the Meaning of Mathematics

Eric C.R. Hehner
University of Toronto

Abstract. Programming can provide a foundation for mathematics.

Keywords. formalism, foundations, metamathematics, semantics.

Introduction

Like many other subjects, computer science makes use of mathematics. Unlike other subjects, computer science has sought to use mathematics as a foundation. We do not like to define computation by pointing to a computer, but by constructing a mathematical model, or set of equivalent mathematical models. We do not like to define the semantics of a programming language by pointing to a compiler, but by constructing mathematical functions or relations or predicate transformers. We thus hope to base computer science firmly upon mathematics.

What is the foundation of mathematics? There are many schools of thought with many variations. This paper outlines a new entry in that old debate: programming provides a foundation for mathematics. The position is a variation of the formalist position; if the position needs its own name, we may call it "programmism". We hope it will find some favor among both mathematically-minded programmers and programming-minded mathematicians.

We present the programmist viewpoint briefly in the context of the dominant mathematical philosophies: platonism and formalism.

Platonism and Formalism

A map of the world would be the same if the shorelines had been discovered in a different order. The world exists independent of our knowledge of it. That, at least, is the opinion of most people. Similarly, it is the opinion of most mathematicians that mathematical objects exist (in some abstract sense), and that truths about them are discovered. This opinion is called "platonism". According to a platonist, the order of discovery may be partly a historical accident, and our way of expressing truths may be a product of human design, but the truths themselves are independent of us, timeless and universal.

The opposing opinion is called "formalism". According to a formalist, mathematics is not discovered but invented. It is a kind of language whose expressions can be used to describe the world, if we choose, in any way we choose. By themselves the expressions are neither true nor false. The design and use of this language constitute the subject. Mathematical language can include diagrams, graphs, and anything else; usually and most usefully the expressions are sequences of symbols. Some of the expressions are called "sentences", and some sentences are called "theorems". A choice of expressions (formulas) is called a "formalism", and a choice of theorems is called a "theory". Formalisms and theories are human creations, and they are influenced by their predecessors.

According to formalism, mathematics is a language, and that means that people must share in a common understanding of it. We must have some definite way of stating what the expressions are, which expressions are sentences, and which sentences are theorems. Programmism is

formalism with the additional contention that the expressions, sentences, and theorems of mathematics are defined by programs (parsers, typers, and provers). The programs that define a formalism are usually stated informally; sometimes they are not stated at all and we are left to guess them from examples. Nonetheless, according to programmism, mathematics is the design, study, and execution of these programs. It is not the existence of artificial computers that defines a formalism, but the existence of a program to define the rules in a way that can be followed objectively, whether by human or machine.

A common criticism of the formalist position is that formal arguments are lengthy, tedious, and error-prone; insistence on formal proof, quoting axiom and proof-rule, prevents us from getting very far. This criticism likens formal proofs to machine-language programs. According to programmists, axioms and proof-rules are indeed the machine-language of mathematics. But the criticism is misdirected: programmists do not insist that proofs must be at that level. Just as a good programmer structures a program in a high-level language, making use of previously written procedures, a good mathematician structures a formal proof, making use of previously proved theorems.

A formalist does not insist that all arguments must be formal, but only that they must be formalizable. When a proof is presented to a machine, then it must be formal, but to a human, a convincing informal argument, perhaps containing a "you know what I mean" in the form of three dots, is usually welcome. But we must be able to remove the three dots, to fill in the detail and produce a machine-checkable proof if asked. One obtains a licence to be informal by demonstrating the ability to be formal.

A formalist does not pretend that formalisms are created at one stroke, nor that the process of creating or improving one is "formal". But a formalist insists that mathematics is the creation and use of formalisms. Although we can agree, given a theory, what the theorems are, we certainly have our differing preferences for theories. The design of a good one is shaped by tradeoffs and criticisms. It requires good taste and judgement, like other human activities.

Some mathematicians may feel too constrained, or limited, by the requirement that all proofs be according to rules that could be programmed. It is indeed a severe limitation: it limits proofs to those that mathematicians can agree on. If there were a dispute, then at least in principle a machine could be arbitrator. According to programmism, the much-vaunted universality of mathematics, the ability of mathematicians to agree, is just the mechanical nature of formal proof.

Semantics and Meaning

Just as we can write the word "table" on a piece of paper and use it to refer to a physical object, so we can write number-expressions, set-expressions, predicate-expressions, function-expressions, and so on, and if we are platonic we use them to refer to numbers, sets, functions, predicates, and other mathematical objects. "Syntax" tells us how to write and read the expressions, and "semantics" tells us what objects they refer to. The platonist position is clear and simple, but it has one insurmountable difficulty: how can we give the semantics? To give people the meaning of the word "table" without pushing the problem onto another language, we take them to a table, and let them see it, feel it, and experience it in any way they can. We cannot give the meaning of number-expressions, set-expressions, predicate-expressions, or function-expressions the same way.

Formalists, whose world is not populated with abstract mathematical objects, would find it cumbersome to have to call their subject matter "number-expression", "set-expression", "predicate-expression", and "function-expression"; they prefer to say simply "number", "set", "predicate", and "function", meaning in each case a kind of expression. The formalist position is that mathematical expressions have no intrinsic meaning, but they can be applied as we choose. Applied mathematics is not about numbers or triangles or sets. It is about the height of water behind a dam, and the position of Jupiter in the night sky, and what happens when hydrogen and oxygen are combined. Numbers and triangles and sets are the words (symbols, expressions) mathematics uses to talk about the world. Theories are designed so that their theorems can be used to represent truths of the world (whatever they may be). For example, the theory of natural numbers is designed for the quantification of discrete objects. In that theory, the sentence 1+1=2 is a theorem representing the truth that if one apple is placed next to another, they remain distinct. If the application is fixed, perhaps it does no harm to say that 1+1=2 is true, meaning that it represents a truth. But when one raindrop is combined with another, they do not remain distinct; for that application we may want a theory in which 1+1=2 is not a theorem.

The English sentence "Our planet has an atmosphere." is true, but we do not call it an English truth, or a truth of English; it is a physical truth. There *are* truths of English, for example that a sentence ends with a period. Similarly there are mathematical truths, for example that "1+1=2" is a theorem of arithmetic. But the mathematical sentence "1+1=2" is no more a truth of mathematics than the English sentence "Our planet has an atmosphere." is a truth of English.

It has often been thought to be beyond coincidence that the world can be described by mathematics, and so the world must be "mathematical", and that mathematicians are therefore discovering truths. Yet it is no less amazing that we can describe the world with words or with paint on canvas. According to formalists, mathematics is a marvellous way to organize and summarize some of our physical knowledge, but it is ridiculous to ever say that we have discovered the "true" organization or summary.

Not all mathematics is created as a model, like a drawing. According to formalists, pure mathematics is created just for its beauty, like music. It may be called "arbitrary symbol manipulation" by anyone who would call music "arbitrary sound manipulation". Most of us would not. Although I cannot define "beauty", and I grant that it leaves room for variety and difference, it nonetheless serves as a criterion for mathematics. To my taste, beauty requires simplicity, and I believe that simplicity is also a necessary quality of the mathematics that is most useful for describing the world. This is not because the world is simple, but because a good model is.

To a formalist, the meaning of mathematics depends on its application, if any. To a programmist, there is a second kind of meaning, another way to define the semantics of mathematics. The programs that define mathematics (formalisms) are divided into syntactic and semantic. (The parser is syntactic, the prover is semantic, and we can argue about where the boundary falls between. Of course, a platonist considers all of them to be syntactic.)

If the semantics of mathematics is programs, what is the semantics of programs? Programs specify or describe activity, that of a human or computer; we can perform and observe the activity. Programs can be explained by physical experience. The direction of platonist semantics, from program to mathematical expression to abstract object, is reversed by the programmist: from mathematical expression to program to activity.

I hasten to add that mathematicians need not be aware of the programs that a programmist claims are the semantics of mathematics; productive platonic mathematicians demonstrate this every day. It is likewise true that a programmer need not be aware of the activity described by programs; sometimes it would be a considerable hindrance. What a programmer needs most is a good theory of programming.

The programmist position says that the semantics of mathematics is ultimately the activity of symbol manipulation according to a set of rules. It does not say that mathematics is just the manipulation of symbols according to rules. One must follow the rules whenever one is exploring a theory, or using a theory to describe or predict events in the world. But one should spend considerable time thinking about what the theory models, and how well it models, and whether the theory can be improved by changing the choice of symbols and rules of manipulation.

Metamathematics: the Study of Formalisms

To study the stars, it is helpful to design a mathematical formalism for the purpose. Mathematics is not limited, however, to the study of nature. It also helps us to study the artificial worlds of bridges, economics, music, and even thought processes. And if we want to make mathematical formalisms objects of study, it is helpful to design a mathematical formalism for the purpose.

We design a formalism so that its sentences represent statements about the objects of study. Some of these statements are true, and others are false; we design the theory so that its theorems (provable sentences) represent true statements and its antitheorems (disprovable sentences) represent false statements. Suppose, for example, that the object of study is a version of number theory; let us call it NT. Here are eight examples of true statements about (not in) NT.

(a) 1+1=2 is a theorem of NT.
(b) 1+1=3 is an antitheorem of NT.
(c) 0÷0=4 is neither a theorem nor an antitheorem of NT.
(d) (0÷0=4) ∨ ¬(0÷0=4) is a theorem of NT.
(e) A sentence is an antitheorem of NT if and only if its negation is a theorem of NT.
(f) A sentence is a theorem of NT if and only if its negation is an antitheorem of NT.
(g) Either a sentence is not a theorem or it is not an antitheorem of NT.
(h) Every sentence of the form $s \vee \neg s$ is a theorem of NT.

Here are six false statements about NT.

(i) 1+1=2 is an antitheorem of NT.
(j) 1+1=3 is a theorem of NT.
(k) 0÷0=4 is either a theorem or an antitheorem of NT.
(l) A sentence is an antitheorem if and only if it is not a theorem of NT.
(m) A sentence is either a theorem or an antitheorem of NT.
(n) Either a sentence or its negation is a theorem of NT.

Statement (a) shows us a simple theorem. In mathematics texts, it saves space to write a mathematical sentence such as 1+1=2 without saying anything about it, thereby meaning that it is a theorem. But in this paper we shall not do so. Statement (g) says that NT is consistent. The false statement (m) says that NT is complete. We included division in NT to give a simple sentence that is neither a theorem nor an antitheorem; Gödel's surprise result was that, even without division, with only addition and multiplication, there are sentences that are neither theorems nor antitheorems.

Let us call our theory to study theories TT . For any theory T and sentence *s* of T we introduce the sentence (of TT)

T ⊢ *s* (pronounced " T proves *s* ")

to represent the (true or false) statement that *s* is a theorem of T . And we introduce

T ⊣ *s* (pronounced " T disproves *s* ")

to represent the statement that *s* is an antitheorem of T . When it is clear which theory is under study, we may omit its name and write simply

⊢ *s* (pronounced "theorem *s* ")

⊣ *s* (pronounced "antitheorem *s* ")

Because of statement (e) we may be tempted to dispense with the symbol for "antitheorem", and to speak instead of the negation of a theorem. However, it is not necessary for all theories to include negation, and it may be interesting to study some that do not. Because of (c) we certainly cannot take "is an antitheorem" to mean "is not a theorem". We need a symbol for "antitheorem" for the same reason that Binary Theory (boolean algebra) needs symbols for both "true" and "false", and indeed *true* and *false* are a primitive theorem and antitheorem respectively.

As we have seen, some sentences are theorems, some are antitheorems, and some are neither. It is tempting to say that the sentences which are neither are in a third class, and to invent a third symbol ⊥ to go with ⊢ and ⊣ . In some circles, three-valued logic is popular. Unfortunately, any attempt to formalize the third class will run into the same problem: there will be a gap, and a temptation to invent a fourth class, and so on. I prefer to say that those sentences that are neither theorem nor antitheorem are "unclassified".

Binary Theory includes symbols for "or", "and", "not", and "if and only if", namely ∨ , ∧ , ¬ , and = . In TT , we also need symbols for "or", "and", "not", "if and only if", as seen by our example statements. We therefore reuse ∨ , ∧ , ¬ , and = in TT . To avoid confusion, we could have chosen symbols for theory TT that differ from those of the theories under study. But we may want to use TT to study TT as well, and there is a better way to avoid confusion. The way was known to Gödel and is well-known to programmers: it is to distinguish program from data. A compiler writer knows the difference between her program and her data, even though her data is someone else's program, even if it is in the same language. To her, the incoming data is a character string, and her program examines its characters. Similarly in metamathematics we can use one theory to describe another without confusion, even if that other theory is itself, by realizing that, to the describing theory, expressions of the described theory are data of type character string.

In modern logic, the distinction between program and data is not always made, and ⊢ is applied directly to sentences. To partially compensate, logicians distinguish between "extensional" and "intensional" operators, and make rules stating when something cannot be substituted for its equal. For the sake of simplicity and clarity, let us maintain the programmer's distinction: we apply ⊢ to a character string representing a sentence. Thus

NT ⊢ "1+1=2"

is the sentence of TT representing statement (a).

Omitting the name NT , the statements (a) to (n) are represented (formalized) in TT as follows. (Juxtaposition of character strings indicates (con)catenation.)

(aa) ⊢ "1+1=2"

(bb) ⊣ "1+1=3"

(cc) $\neg \vdash$ “0÷0=4” $\wedge \neg \dashv$ “0÷0=4”
(dd) $\vdash$ “(0÷0=4) $\vee$ ¬(0÷0=4)”
(ee) $\dashv s = \vdash$ (“¬” s)
(ff) $\vdash s = \dashv$ (“¬” s)
(gg) $\neg \vdash s \vee \neg \dashv s$
(hh) $\forall s \cdot \vdash (s$ “$\vee$ ¬” $s)$

(ii) $\dashv$ “1+1=2”
(jj) $\vdash$ “1+1=3”
(kk) $\vdash$ “0÷0=4” $\vee \dashv$ “0÷0=4”
(ll) $\dashv s = \neg \vdash s$
(mm) $\vdash s \vee \dashv s$
(nn) $\vdash s \vee \vdash$ (“¬” s)

We try to design TT so that (aa) to (hh) are theorems, and (ii) to (nn) are antitheorems. When we design a theory to study the stars, we should always retain some doubt about how well our theory matches the facts. The same goes for a theory to study theories.

Classical and Constructive Mathematics

In the previous section, statements (h) and (n) are very similar, but (h) is true and (n) false. Which of them is the Law of the Excluded Middle? The truth of this “law” was hotly disputed for a while by mathematicians who conducted their mathematics informally; perhaps the informality was necessary to the dispute. For a formal theory like NT, we can distinguish (h), which is the “law”, from (n), which (given (e)) is a statement of completeness.

There are interesting theories for which the Law of the Excluded Middle does not hold; instead a proof of a disjunction requires a proof of one of the disjuncts. We may represent this in TT as follows.

$$\vdash (s \text{ “}\vee\text{” } t) = \vdash s \vee \vdash t$$

These theories are called “constructive”. Similarly a proof of $\exists x \cdot p\ x$ requires a term t such that $p\ t$ is a theorem. A proof of $\forall x \cdot \exists y \cdot p\ x\ y$ is a program for constructing from input x a satisfactory output y.

Theories in which the Law of the Excluded Middle holds are called “classical”. Most people are willing to agree that “either God exists or God does not exist” without a proof of either disjunct. These people prefer a classical theory in which

$$(0\div0=4) \vee \neg(0\div0=4)$$

is a theorem even though neither disjunct is a theorem.

The preceding discussion conceals a subtle point. We have said that in a classical theory, a disjunction may be a theorem even though neither of its disjuncts is. If our metatheory TT is classical, then perhaps $\vdash s \vee \vdash t$ can be a theorem (for some choice of s and t) even though neither of its disjuncts is. If so, then the TT sentence

$$\vdash (s \text{ “}\vee\text{” } t) = \vdash s \vee \vdash t$$

does not represent our intention to describe a constructive theory. For this reason, we may prefer our metatheory to be constructive. Unfortunately, if the metatheory is used to describe itself, it cannot tell us whether it has this constructive property.

Semantics as Interpreter

In this section we present a simpler and less expressive metatheory than in the previous sections, using only one symbol. For any theory T and string s we introduce the sentence

$$\mathsf{T}\,\mathsf{I}\,s$$

Predicate I is said to interpret string s in theory T. When it is clear which theory is meant, we may omit its name. For each theory, we want $\mathsf{I}\,s$ to be a theorem if and only if s represents a theorem, and an antitheorem if and only if s represents an antitheorem. It is related to $\vdash$ and $\dashv$ by the two implications

$$\vdash s \;\Rightarrow\; \mathsf{I}\,s \;\Rightarrow\; \neg \dashv s$$

In fact, if we have defined $\vdash$ and $\dashv$, those implications define I. But we want I to replace $\vdash$ and $\dashv$ so we shall instead define it by showing how it applies to every form of sentence. Here is the beginning of its definition.

$$\begin{aligned}
&\mathsf{I}\,\text{“}\mathit{true}\text{”} = \mathit{true}\\
&\mathsf{I}\,\text{“}\mathit{false}\text{”} = \mathit{false}\\
&\mathsf{I}\,(\text{“}\neg\text{”}\ s) = \neg\,\mathsf{I}\,s\\
&\mathsf{I}\,(s\ \text{“}\wedge\text{”}\ t) = \mathsf{I}\,s \wedge \mathsf{I}\,t\\
&\mathsf{I}\,(s\ \text{“}\vee\text{”}\ t) = \mathsf{I}\,s \vee \mathsf{I}\,t
\end{aligned}$$

And so on. Notice that I acts as the inverse of quotation marks; it “unquotes” its operand. That is what an interpreter does: it turns passive data into active program. It is a familiar fact to programmers that we can write an interpreter for a language in that same language, and that is just what we are doing here.

To finish defining I we must decide the details of an entire theory. We shall not do so here, but we give one more case of special interest. I is defined on strings beginning with I as

$$\mathsf{I}\,(\text{“}\,\mathsf{I}\,\underline{\text{“}}\,\text{”}\ s\ \text{“}\,\underline{\text{”}}\,\text{”}) = \mathsf{I}\,s$$

where $\underline{\text{“}}$ and $\underline{\text{”}}$ represent the opening and closing quotation marks within a character string. Thus the interpreter becomes part of the interpreted logic.

The way we are defining I above is exactly the way we define a function in a functional programming language: by showing how the function applies to each form of argument. The definition of I is a program, and it is this program that gives meaning to the mathematical theory that it interprets.

Depending on the theory, the interpreter program might be nondeterministic: maybe two or more interpreter equations apply to some expressions in the theory. Due to nondeterminism, the result of interpreting some particular sentence might be *true* or *false*. When both results are possible, the theory is said to be inconsistent, and that is usually considered to be undesirable. It is also possible that the interpreter might evaluate a sentence forever, without reaching a result. When that is possible, the theory is said to be incomplete, as most theories are.

Conclusion

Most descriptions and reasoning are informal, conducted in a natural language. But natural languages are fraught with pitfalls, so we invent formalisms when we need precise languages. To a formalist, mathematics is the design, use, and evaluation of formalisms. Numbers,

triangles, and sets are the terminology or words of mathematics. We can use these words to talk about matter and motion (applied mathematics), or we can use these words just to talk, to make a sort of mathematical music (pure mathematics). There are mathematical expressions, but a formalist has no reason to suppose that there are abstract, mathematical objects about which the expressions are speaking.

Programmism is a version of the formalist position holding that the precision of mathematics comes from the way we state the rules defining formalisms: they must be followed the same way by everyone, so they must be, in effect, a program.

To many students, mathematics is a dry and lifeless subject; it is presented as a long list of dull facts about abstract objects, not as a creative human subject like music, novel-writing, and film-making. That is the platonist legacy. To a formalist, mathematics is pure creation, to be judged by its beauty and usefulness. It has fashions and disagreements. It is an attitude, a method of thinking, an attempt to make sense of the world. It is not a collection of facts or truths.

Acknowledgement. I thank Andrew Malton and Gary Levin for discussions; Douglas Hofstadter and Willard Quine for words of encouragement; Vassos Hadzilacos, Evan Cameron, Bill McKeeman, and Pierre Berlioux for comments.

This essay was originally part of a series of lectures given at the Marktoberdorf Summer School in 1986, and published in

- E.C.R.Hehner: Programming Based on Logic and Logic Based on Programming, 71 pages, four chapters in Broy (ed.): *The Logic of Programming and Calculi of Discrete Design*, NATO Advanced Studies Institute Series, Springer-Verlag, Heidelberg, 1986

The lectures include sections on

Cantor's uncountability argument
Gödel's incompleteness proof
Turing's incomputability argument (the Halting Problem)

other essays