. P., Møller A., and Vardoulakis D. (2015). In defense of soundiness: a manifesto. *Communications of the ACM, 58*(2), 44–46. https://doi.org/10.1145/2644805

Mitchell, J. (1996). *Foundations for programming languages*. Foundation of computing series. MIT Press.

Morris, F., & Jones, C. (1984). An early program proof by Alan Turing. *IEEE Annals of the History of Computing, 6*(2), 139–143. https://doi.org/10.1109/MAHC.1984.10017

Rorty, R. (1989). *Contingency, Irony, and Solidarity*. Cambridge University Press, https://sites.pitt.edu/~rbrandom/Courses/Antirepresentationalism%20(2020)/Texts/rorty-contingency-irony-and-solidarity-1989.pdf

СВЕДЕНИЯ ОБ АВТОРЕ / INFORMATION ABOUT THE AUTHOR

Райнер Ханле, reiner.haehnle@tu-darmstadt.de
ORCID 0000-0001-8000-7613

Reiner Hähnle, reiner.haehnle@tu-darmstadt.de
ORCID 0000-0001-8000-7613

Статья поступила 22 апреля 2022
одобрена после рецензирования 18 мая 2022
принята к публикации 27 мая 2022

Received: 22 April 2022 /
Revised: 18 May 2022
Accepted: 27 May 2022