On the usefulness of formal methods

Freek Wiedijk

Institute for Computing and Information Sciences
Radboud University Nijmegen

The editor of this newsletter has asked me to write something for it. Now it would be easiest for me to just write about my research subject, *formalization of mathematics*, and how that field will fundamentally change the way that mathematicians look at proof. ('There will be a time when mathematicians will only consider something to be "proved" if the proof has been encoded in full detail in a computer, all the way to the axioms of set theory, and if that encoding has been completely checked by that computer. When that time has arrived, a mathematical article will only be accepted for publication if it is accompanied by a formal, computer-checked counterpart.[1] The referees then only will have to judge whether the result is new and interesting, and not whether it is correct ... because that will then be known already.')

However, that will not be the main topic of this essay. Instead, I will focus on the subject of the usefulness of working on technology for proving *software* correct. That is, the usefulness of using methods from mathematical logic to develop a technology for creating programs that are free of 'bugs'.[2]

Some years ago I was working on the formalization of the Fundamental Theorem of Algebra[3] in the Coq system. A *formalization* is an encoding of a mathematical proof in such detail, that the computer can verify the mathematical correctness without any further human interference. When one formalizes a proof, it is not the computer that provides the proof. Instead it is the human who 'guides' the

[1] In the field of mathematical logic this already occasionally happens today.

[2] In the NOAG-ict 2005–2010, which is the research agenda for Dutch computer science research, the field of formal methods does not have a research theme of its own. It has been put there as part of theme number six, *intelligente systemen*, the theme for artificial intelligence. To me that shows a fundamental misunderstanding of where the main promise of formal methods lies, what formal methods is all about. It really should have been in the theme that contains the study of algorithms and programming languages, theme number seven: *methoden voor ontwerpen en bouwen*.

[3] The Fundamental Theorem of Algebra was the subject of the 1799 PhD thesis of Carl Friedrich Gauss. It states that the field of the complex numbers is 'algebraically closed', which is a compact way of saying that every non-constant polynomial has a zero, and that therefore every polynomial can be written as a product of linear factors. The Fundamental Theorem of Algebra has three formalizations, which were all finished in the year 2000. The first formalization was done using the Polish system Mizar by Robert Milewski. The second formalization was done using the British system HOL Light by John Harrison. The third formalization was done using the French system Coq by Herman Geuvers, with a group of people to which I also belonged. This third formalization encoded a proof that is more complicated than the usual ones, because it also is *intuitionistically* valid.

proof, although he is supported by the automation of routine proof tasks that the computer provides.

Writing a formalization is an activity that is very similar to writing a large computer program. The languages that one uses when formalizing are very similar to programming languages, and the activity of modelling a proof in such a language is also very similar to the activity of programming. To show the kind of code that one writes when formalizing, here are the final lines in the Coq proof of the Fundamental Theorem of Algebra:

```
...
intro H0. apply H0.
intro i. generalize (Hs i).
intro H1; inversion_clear H1; assumption.
exact (seq_is_CC_Cauchy n H0n q qnonneg qlt10 (AbsCC p ! Zero[+]One)
  Hp s Hs2).
intro i; generalize (Hs i); intro Ha; elim Ha; intros; assumption.
exact (less_plusOne _ (AbsCC p ! Zero)).
apply zero_lt_posplus1.
apply AbsCC_nonneg.
Qed.
```

Note that this looks much like 'computer code'. However, there is a *big* difference between writing a computer program and writing a formalization. When one writes a formalization, one can be completely sure that it will be correct. In contrast with this a program of a non-trivial size will always have 'bugs'. (Every programmer knows that.[4])

At a recent Dagstuhl conference about the use of computers in mathematics, a mathematician in the audience who did not know much about formalization could not believe that this kind of total correctness is possible. He clearly believed that because formalization is a human activity, it surely should be taken into account that there would be mistakes. But no. (For instance, I am quite confident that our formalization of the Fundamental Theorem of Algebra is 100% correct.) At that same conference there was a girl who had formalized something, also using the Coq system. *She* knew about formalization. And she confidently proclaimed that she would be willing to bet 100,000 dollars that her work was totally free of errors. Now I would not go *that* far that I would bet such an amount on it, but I still am sure that she was right about the correctness of her work.[5]

[4] When I learned how to program, I was very much surprised that one makes mistakes all the time. My programs would often behave in ways that to me seemed completely impossible, even when I took into account that there would be some mistakes in it. (Of course like everyone I got used to this phenomenon, but initially this was very surprising to me.)

[5] Let me analyze what are the possibilities for mistakes when doing a formalization. I think that there are three. The first is the possibility of bugs in the software that checks the formalization. Now of course there will be bugs in that software. But it is rather unlikely that such bugs would *accidentally* allow someone to have

When we were formalizing the Fundamental Theorem of Algebra, we were using the versioning system CVS to keep track of our files. With this system a group of people can all work on a bunch of files simultaneously. When I described CVS to a programmer friend who had no experience with it, he told me that with such a system he would very much worry about different people making inconsistent changes to the files. And then I told him that if I would use CVS for programming, then I would worry about that too. But we used CVS with Coq, and therefore we did not have to worry about this at all! We used the rule that one was only allowed to check in files when the whole set was accepted by the system as being correct. This meant that at any time the files in the system would be a consistent whole. And therefore one could work with CVS without having to worry about modifications of someone else in the group 'breaking' something that one had done.

The experience of formalizing with Coq was wonderful. It was like programming, but then knowing that there were no mistakes, that there were no bugs at all.

But then, after the formalization of the Fundamental Theorem of Algebra was finished, I went back to 'normal' programming. That was scary! It was like someone had removed the safety net: I could make mistakes again!

Now the moral of this story is that I think that in working with Coq I have tasted the future of programming. Eventually (and I do not say that I know how or when) I expect the technology of programming to develop into a form that gives one the experience that I had with Coq. When that happens, programs will generally be free of bugs. Today this sounds unthinkable, but I really do believe that this will come about.

---

the system proclaim incorrect mathematics to be correct. Problems with normal programs are hardly ever caused by compiler bugs either. Also, the systems that check formalizations for correctness generally have an architecture that localizes the correctness of the check to a very small part of the program, a 'proof checking micro-kernel'. (Henk Barendregt calls this principle of having such a micro-kernel the *de Bruijn criterion*, after the pioneer of formalization N.G. de Bruijn.) This makes it even more unlikely that bugs in the checker will allow incorrect mathematics to accidentally 'slip through' the system.

The second possibility to have incorrect mathematics accidentally be accepted by a proof checking system is that the logical foundations of that system are inconsistent. While this certainly is not unthinkable (for instance, the Coq system is closely related to the logical systems of Per Martin-Löf, whose very first logical system happened to be inconsistent; so maybe his current crop of logics also is inconsistent, and we just are not smart enough to see this), again I think it is very unlikely that because of this one would prove falsehoods *by accident*.

The third possibility to make 'mistakes' in a formalization is that the definitions of the notions that one reasons about are not what one thinks them to be. This is the only significant problem of the three. However, once one proves many lemmas about a small number of definitions, one can be quite confident that these definitions mean what the formalizer thinks they mean.

I expect this development to come from the culture of the research field that is called 'formal methods'. Eventually. It is the reason that I think formal methods are a useful part of computer science.

Does this mean that I am advocating people to download Coq and start using it to prove their programs correct. No, certainly not! What I am saying is that the culture of proof assistants like Coq will be a fertile ground from which a new kind of programming will develop eventually. I am *not* saying that these proof assistants already are useful for more than proving the correctness of rather simple programs (programs like the `Vector` class from the Java library; proving that correct does not seem very important for practical work.)

Still, researchers that use proof assistants are already doing interesting things with it. For example there is the 'Cminor' compiler from INRIA, which has been built by Xavier Leroy. This compiler, called Compcert, has been proved correct using Coq (so there is a real possibility that its correctness approaches the correctness of our Fundamental Theorem of Algebra formalization). It compiles a realistic subset of the C programming language called Cminor to PowerPC assembly code, applying non-trivial code optimizations on the way. Xavier Leroy is the man behind one of the best compilers of functional languages in the world – the ocaml compiler of the ML language – and I consider it very interesting that he wants to spend his time on using proof checking technology for creating a provably correct program.

Another program, one that can *really* be relied on to be correct is the Four Color Theorem[6] program by Georges Gonthier. (At some point Georges Gonthier went to work for Microsoft. Please note: the most impressive formalization in the world is owned by Microsoft!) The Four Color Theorem was proved in the seventies using a very controversial approach. Part of the proof was the running of a computer program that went through millions of graph colorings. This program ran for days, then, and a modern version of that program, coded in C, still takes minutes to run. At that time the mathematicians were not satisfied by this approach. They wondered how one could be sure that the program did not contain bugs. What Georges Gonthier did was rewrite the program in a purely functional subset of ML, and then prove that version correct in Coq. And he did more: he also formalized all the graph theory and topology needed to prove a very clean and natural version of the Four Color Theorem statement. (This proof was an extension of the correctness proof of his version of the program.) In case you are interested, the statement that he proved was:

```
forall m : map R, simple_map m -> map_colorable 4 m
```

This was formalized in a way that the micro-kernel of the Coq system can check the correctness of the formalization all the way down to the Coq axioms. And to do this checking Coq also runs the – provably correct – program that is part of

[6] The Four Color Theorem states that every 'map' can be colored using four colors in such a way that two neighboring countries always have a different color. For a very long time this was a famous open problem, until it finally was proved in 1975 by Kenneth Appel and Wolfgang Haken.

the formalization. The running of that program only takes on the order of days (on an ordinary PC.)

In case you wonder how the notions `map`, `simple_map` and `map_colorable` that occur in this statement are defined, here are some relevant definitions that I lifted from Gonthier's files:

```
Inductive point : Type := Point (x y : R).

Definition region : Type := point -> Prop.

Definition intersect (r1 r2 : region) : region :=
  fun z => r1 z /\ r2 z.

Definition subregion (r1 r2 : region) := forall z, r1 z -> r2 z.

Definition map : Type := point -> region.

Definition inmap (m : map) : region := fun z => m z z.

Definition covers (m m' : map) := forall z, subregion (m z) (m' z).

Definition size_at_most n m :=
  exists f, forall z, inmap m z -> exists2 i, i < n & m (f i) z.

Record proper_map (m : map) : Prop := ProperMap {
  map_sym : forall z1 z2, m z1 z2 -> m z2 z1;
  map_trans : forall z1 z2, m z1 z2 -> subregion (m z2) (m z1)
}.

Record simple_map (m : map) : Prop := SimpleMap {
  simple_map_proper :> proper_map m;
  map_open : forall z, open (m z);
  map_connected : forall z, connected (m z)
}.

Definition border (m : map) z1 z2 :=
  intersect (closure (m z1)) (closure (m z2)).

Definition corner_map m z : map :=
  fun z1 z2 => m z1 z2 /\ closure (m z1) z.

Definition not_corner m z := size_at_most 2 (corner_map m z).

Definition adjacent m z1 z2 := meet (not_corner m) (border m z1 z2).

Record coloring (m k : map) : Prop := Coloring {
  coloring_proper :> proper_map k;
  coloring_inmap : subregion (inmap k) (inmap m);
  coloring_covers : covers m k;
  coloring_adj : forall z1 z2, k z1 z2 -> adjacent m z1 z2 -> m z1 z2
}.

Definition map_colorable n m :=
  exists2 k, coloring m k & size_at_most n k.
```

(I left out the definitions of topological notions like `open`, `connected` and `closure`, to prevent this list of definitions from becoming too long. These notions mean what you would expect them to mean.)

Now the Four Color Theorem program is certainly not a trivial program. The ML version that Georges Gonthier proved correct is around 2,500 lines long (even the C version is around 1,000 lines), and it uses very smart tricks to make it run as fast as possible. It certainly has not been designed to be straight-forward, to make the proofs easy. That it can be proved correct shows that proof checking of programs has reached some maturity.

Various technologies have been developed for proving programs correct. They fall roughly in two classes:

- Either one takes existing programming technology, and then *adds* a layer on top of that, which guarantees that the program is correct. This means that one writes a program like one always does it, but then uses software that analyzes it in some way (often the *statement* that is proved about the program then is just something like 'all array indices will stay within bounds and no nil pointers will be dereferenced'[7]), or maybe annotates it with invariants and then generates a series of lemmas from that which then can be proved using a proof assistant.[8]
- Alternatively, one can change the way one develops software, to make it more abstract, more *mathematical*. For instance one might restrict programming to a purely functional language (as is the case both with the Cminor compiler and with the program from Georges Gonthier's Four Color Theorem formalization). Or one might require the user to develop his or her program by first making an abstract specification, and then *refine* that specification to the real program. (As far as I understand it, that is what one does when using the 'B method'[9]).

[7] An example of a system like this is Microsoft's SDV (Static Driver Verifier), see

`http://research.microsoft.com/slam/`

Apparently at Microsoft some people seem to be thinking that formal methods are useful for analyzing production code. Bill Gates even is quoted calling the use of 'actual proof about the software' to be 'the Holy Grail of computer science'.

[8] A very nice example of systems that implement this are Jean-Christophe Filliâtre's Why and Caduceus tools. See the web page at:

`http://why.lri.fr/caduceus/index.en.html`

It is surprising to me that there is not more work being done on systems that follow this general approach.

[9] The B method was used to prove the software correct that manages a line of the Paris metro that has driverless trains. (One would like not to have metro trains collide because of bugs in the software.) This is most that I know about the B method, really. Well, I also know that the logical foundations that it uses is a variant of ZF set theory.

> Or, a method that used to be popular in the type theoretical community (although no one believes much in it anymore): one can just take a formalization of a proof, and then automatically *extract* a program from that.[10]

I do not know very well what position I take between these two approaches. On the one hand I do not think these methods will find wide adoption when programmers are forced into a mathematical straight-jacket, when they are required to program with their hands bound behind their backs so to say. For instance, I *certainly* do not think that *purely* functional programming (where you have to model all input/output, mutable data structures, exceptions etc. using so-called *monads*) will ever conquer the world.

On the other hand I think that the most efficient way to really reduce the number of software problems is *not* to have programmers prove things about programs the way they are now, but instead to have them change to a bit more abstract world view. Let me try to give two examples of what I mean by this.

I would expect that many of the problems with the – surprisingly popular – Microsoft Windows platform (bugs; but also the proliferation of viruses and worms that occurs there) often are caused by simple things like buffer overruns and dereferencing of nil pointers. Now suppose that Windows was not programmed in languages like C++, which really just is sugared assembly language, but instead was programmed in a more abstract language like ML.[11] Then the worst that could happen would be an uncaught exception, and nothing really *bad*

[10] I have a nice story about program extraction. The Fundamental Theorem of Algebra formalization that we created was a formalization of an *intuitionistic* proof. Therefore one could extract programs from it (although the proof had not been especially chosen for that.) Now the Fundamental Theorem of Algebra says that every non-constant polynomial has a root. This means that we could extract a *root-finding* algorithm. It took a couple of months to extract the program (there were some bugs in the extraction code), but when we finally got it and ran it, nothing happened. The program just ran and ran without producing any output. Then we tried something simpler. Part of our formalization was the Intermediate Value Theorem, the statement that says that if a function is negative somewhere and positive somewhere else, that it then has a zero in between. (This happens not to be provable intuitionistically: our formalization had an intuitionistic form of it.) By applying this theorem to the polynomial $x^2 - 2$ we extracted a program that calculates $\sqrt{2}$. The output of this program is a stream of fractions that converges to that value. When we ran it, the program *immediately* output the value 0 (every instance of the Intermediate Value Theorem algorithm would do this, regardless of the function that one would put in), and then, after running for hours, it finally output a second 0. However, this second 0 had a smaller limit on how far it was from the desired value of $\sqrt{2}$. There never was a third output. And, although we tried very hard, we never understood what the program was doing all that time.

[11] The ML programming language was *spin off* from the formal methods world. One of the first proof assistants in the world was the LCF system from the seventies. The ML language was at first only developed to program 'tactics' for this system. (In this respect it is similar to the Ltac language for the Coq system that is being developed right now.)

could happen. This is not just something that I would expect. It is my experience that when one programs in ML it might be a bit harder to fit what one wants to do into the framework of the language, but then one needs to spend much less time on debugging. There is an anecdote related to this: when a friend of mine first learned a functional programming language, he decided to try programming in it by 'cloning' his programmable pocket calculator, with all the functions and buttons. It took him quite some time to fit the behavior of this calculator in the type system of the functional language. When he finally succeeded with this, he expected that he would need to debug it for a similar long time. But the program ran, and ran correctly, at the first try.

Another example of what I mean when I say that 'an abstract world view is important' is the example of how one looks at files. In the operating systems of the sixties and seventies files where rather involved objects. One needed to tell the system in some detail where the files would be put on the sectors of the disk, and there would be different *kinds* of files related to how one did this, with names like 'direct' and 'indexed sequential' files. But then with systems like Unix, a file became a much more abstract object. It now just was a sequence of bytes, and how it would be put on the disk was *abstracted away*. This was a huge improvement from the point of view of understanding what was going on. I would like to see more of that kind of abstraction in programming.

For instance, I think it would be a good thing if there would be no integer overflow.[12] I would think that systems would be better behaved if they were programmed in languages like Lisp and Smalltalk where the system automatically switches to 'bignum' behavior when the numbers go outside of the range of machine words. This might sound like a trivial change, but not many programming languages provide this, and wanting this is a good example the 'mind set' that I think will change the way we program.

Let me talk a bit about what research in formal methods currently looks like, and how it currently is applied to get better software.

My impression of the field of formal methods is that most papers are about deductive systems (carrying fancy names like 'ACP_τ', 'the π-calculus', 'STTwU', '$\mathrm{MLW}^{\mathrm{ext}}\mathrm{PU}_{<\omega}$', '$\bar{\lambda}\mu\tilde{\mu}$', etc.) These papers define those systems using pages of deductive rules, and then prove all kinds of things about them and (sometimes) describe experiments with implementations that are based on them. If I am honest, I expect that most of those deductive systems will go nowhere. However, they are part of the *culture* that will produce the few systems that *will* matter, and, like I already said, I expect that this eventually will be very important for software quality.

The technology of formal methods already is being used to prove the correctness of various kinds of software and hardware. Now I think that the word *prove* is a bit misleading here. That suggests *certainty*, and indeed, a proof will give absolute certainty about some property of the program. However *what* should be proved is generally not totally clear. This means that the application of formal

[12] I have been told that some kinds of election fraud make use of the fact that voting machines occasionally fail to notice integer overflow.

methods in computer science, despite its use of 'proofs', does not really give a guarantee that the program will behave as desired, but instead should just be considered to be a rather thorough (and expensive) form of debugging.

To explain why I think that it is not possible to get *total* certainty about the correctness of programs using proofs: consider proving the LaTeX program of Donald Knuth and Leslie Lamport (and after them many others who extended and improved their system) to be correct. What should be the statement that one should prove for this? Suppose that LaTeX puts some text in some document in a wrong font style because of a bug somewhere: how could I have prevented this by proving things about it? The statement that describes how LaTeX should behave (the specification) is of a similar size as the LaTeX source code itself. And it is almost as difficult to get this statement correct as it is to get the program itself correct!

And then, even when the statement that you prove actually is what you want it to be, often assumptions in the specification about the real world will be able to cause trouble. This means that even if you prove that something will not happen, it still might happen. For example, even if you prove that trains will not collide, there is not a *total* guarantee that the trains actually will not do this. I once saw this demonstrated very clearly. There was a demo of an application of formal methods where, as a case study, a little toy train was running on a toy track. The software that drove that toy train had been proved correct. However, the toy train had a toy accident right in front of my eyes, because the sun was shining in its sensors. Apparently the assumptions in the model that had been used in the proof of the safety of this toy train were not satisfied at that time.

I also would like to say something about the relationship between formalization of mathematics and formalization in computer science. There seems to be a trend in formal methods to move to tools that check specific properties of programs (like for instance that variables will not be used when they have not yet been initialized, or that there will be no overflow) without any human intervention, this in contrast with systems like proof assistants that are very general and therefore by necessity interactive. Of course we eventually will need both kinds of system: one should not underestimate the importance of being able to step in and tell the system what to do when the automation does not cut it anymore.

When interactively proving properties of programs, it is important to be able to reason in a 'mathematical' style. Therefore we should have good technology for formalizing mathematical proofs too. I do not see a dichotomy between formalization of mathematics and the applications of formalization in computer science. On the contrary: I think that the thing that is most needed to make formal methods more powerful is the ability to work in a style that is as mathematical as possible.

Luckily, there recently has been much progress in the mathematics that can be formalized. At the start of 2005, formalizations of three famous theorems were finished:

- the Four Color Theorem, using Coq, by Georges Gonthier
- the Prime Number Theorem, using Isabelle/HOL, by Jeremy Avigad

– the Jordan Curve Theorem, using HOL Light, by Tom Hales[13]

(There are two other theorems that have not been formalized yet, but that are always mentioned when people talk about formalization of mathematics. They are:

– the classification of finite simple groups
– Fermat's last theorem

Georges Gonthier told me that he will start working on the first one. And Jan Bergstra put the second one up as a 'grand challenge' for computer science. It will be interesting to see whether anyone will take him up on this challenge.[14])

Nowadays 'big' theorems can be practically formalized, with a lot of work. But on a more mundane scale, the technology also has become good enough to *routinely* formalize ordinary mathematics.

Some time ago I found a 'top hundred' of nice theorems on the web. I decided to investigate how many of them had already been formalized. The result is on:

`http://www.cs.ru.nl/~freek/100/`

Currently three quarters of this list has been formalized, and this fraction is growing fast. The three systems that formalized the most are HOL Light and ProofPower (both are variants of the HOL system) and the Mizar system. Apparently those three systems are currently the best for formalization of mathematics.

In this essay I tried to argue for two things. First, in the field of formalization of mathematics interesting things are happening. And second, developments from the field of formal methods might lead to a culture change in software development that will lead to better software quality everywhere. The first statement I know to be true from my own research experience. The second statement I hope and expect to become true too.

[13] Later in 2005 a formalization of the proof of the Jordan Curve Theorem using the Mizar proof assistant was also finished.

[14] A challenge that already has been taken up is the formalization of the proof by Tom Hales of the 'Kepler conjecture', which states what is the densest way to pack spheres in space. Hales calculates that the formalization of his proof will take twenty years, and calls his project *Flyspeck*.