Sādhanā Vol. 34, Part 1, February 2009, pp. 1–23. © Printed in India

Proof assistants: History, ideas and future

H GEUVERS*

Institute for Computing and Information Science, Faculty of Science, Radboud University Nijmegen, P. O. Box 9010, 6500 GL Nijmegen, The Netherlands
e-mail: herman@cs.ru.nl

Abstract. In this paper we will discuss the fundamental ideas behind proof assistants: What are they and what is a *proof* anyway? We give a short history of the main ideas, emphasizing the way they ensure the correctness of the mathematics formalized. We will also briefly discuss the places where proof assistants are used and how we envision their extended use in the future. While being an introduction into the world of proof assistants and the main issues behind them, this paper is also a position paper that pushes the further use of proof assistants. We believe that these systems will become the future of mathematics, where definitions, statements, computations and proofs are all available in a computerized form. An important application is and will be in computer supported modelling and verification of systems. But there is still a long road ahead and we will indicate what we believe is needed for the further proliferation of proof assistants.

Keywords. Proof assistant; verification; logic; software correctness; formalized mathematics.

1. Introduction

Proof assistants are computer systems that allow a user to do mathematics on a computer, but not so much the computing (numerical or symbolical) aspect of mathematics but the aspects of *proving* and *defining*. So a user can set up a mathematical theory, define properties and do logical reasoning with them. In many proof assistants one can also define functions and compute with them, but their main focus is on doing proofs. As opposed to proof assistants, there are also automated theorem provers. These are systems consisting of a set of well chosen decision procedures that allow formulas of a specific restricted format to be proved automatically. Automated theorem provers are powerful, but have limited expressivity, so there is no way to set-up a generic mathematical theory in such a system. In this paper we restrict attention to proof assistants.

Proof assistants are now mainly used by specialists who formalize mathematical theories in it and prove theorems. This is a difficult process, because one usually has to add quite a

*Second affiliation: Faculty of Mathematics and Computer Science, Technical University Eindhoven, NL

lot of detail to a mathematical proof on paper to make the proof assistant accept it. So, if you are already completely convinced that the theorem is correct, there may not be much use in formalizing it. However, there are situations where one is not so convinced about a proof. This happens most notably in computer science, where correctness proofs are e.g. about intricate pieces of software that act in a physical environment, or about a compiler that involves syntax of a programming language that has many possible instructions. But also in mathematics, it occurs that proofs are not really accepted by the community, because they are too big or involve computer programs to verify.

In the paper, we will first consider the notions of 'proof' and 'proof checking' in general, and also say something about the various styles of 'computer proofs' that proof assistants work with. Then we give a historic overview of the earliest systems, indicating the main techniques and issues that deal with correctness of proofs and proof checking. Then we will briefly sketch where these systems are used and we will conclude with how we see the development of these systems in the future: What is needed to enhance their use and applicability?

1.1 *Proofs*

A proof—according to Webster's dictionary—is 'the process or an instance of establishing the validity of a statement especially by derivation from other statements in accordance with principles of reasoning'.

In mathematics, a proof is absolute. Basically, the correctness of a proof can be determined by anyone. A mathematical proof can be reduced to a series of very small steps each of which can be verified simply and irrefutably. These steps are so small that no mathematician would ever do this, but it is a generally held position that mathematical proofs that we find in books and articles could in principle be spelled out in complete detail. It happens that a mathematical theorem turns out to be false. In that case, not all steps have been spelled out and checked in detail, and it is always possible to point at a proof step that cannot be verified.

We will not get into the philosophical question why all this works, but we will content with the observation that mathematicians agree on the validity of the basic proof steps and on the methods of combining these into larger proofs. This moves the problem of mechanically checking a mathematical proof to the *translation* of the informal proof to a formal one, i.e. from the natural language to a formal language. One could think that in mathematics, this is not a big problem, because mathematicians already write their results in formulas and use a quite restricted technical 'jargon' in their proofs. However, there is still a considerable gap—especially in level of detail—between the proofs that mathematicians write in books and articles and the proofs that can be understood by a computer.

1.1a *Two roles:* Let us now look a bit more into the role of a proof in mathematics itself. A proof plays two roles.

(i) A proof *convinces* the reader that the statement is correct.
(ii) A proof *explains* why the statement is correct.

The first point consists of the administrative ('*bookkeeper*') activities of verifying the correctness of the small reasoning steps and see if they constitute a correct proof. One doesn't have to look at the broad picture, but one just has to verify step by step whether every step is correct. The second point deals with giving the intuition of the theorem: Why is it so natural that this property holds? How did we come to the idea of proving it in this way?

In a proof that we find in an article or book, both 'roles' are usually interwoven: some intuition is given and sometimes explanation on how the proof was found and why it is the

way it is, but the proof also provides enough information to be able to verify step by step that the statement holds.

The mathematician Paul Halmos emphasizes that a mathematical proof is not written up in the way it has been found:

> 'Mathematics is not a deductive science. When you try to [solve a problem] . . . what you do is trial and error, experimentation, guesswork. You want to find out what the facts are, and what you do is in that respect similar to what a laboratory technician does, but it is different in its degree of precision and information.' (Halmos 1985).

A proof has three stages:

(i) *Proof finding:*
In this phase 'anything goes': experimenting, guessing, . . . This phase is usually not recorded but for students to learn mathematics it is indispensable to practice.

(ii) *Proof recording:* The written up proof that contains explanation on why the statement holds, why the proof is as it is (point B above) and it contains the proof steps needed to verify the statement step-by-step (point A above).

(iii) *Proof presentation:* After a proof has been found, it goes through a phase of being communicated to others. This happens both before and after it has been written up. The proof is read by others and sometimes redone and the main points in the proof are emphasized. This phase, which may go hand in hand with the previous one, focusses mainly on point B above.

That there are more proofs of one theorem exemplifies that a proof not only verifies but also explains. Different people have different preferences as to which proof they like best, which is usually the one that explains best the aspect of the statement that they find most relevant. Other reasons for preferring a specific proof over another are: because it is surprising, because it is short, because it uses only basic principles, because it uses another—at first sight completely unrelated—field of mathematics, . . . Nice examples of different proofs of one result are given by the book *Proofs from THE BOOK* (Aigner & Ziegler 2004) that contains 'beautiful' proofs of well-known theorems. The book contains six proofs of Euclides' theorem that there are infinitely many primes. Each proof is interesting in its own right, shedding light on a different aspect of prime numbers.

1.2 *Proof checking*

Let us now look at the possible role a machine can play in the proof roles (A) and (B) and in the different proof stages 1, 2 and 3.

If we look at role (A) of proofs, it will be clear that in the verification that all the small proof steps together constitute a correct proof a computer can be a very helpful tool. This is the bookkeeper's work that we can leave to the machine: we can write a *proof checking* program that establishes the validity of a theorem by mechanically checking the proof. The first proof assistants were basically just that: programs that accept a mathematical text in a specific (quite limited) syntax and verify whether the text represents a proof. We would probably only be fully convinced if we had written the proof checking program ourselves, but in practice we will have to rely on the quality of the programmer and the fact that the proof checking program has been used and tested very often, because usually there is no way we could simply write a checker ourself.

However there are other ways that may help in making the proof checking more reliable and strengthen our conviction that a checked proof is correct indeed.

1.2a *Description of the logic:* If we have a system independent description of the logic and its mathematical features (like the mechanisms for defining functions and data types), we can establish whether we believe in those, whether our definitions faithfully represents what we want to express and whether the proof steps make sense.

1.2b *Small kernel:* Some systems for proof verification have a very small kernel, with rules that a user can verify by manually checking the code. All other proof rules are defined in terms of those, so a proof step is a composition of basic proof steps from the kernel. In this case one only has to trust the small kernel.

1.2c *Check the checker:* The proof assistant itself is 'just another program', so its correctness can be verified. To do this, one first has to specify the properties of the program, which means that one has to formalize the rules of the logic. (Obviously this requires the first item in this list: a system independent description of the logic.) Then one would prove that the proof assistant can prove a theorem φ if and only if φ is derivable in the logic. A way to do this is to prove that all proof-tactics are sound in the logic and that there is a proof-tactic for every inference rule. Another way to proceed is to construct a complete model for the logic within the system, but then one needs extra principles, because this would also imply the consistency of the logic. (And Gödel says that a logic can't prove its own consistency.)

1.2d *De Bruijn criterion:* Some proof assistants create an 'independently checkable proof object' while the user is interactively proving a theorem. These proof objects should be checkable by a program that a skeptic user could easily write him/herself. De Bruijn's Automath systems were the first to specifically focus on this aspect and therefore this property was coined 'De Bruijn criterion' by Barendregt (Barendregt & Geuvers 2001). In De Bruijn's systems, the proof objects are basically encodings of natural deduction derivations that can be checked by a type checking algorithm.

If we look at the four mechanisms for improving the reliability of a proof assistant, we see that the first is definitely needed for either one of the other three. Also, a small kernel obviously helps to be able to check the checker. In our overview of proof assistants we will see in some more detail how different systems deal with the four possible ways of guaranteeing correctness.

If we look at role (B), the explanation of proofs, we see that proof assistants do not have much to offer. At best they force a user to consider every small detail, which sometimes brings to light implicit assumptions or dependencies that were lost in the high level mathematical proof. But most of the time the required amount of detail is considered to be a hindrance to a proper high level explanation of the proof.

If we look at the three stages of proofs, we see that proof assistants have little to offer in stage 1, the *proof finding* process. For the average user it is just impossible to formalize a proof that one hasn't spelled out on paper before and it is erroneous to expect the machine to help the user in finding a proof. In some cases, like software or hardware verification where proofs may involve large case distinctions with many trivial but laborious cases, the machine may help to solve these cases automatically, helping the user to focus on the important ones. But in general the user has to provide the intelligent steps, and for most users it is difficult to make a good mental picture of the mathematical situation when one is looking at a computer prompt and ASCII code.

Of course, proof assistants are perfect for recording proofs (stage 2): they are verifiable pieces of computer code that can be inspected at every level of detail. But then, what is not recorded is the key idea, because it is hidden in the details. The problem is the communication of computer formalized proofs on a higher level of abstraction (stage 3). What is the important proof step or the key idea? Can we give a diagrammatic (or geometric) representation of the proof? The system PVS (PVS) has a diagrammatic representation of the proof that is interactively created by the user. This is found to be very helpful by PVS users, but it is already way too detailed to convey the high level idea.

Here is the general picture of how proof assistants support the roles and stages of proofs. Of course, some aspects hold more (or less) for some systems than for others, but there isn't really much difference between them so this diagram is valid quite generally.

Proofs		Proof Assistants
Roles	Check	$++$
	Explain	$--$
Stages	Finding	$-$
	Recording	$+$
	Communicating	$-$

We will come back to the two roles and the three stages of proofs in our overview of proof assistants.

1.3 *Input language*

The input language of a proof assistant can be *declarative* or *procedural*. In a procedural language, the user tells the system *what to do*. In a declarative language, the user tells the system *where to go*. This may seem almost the same, but it isn't, as we can illustrate using an example. Here is a formalized proof in *procedural style* of the simple theorem that if we double a number and then divide by 2, we get the same number back. This is a Coq formalized proof of Théory taken from (Wiedijk 2006).

```
Theorem double_div2: forall (n : nat),  div2 (double n) = n.
simple induction n; auto with arith.
intros n0 H.
rewrite double_S; pattern n0 at 2; rewrite <- H; simpl; auto.
Qed.
```

A reader cannot see what this proof does, because it only has a meaning if we execute it in Coq. Then we see what the *proof state* is after line 3 and we can understand what line 4 does and why it is a useful next step. Here is a proof in *declarative style* of the same theorem by Corbineau.

```
Theorem double_div2: forall (n : nat), div2 (double n) = n.
proof.
  assume n:nat.
  per induction on n.
    suppose it is 0.
      thus thesis.
    suppose it is (S m) and IH:thesis for m.
```

```
      have (div2 (double (S m))= div2 (S (S (double m)))).
           ~= (S (div2 (double m))).
      thus ~= (S m) by IH.
  end induction.
end proof.
Qed.
```

We can see what this proof does without executing it in Coq. (NB. The `~=` notation is used for denoting a sequence of equalities.) We may find the syntax a bit baroque, but a mathematician can easily see the reasoning that is going on here.

From this example we can make another observation: the declarative proof is longer. That turns out to be the case in general. This is nice for the reader, but not so nice for the user who has to type it. Another issue arising from this example is the question what we want to use the proof for. Do we want a human to read it? Or is it just a set of computer instructions that a machine should be able to read?

The 'proofs' above are usually referred to as *proof scripts*. A proof script may not necessarily correspond to a proof in a logic; it is the list of computer instructions given to the machine to direct the proof assistant to accept the theorem. As we can see from the proof scripts above, the procedural one certainly doesn't correspond to a proof in a logic, but the declarative one does, to some extent. The declarative script above can be seen as some form of a Fitch (Fitch 1952) style natural deduction.

Another issue is how *robust* and *adaptable* the proof scripts are. If we change a definition, how difficult is it to adapt the proof script? In general, proof scripts are not very robust, but declarative proof scripts are slightly easier to adapt than procedural ones. In a declarative proof script, we find information that the system could generate itself, so which is in a sense redundant, but very helpful when adapting (or debugging) a proof script. In a procedural proof, one usually tries to give the minimal amount of information that yields the next proof state. It makes sense to try to combine the two styles as much as possible. This may be possible by letting the user *type* a procedural proof script but let the system expand this into a proof script with more declarative components. Interesting ideas about this can be found in (Wiedijk 2004, Barendregt 2003).

2. History

In this section, we will shortly describe the history of proof assistants by describing the first systems and the main ideas behind them. We will look at these systems by taking the issues raised in the Introduction as a starting point. We also give an overview of how these ideas were used in later (present day) proof assistants. It is interesting to see that around 1970 at several places simultaneously the idea of *computer verified mathematical proofs* came up. We will try to give a complete overview of these initial systems, hoping to capture all the 'founding fathers', some of which have been more influential than others. Before doing that, we will discuss a general issue, which is the input language of a proof assistant.

2.1 *Automath*

The Automath project (Nederpelt *et al* 1994, De Bruijn 1983) was initiated by De Bruijn in 1967 and had as aim to develop a system for the mechanic verification of mathematics. A related aim of the project was to develop a mathematical language in which all of

mathematics can be expressed accurately, in the sense that linguistic correctness implies mathematical correctness. This language should be computer checkable and it should be helpful in improving the reliability of mathematical results. Several Automath system have been implemented and used to formalize mathematics. We will discuss some crucial aspects of the systems that had an influence on other systems.

2.1a *Proofs as objects, formulas as types:* In the Automath systems the idea of treating proofs as first class objects in a formal language, at the same footing as other terms, occurs for the first time. In logic, this idea is known as the Curry–Howard formulas-as-types isomorphism, for the first time written up in 1968 by Howard (Howard 1980), going back to ideas of Curry who had noticed that the types of the combinators are exactly the axioms of Hilbert style deduction. De Bruijn reinvented the idea, emphasizing the proofs-as-objects aspect, which comes down to the following: There is an isomorphism T between formulas and the types of their proofs giving rise to

$$\Gamma \vdash_{\text{logic}} \varphi \text{ if and only if } \overline{\Gamma} \vdash_{\text{type theory}} M : T(\varphi)$$

where M is a direct encoding (as a λ-term) of the deduction of φ from Γ. In logic, Γ just contains the assumptions, but in type theory, $\overline{\Gamma}$ also contains the declarations $x : A$ of the free variables occurring in the formulas. The formulas-as-types correspondence goes even further: assumptions in $\overline{\Gamma}$ are of the form $y : T(\psi)$ (we assume a hypothetical 'proof' y of ψ) and proven lemmas are definitions recorded in $\overline{\Gamma}$ as $y := p : T(\psi)$ (y is a name for the proof p of ψ).

An interesting consequence of this analogy is that 'proof checking = type checking'. So, a type checking algorithm suffices to satisfy the De Bruin criterion of the previous section. Depending on the type theory, this can be more or less difficult. The original Automath systems had a small kernel, so for those it is rather simple. Later developments based on the same idea are the systems LF (Harper *et al* 1993, Twelf (Twelf), Lego (Luo & Pollack 1992), Alf (Magnusson & Nordström 1994), Agda (Agda), NuPrl (Constable *et al* 1986) and Coq (Coq), which have increasingly complicated underlying formal systems and therefore increasingly complicated kernels and type checking algorithms. (NuPrl is based on a type theory with undecidable type checking, so a λ-term is stored with additional information to guide the type checking algorithm.)

It should be noted that the original Automath systems were just *proof checkers*: the user would type the proof term and the system would type check it. The other systems mentioned are *proof assistants*: the user types *tactics* that guide the *proof engine* to interactively construct a proof-term. This proof-term is often not explicitly made visible to the user, but it is the underlying 'proof' that is type-checked. This is made precise in figure 1.

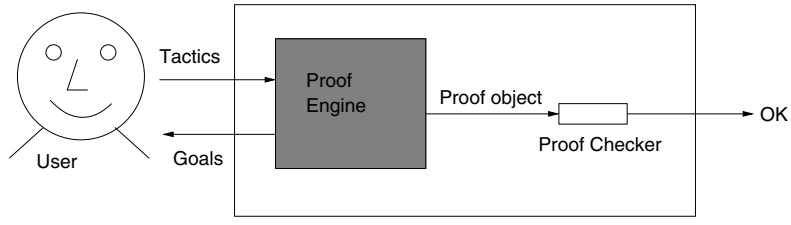

Figure 1. Proof development in a type theory based proof assistant.

The De Bruijn criterion means that there is a notion of 'proof object' and that a skeptical user can (relatively easily) write a program to proof check these proof objects.

2.1b *Logical framework:* Another important idea that first appears in Automath is that of a *Logical framework*. De Bruijn emphasized the idea that his system would only provide the basic mathematical mechanisms of substitution, variable binding, creating and unfolding definitions, etc. and that a user would be free to add the logical rules he/she desires. Following the formulas-as-types principle, this amounts to

$$\Gamma \vdash_L \varphi \text{ if and only if } \Gamma_L, \overline{\Gamma} \vdash_{\text{type theory}} M : T(\varphi),$$

where L is a logic and Γ_L is a context declaring the constructions of the logic L in type theory. It is a choice which logical constructions one puts in the type theory and which constructions one declares axiomatically in the context: there have been various Automath systems that represented weaker or stronger type theories. The idea of a rather weak type theory, which is then used as a logical framework has been further developed in Edinburgh LF (Harper *et al* 1993) and in the system Twelf (Twelf).

2.2 *Martin–löf type theory*

The Curry–Howard formulas-as-types isomorphism gives a connection between proofs in *constructive logic* and typed λ-terms. (In constructive logic, one does not use the double negation law $\neg\neg A \to A$.) Martin-Löf has extended these ideas, developing *constructive type theory* as a foundation for mathematics, where inductive types and functions defined by well-founded recursion are the basic principles (Martin-Löf 1984, Nordström *et al* 1990). (This goes also back to work of Scott (Scott 1970), who had noticed that the Curry–Howard isomorphism could be extended to incorporate induction principles.)

Martin-Löf has developed several type theories over the years. The first one has been implemented in the proof assistant NuPrl (Constable *et al* 1986). This was an extensional type theory. Later systems were intensional and have been implemented in ALF (Magnusson & Nordström 1994) and Agda (Agda). But many of the ideas underlying Martin-Löf type theory have also found their way in other proof assistants, like LF, Lego and Coq.

2.2a *Proofs as programs:* In a constructive logic, if one has a proof of $\forall x\, \exists y\, R(x, y)$, then there is a computable function f such that $\forall x\, R(x, f(x))$. The function (construction) f is 'hidden' in the proof. In constructive type theory, this is extended to incorporate arbitrary algebraic data types and we have the phenomenon of *program extraction*: from a proof (term) $p : \forall x : A.\exists y : B.R(x, y)$ we can extract a functional program $f : A \to B$ and a proof of $\forall x : A.R(x, f(x))$. So we can see $\forall x : A.\exists y : B.R(x, y)$ as a *specification* that, once 'realized' (i.e. proven) produces a program that satisfies it.

As an example, we show the specification of a sorting algorithm. The goal is to obtain a program sort : $\text{List}_{\mathbb{N}} \to \text{List}_{\mathbb{N}}$ that sorts a list of natural numbers. So, given a list l, we want to produce as output a sorted list k such that the elements of k are a permutation of the ones of l. So we have to define the predicates $\text{Sorted}(x)$ and $\text{Permutation}(x, y)$, denoting, respectively that x is sorted and that x is a permutation of y. (For example, we can define $\text{Sorted}(x) := \forall i < \text{length}(x) - 1(x[i] \leq x[i+1])$.) Now, we have to prove our specification, i.e. we have to find a term $p : \forall x : \text{List}_{\mathbb{N}}\, \exists y : \text{List}_{\mathbb{N}}(\text{Sorted}(y) \wedge \text{Permutation}(x, y))$. From this we can then extract the program sort : $\text{List}_{\mathbb{N}} \to \text{List}_{\mathbb{N}}$.

The proofs-as-programs paradigm is one of the key features of the proof assistant Coq (Paulin-Mohring 1989, Letouzey 2003). From proofs one can extract programs in a real

functional language. But also one can program a lot of functions as programs *within the proof assistant*, because the system includes a (small) functional language itself, with abstract data types, etc.

2.2b *Checking the checker:* In a type theory based system that includes a functional programming language, like Coq, one can program and verify the type checker within the system itself. This has been done in the 'Coq in Coq' project (Barras 1999). What one verifies (inside Coq) is the following statement:

$$\Gamma \vdash M : A \Leftrightarrow \mathrm{TC}(\Gamma, M) = A,$$

where TC is the type checking algorithm that takes as input a context and a term and produces a type, if it exists, and otherwise produces 'fail'. As this statement implies the consistency of Coq as a logic, this cannot be proved inside Coq without extra assumption, so the statement is proved from the assumption that all terms are strongly normalizing (i.e. all functions definable in Coq are total; this is a meta-theoretic property one can prove about Coq).

The statement above is proved inside Coq, so the type theory is coded inside Coq. The program TC is also written inside Coq, but it can be extracted and be used as the type checking algorithm of a new version of Coq.

2.3 *LCF*

LCF stands for 'Logic for Computable Functions', the name Milner gave to a formal theory defined by Scott in 1969 (Scott 1993) to prove properties of recursively defined functions, using methods from denotational semantics. It is a predicate logic over the terms of typed λ-calculus. The first system was developed at Stanford in 1972 and later systems were developed at Edinburgh and Cambridge. The systems Isabelle (Isabelle, Wenzel *et al* 2008), HOL (HOL, Gordon & Melham 1993) and HOL-light (HOL light) are descendants from LCF, using the 'LCF approach'. The first system that Milner developed in Stanford was a goal-directed system that had special tactics to break down goals into smaller ones and a simplifier that would solve simple goals. To be able to safely add new proof commands and to not have to store proofs in memory (but only the fact that a result had been proven), Milner developed what is now know as the LCF approach.

2.3a *LCF approach:* The basic ideas of Milner's LCF approach (Gordon *et al* 1979, Gordon 2006) is to have an abstract data type of theorems `thm`, where the only constants of this data type are the axioms and the only functions to this data type are the inference rules. Milner has developed the language ML (Meta Language) specifically to program LCF in. In LCF, the only terms of type `thm` are derivable sequents $\Gamma \vdash \varphi$, because there is no other way to construct a term of type `thm` then via the inference rules. (For example, one of the operators of this type would be assume: `form -> thm` with `assume phi` representing the derivable sequent $\varphi \vdash \varphi$.) The LCF approach gives soundness by construction: if we want to prove a goal φ and we only apply operators of the abstract data type, we can never go wrong.

Another advantage of LCF is that a user can write tactics by writing more sophisticated functions. This enhances the power without jeopardizing the system, because in the end everything can be translated to the small kernel, which is the abstract data type `thm`.

So LCF very much satisfies the 'small kernel' criterion: every proof can be written out completely in terms of a small kernel, which represents exactly the inference rules of the logic. One of the reasons Milner invented this mechanism was to suppress the rechecking of large

proof objects. So, no independently checkable proof objects are created in LCF. However, it is easily possible to generate proof objects 'on the side', making the system satisfy the De Bruijn criterion as well. The LCF approach has been very influential. Also a system like Coq has been implemented in this manner.

2.3b *Checking the checker* The system HOL light has been checked within the system itself (Harrison 2006). In HOL, there is no real point in checking the system with respect to the theory, because in the LCF approach one implements the theory almost directly. So, the statement proven inside HOL light is

$$\mathcal{M} \models \varphi \Leftrightarrow \mathrm{Prov}_{\mathrm{HOL}}(\varphi),$$

where $\mathrm{Prov}_{\mathrm{HOL}}$ denotes provability in the HOL light system. This notion of provability does not just capture the logical derivation rules of HOL, but their actual implementation in OCaml. So the formalized statement also deals with the operational semantics of the HOL light system that is actually implemented. The statement above implies the consistency of HOL light, so it can't be proven inside it without extra assumptions. The model $\mathcal{M}$ is constructed using assumptions about the existence of large cardinals.

2.4 *Mizar*

The Mizar system (Mizar) has been developed by Trybulec since 1973 at the university of Białystok. It is the longest continuously running proof assistant project. As a matter of fact there are two Mizars: the Mizar language, which aims at being a formal language close to ordinary mathematical language, and the Mizar system, which is the computer program that checks text files written in Mizar for mathematical correctness.

The underlying system is Tarski–Grothendieck set theory with classical logic and the proofs are given in Jaskowski style (Jaśkowski 1967), which is now better know as Fitch-style (Fitch 1952) or flag-style natural deduction. In the beginning (see Matuszewski & Rudnicki 2005) the emphasis was very much on editing and recording mathematical articles and not so much on proof checking. Mizar has always put a strong emphasis on creating a *library of formalized mathematics*, which should be a coherent collection of formalized mathematical results. This is achieved via the Mizar Mathematical Library, which consists of Mizar articles, pieces of mathematics formalized in Mizar, and is by far the largest repository of formalized mathematics. The coherence of MML is achieved by not making Mizar a distributed system: it does not support large developments (because it gets slow) and thus requires users to submit articles to the 'library committee', that includes them in the Mizar library.

2.4a *Declarative language and batch proof checking:* Mizar has mainly been developed and used in Poland, with one major Mizar group in Japan. Only in the last ten years, Mizar has become more well-known outside Poland and ideas and techniques of Mizar have been taken over by other proof assistants.

One such idea is the strong emphasis on a declarative input language, which aims at being close to the texts that ordinary mathematicians write and understand. This also means that the language has all kinds of special features and constructs that support built in automation. Therefore, the system does not have a small kernel and does not satisfy the De Bruijn criterion. But the language feels quite natural for a mathematician and proofs can be read off-line (without processing them in Mizar) by a human.

Another positive aspect of having a well-developed declarative proof language is that one can do *batch proof checking*. This means that Mizar can check a whole file at once and indicate

where proof-steps have been made that the solver did not accept. This works because, even if the solver does not accept a proof-step, the declarative proof still records 'where we are and what the system is supposed to know', according to the user. So the system can reason onwards from that. For example, in the example Coq text of Section 1.3, if the system can't prove the first equation, it may still be able to prove the third and the fourth one, thus being able to complete the proof, modulo that equation that needs solving. This idea has been made precise in (Wiedijk 2004), by defining the notion of 'Formal Proof Sketch', which is a declarative high level proof which is syntactically correct but possibly has 'holes'.

The Mizar language has inspired other proof assistant to also develop a declarative proof language. Most notably, Wenzel has developed the Isar language (Wenzel 2007) as a declarative input language for the Isabelle theorem prover, which is now used by very many Isabelle users. Also, Harrison has developed a Mizar mode for HOL-light (Harrison 1996) and Corbineau has developed a declarative proof language for Coq (Corbineau 2007) (of which the proof in Section 1.3 is an example).

2.5 *Nqthm*

Nqthm (Boyer & Moore 1998), also known as the 'Boyer–Moore theorem prover' is an automated reasoning system implemented in Lisp. The first version originates from 1973 and the system was very much inspired by earlier work of McCarthy (McCarthy 1962). The logic of Nqthm is quantifier-free first order logic with equality, basically primitive recursive arithmetic, which makes the automation very powerful, but the expressivity limited. Interesting ideas from Nqthm that have found their way to the wider community of proof assistant developers are its focus on automation and its way to combine automation with interactiveness by letting the user add lemmas.

The automation has as disadvantage that one often has to rewrite a formula before being able to prove it (e.g. a formula $\forall x.\exists y.\varphi(x, y)$ can only be proven if we can write it as $\varphi(x, f(x))$). The advantages are obvious and the system has had a lot of good applications in verification studies within computer science.

The user can work interactively with the system by adding intermediate lemmas that act as a kind of 'stepping stones' towards the theorem one wants to prove: the system first tries to prove the lemmas automatically and then tries to prove the theorem using the new lemmas. This idea has also been used in the logical framework Twelf. The system Nqthm has evolved into the system ACL2 (ACL2, Kaufmann *et al* 2000), which is very much based on the same ideas.

2.5a *PVS:* The PVS (Prototype Verification System) (PVS) has been developed at SRI since 1992. It aims at combining the advantages of fully automated theorem provers, that have strong decision procedures, but limited expressivity, with the advantages of interactive theorem provers, that have a much more expressive language and logic. PVS has a classical, typed higher-order logic. The interactive theorem prover supports several quite powerful decision procedures and a symbolic model checker. The combination of a very expressive specification language and powerful techniques for proof automation makes PVS quite easy to use. It has been applied in several industrial verification studies (Muñoz & Dowek 2005).

The logic of PVS is not independently described and the system does not have a small proof kernel or independently checkable proof objects. Every now and then a bug (inconsistency) is found in the system, which is then repaired by the implementors. Due to its automation and flexible specification language, PVS is relatively easy to learn and use. A small drawback of the automation is that it may sometimes lead the proof in an undesired direction.

2.6 *Evidence algorithm*

In the end of the 1960s, the Russian mathematician Glushkov started investigating automated theorem proving, with the aim of formalizing mathematical texts. This was later called the Evidence Algorithm, EA, and the first citation is from the Russian journal 'Kibernetika' in 1970. The EA project seems to have run since the beginning of the 1970s, but it hasn't become known outside of Russia and Ukraine and publications about the project are all in Russian. The project has evolved into a system SAD (System for Automated Deduction) (Lyaletski *et al* 2004), which checks mathematical texts written in the language ForTheL (FORmal THEory Language) (Verchinine *et al* 2008). The latter is a declarative formal language for writing definitions, lemmas and proofs, very much in the spirit of Mizar, but developed independently from it. An interesting aspect of the ForTheL language is that it can be checked (through a web interface) with various (automated) theorem provers. The SAD seems quite powerful, as the ForTheL texts it can check are quite high level mathematical proofs.

3. Use of proof assistants

3.1 *Formalizing mathematics and mathematical libraries*

Various large formalizations of mathematical proofs have been done in various proof assistants. In Mizar there is a proof of the Jordan Curve theorem (Korniłowicz 2005), which has also been proved in HOL light (Hales 2007). In Isabelle, the prime number theorem has been proved (Avigad *et al* 2007). Another formalization of a large proof and a well-known theorem is Gonthier's formalization of the 4-colour theorem in Coq (Gonthier 2004). The proof of this theorem consists of the reduction of the problem to 633 cases, that then have to be verified using computer algorithms. Mathematicians sometimes feel uncomfortable with this proof, because of the use of computer programs in it. Gonthier has formalized all of this in Coq: the reduction to the 633 cases, the definition of the algorithms and their correctness proofs, and their execution on the 633 cases.

Doing large mathematical proofs on a computer, also the issue comes up as to *what exactly* one is proving. Are the definitions right and are the statements of the theorems the ones we want? Gonthier emphasizes that the precise statement of the 4-colour theorem, including all definitions to exactly understand its content, fits on one A4 page. So if one trusts the proof assistant, all one needs to do is to verify that this one page corresponds to the statement of the theorem.

So, some very impressive formalizations of mathematical theorems have been done ((Wiedijk 2008) gives an overview of 100 well-known theorems in mathematics and in which system they have been formalized) and the volume of formalized mathematics is constantly increasing, but we cannot speak of a *coherent* formalized library. The Mizar Mathematical Library (Mizar) is the only one that comes a bit close. But also in MML it is hard to find proven results and for an outsider it is difficult to obtain a high level view of what has been formalized. The Mizar community has set-up the *Journal of Formalized Mathematics* to ensure that results that are formalized in Mizar also appear in a more standard mathematically readable format.

In general the documentation problem with these libraries remains: how do we tell the user what he can find where? If we just print the whole repository, without proofs, in a nicely readable way, with mathematical notations, then we have a nice overview of the lemmas, but there are very many, with hardly any structure. In case of our own CoRN repository

(Cruz-Filipe *et al* 2004), which aims at being a coherent library of constructive algebra and analysis in Coq, we have 962 definitions and 3554 lemmas on 394 pages. There are searching tools, like Whelp (Asperti *et al* 2006b) that assist searching for lemmas of a certain structure. But we also want to search on a higher level and get a high level overview of the material, including motivations, intuitions and so forth.

Adding more *meta-data* to our formal definitions, theorems and proofs, e.g. adding the information that 'this is the Intermediate Value Theorem', helps to search on a high level. But it does not yet provide us with a logical 'overview' of the material. This requires a kind of *literate proving* approach, comparable with Knuth's literate programming (Knuth 1992), where the documentation and the formal proofs are developed in one systems and in one file. To explain what the problem is with documentation, let me outline how the proof development in our own FTA-project went (in which we have formalized a constructive proof of the Fundamental Theorem of Algebra, and together with that, built up a repository of constructive algebra and analysis). One starts from a mathematical proof of the theorem, with all details filled in as much as possible. The plan is to formalize that in two steps: first the definitions and the statements of the lemmas and then fill in all the proofs. The mathematical proof—the first document (e.g. in LaTeX) then acts as the documentation of the formalization.

1	Mathematical proof	Document with many details (usually a LaTeX file)
2	Theory development	Computer file in a Proof Assistant (definitions and statements of lemmas)
3	Proof development	Computer file in a Proof Assistant (proofs filled in)

The real situation is that we have to move between these phases all the time: when we start filling in the the proof, it turns out that the lemma should be formulated a bit differently or that the definitions should be changed a bit, because they are inconvenient. So, while we are working at phase 3, we also have to adapt the files of phases 1 and 2. But that's felt as quite annoying, because we want to get on with the formal proof, so in practice document 1 is not adapted and very soon it is out of sync with the other documents. The only way to let this work is if we can work on all three documents at the same time in one file.

3.2 *Program and system verification*

Many studies in program correctness and system design verification have been done in proof assistants. Some of the systems are really used for industrial verification projects: NASA (Muñoz & Dowek 2005) uses PVS to verify software for airline control and Intel uses HOL light to verify the design of new chips (Harrison 1996). There are many other industrial uses of proof assistants that we will not list here, notably of PVS, ACL2, HOL, Isabelle and Coq.

An interesting initiative that we want to mention here, because it reaches beyond the boundaries of one proof assistant, is the 'PoplMark Challenge' (POPLmark). Popl is the conference 'Principles of Programming Languages'. The authors of (Aydemir *et al* 2005) challenge everyone to accompany a paper concerning programming languages with an appendix con-

taining computer verified proofs of its meta-theoretic properties. A concrete challenge that is set is to formalize the meta theory of the system $F_<$ completely. The challenge is not just for the designers of programming languages to provide the proofs, but also on the developers of proof assistants to make this feasible. Thus the slogan of the initiative: 'Mechanized meta-theory for the masses'. The challenge has already produced quite some discussion and research. That one can verify some serious software with a proof assistant has been shown in (Blazy *et al* 2006), where a C compiler has been proved correct in Coq.

In industrial applications, one may often be more inclined to go for speed and automation than for total correctness, thus often preferring automated tools or tools with a lot of automation over more expressive but less powerful systems. The idea of independently checkable proof objects has nevertheless also found its way into computer science in the form of 'Proof Carrying code' (Necula & Lee 1996). The idea is to accompany a piece of code with a proof object that proves a safety property of the program. This proof object may also include a formalization of the architecture it will be executed on, so the receiver can check the proof object and trust the code.

4. Future

4.1 *QED Manifesto*

Are we then now in a situation that we can formalize a large part of mathematics—-definitions, proofs, computation ... – in a proof assistant? Formalization of all of mathematics as a goal has been described in the QED Manifesto (QED), that was presented at the CADE conference in 1994.

- QED is the very tentative title of a project to build a computer system that effectively represents all important mathematical knowledge and techniques. The QED system will conform to the highest standards of mathematical rigor, including the use of strict formality in the internal representation of knowledge and the use of mechanical methods to check proofs of the correctness of all entries in the system.

The manifesto describes the ambitious goals of the project and discusses questions and doubts and the answers to them. In 1994 and 1995 there have been two workshops on QED, but after that no more. That doesn't mean that nothing has happened after that. In various projects people have been and are working on the formalization of mathematics with proof assistants, where the ideas of the QED manifesto often play an important (implicit) role. The QED manifesto gives nine points as a motivation, three of which are most relevant in my view.

(i) The field of mathematics has become so large that it has become impossible to have an overview of all relevant mathematics. A formalized library should enable the search for relevant results.
(ii) When designing new high-tech systems, like software for an automatic pilot, one uses complicated mathematical models. A QED system can be an important component for the modelling, development and verification of such systems
(iii) For education, a QED system can offer course material that students can practice with on an individual basis, for example interactive proving or programming exercises.

These goals are ambitious and since 1994 there has been some progress, especially at the second point, but not very much.

Is the QED manifesto too ambitious? Yes, at this moment in time it is. Formalizing all of mathematics probably isn't realistic anyway, but we also have to acknowledge that the present day proof assistants are just not good enough to easily formalize a piece of mathematics. In this respect it is instructive to read what the authors of the QED manifesto thought that needed to be done. First of all, a group of enthusiastic scientists should get together to determine which parts of mathematics needed to be formalized, in what order and with which cross connections. The authors assume that this phase may take a few years and it may even involve a rearrangement of the mathematics itself, before the actual formalization work can start. Other points in this 'to-do-list' are of a similar top-down organizational nature.

In my view, this is a wrong approach to the problems. Development like Wikipedia show that a more 'bottom up' distributed approach may work better, using a simple lightweight basic technology. One could claim that for the formalization of mathematics, such an approach could never work, but for Wikipedia the same doubts were raised at first: Wikipedia is typically something that works in practice but not in theory.

4.2 *MathWiki*

To create momentum in the use of proof assistants, it is essential to have a large repository of formalized mathematics. But that again costs a lot of time and man power. A way to get that is by a cooperative joint distributed effort, to which many people can contribute in a simple and low level way: a Wikipedia for formalized mathematics. To achieve this, researchers in our research group have developed a web interface for proof assistants (Kaliszyk 2007). This way everyone with an internet connection can simply—without installing a system and letting the server take care of the file management—contribute to a joint repository of formalized mathematics. We are now in the process of extending this into a *MathWiki* system, which should be a Wikipedia-like environment for working with various proof assistants (doing formalizations through a web interface) and for creating high level pages that describe content of the repositories, with pointers to actual proof assistant files. Preliminary work is reported in (Corbineau & Kaliszyk 2007).

A possible high level page is depicted in figure 2: the idea is that we have Wikipedia-like technology to describe a high level mathematical concept and that inside such a page we can render snippets of code from proof assistant repositories, in the example from Coq, Mizar and Isabelle.

The success of Wikipedia rests on the fact that there are many contributors, there is a simple technology for contributing and there are rules and committees for ensuring the stability of the content. For formalized mathematics, there are fewer contributors and they are spread over different formal systems, using their own formal language. There is not much interaction between these systems, so it may look like a better idea to create a MathWiki for each proof assistant separately. We believe however, that a joint MathWiki system is exactly the platform needed to let the different proof assistant communities talk to each other and stimulate them to develop a generic format for high level pages.

For the coherence and stability of the MathWiki pages we distinguish between two types of pages: there will be formal pages, consisting of annotated formal proof code of the proof assistants, and there will be informal pages, consisting of high level descriptions of the mathematics with rendered pieces of formal code. For the informal pages, the procedures know from Wikipedia can be followed. For the formal pages, the consistency can be checked by the proof assistant itself—although it is an interesting challenge to maintain the consistency of a large online formal repository that is continuously changed and extended through the web.

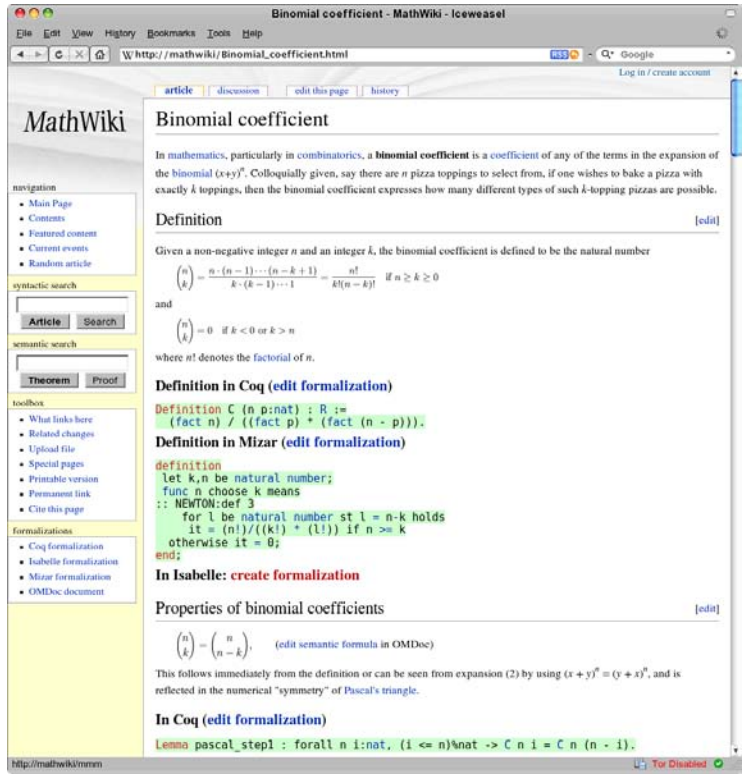

Figure 2. A MathWiki mock up page.

The stability will also have to be ensured by a committee that ensures that the library evolves in a natural and meaningful way.

4.3 *Flyspeck*

Mathematical proofs are becoming more and more complex. that is unavoidable, because there are always short theorems with very long proofs. One can actually prove that: there is no upper bound to the fraction

$$\frac{\text{length of the shortest proof of } A}{\text{length of } A}.$$

It would of course be possible that short theorems with very long proofs are all very uninteresting, but there is no reason to assume that that is so, and then again: what makes a theorem interesting?

Recently, proofs of mathematical theorems have been given that are indeed so large that they cannot simply be verified by a human. The most well-known example is Hales' proof of the Kepler conjecture (Hales 2005). The conjecture states that the face-centered cubic packing is the optimal way of packing congruent spheres in three dimensional space.

The proof, given by Hales in 1988, is 300 pages long and was submitted to the Annals of Mathematics for publication. After five years of peer reviewing, the conclusion was that the proof was 99 percent correct. What was the problem?

Hales reduces the proof to a collection of 1039 complicated inequalities. To verify these inequalities, Hales wrote computer programs that verified the inequalities using interval arithmetic. The referees had a problem with this: verifying the inequalities themselves by hand would be impossible (one week per inequality is still 25 man years of work). The only other possibility would be to verify the computer programs, but that option has never been considered.

Hales drew the conclusion that the proof needed to be formalized on a computer. To do that he has set-up the *Flyspeck*-project (Flyspeck), with the aim to fully formalize his proof. With the proof assistants HOL Light (HOL-light), Coq (COQ) and Isabelle (Isabelle) researchers are working on fully formalizing the proof.

4.4 *Using computer programs to verify mathematics*

In his original proof, Hales uses computer programs to verify inequalities. What's the problem with that in terms of constituting a (mathematical) proof? And doesn't the same problem apply to proofs that are formalized in a proof assistant—which is also a computer program?

(i) The first problem is that a program to verify inequalities may do something else. So it may yield 'true' as the outcome of a check while the inequality doesn't hold at all. To prevent this, the program code has to be verified: one needs to prove that the program does what it claims to do.
(ii) If the program code is (proven) correct, it may still be the case that the compiler is wrong, or that the operating system is faulty, or that the hardware is buggy, causing the output of the verification to be wrong.

The first problem is that of program correctness: how to verify/prove that a program does what it is claimed to do? (The referees have not considered the option of verifying the program code at all.) There is a whole world of techniques, ranging from less to more formal, to ensure program correctness. The formal one, using *formal methods* can be implemented in a proof assistant, allowing computer checked proofs of program correctness. Program extraction, as mentioned in Section 2.2 is another such method.

The second problem is that the (correct) program code may be executed in the wrong way. This also encompasses 'accidental' mistakes. As far as the compiler is concerned, this is a program that can also be formally proven correct, and one can also formally verify other parts of the system, like the hardware design. The problem of accidental mistakes, or mistakes not in the design but in the implementation, can be circumvented by compiling the code with various different compilers and executing it on different platforms. When the output is the same every time, it is extremely unlikely that they would all make the same mistake, and that the answer would be faulty. In this way, one introduces a kind of 'peer review for program-execution': with peer review of articles it is possible that someone overlooks a mistake, but it is unlikely that a lot of people overlook the same mistake. Another analogy may be found in physical experiments: experimental evidence of a physical law is provided by having different researchers repeating the experiment in different labs.

How do these problems apply to proofs formalized in a proof assistant? In the formalization, one proves the correctness of the algorithm that verifies inequalities, so that improves our confidence in them. But what if the proof assistant is faulty as a program (problem 1) or the compiler, operating system or hardware are buggy (problem 2). The first problem can again be tackled by program correctness methods, the techniques described in Section 1.2: checking the checker or independently checkable proof objects. The second problem can be

addressed also by proving the compiler correct, or by executing the code on many platforms. In the end we should realize that, just as with the usual system of peer reviewing, there is always a scenario possible that a (machine) checked proof may still be faulty; we can only work towards obtaining the ultimate degree of confidence.

4.5 *Computer aided system verification*

We believe that proof assistants will be used more and more in computer aided verification of software and hardware. They are already being used in critical pieces of code and in critical hardware designs and we see that the use of tool support for the design, simulation and verification of systems is paramount. When the proof assistants get easier to use and contain more basic knowledge, this will even be enhanced.

A particularly interesting application of proof assistants in computer science is the modelling and verification of hybrid systems. This is also a field where we observe a clear need for precise mathematical modelling. A hybrid system contains both continuous components, like a clock, a thermometer or a speed meter, and discrete components, like an on/off switch or a gas handle with 3 positions. The software in such a system should operate the gas handle, based on all kinds of input parameters that are obtained from sensors, making sure that the temperature or speed remains within certain bounds. An interesting aspect is that, to verify the correctness of this software, we also have to model the environment in which it operates, usually given via differential equations. A hybrid system has both continuous behaviour (governed by differential equations) and discrete behaviour (moving from one position of the discrete controller to another). The state space is uncountable, so if we want to use automated tools, like model checkers, we first have to make a discrete abstraction of this state space (Alur *et al* 2006). Proof assistants will be useful in modelling the continuous environment and in proving properties about the discrete abstraction, making sure that the final correctness claim of our model checker really proves the correctness of the original system.

4.6 *Mathematical knowledge management*

Computers contain a lot of mathematical knowledge, which is more and more stored in a structured semantically meaningful format. Notably this knowledge resides in electronic documents, computer algebra system files and in proof assistant repositories. These formats are very different, but e.g. the computer algebra and the document editing communities are converging on exchangeable formats for mathematical objects, like OpenMath (Openmath) and MathML.

Proof assistants can also deal with OpenMath or MathML objects, but in the case of these systems one also wants to have a format for theorems, definitions and proofs, which is provided by OMDoc (OMDoc, Kohlhase 2000). The OMDoc format is not yet very much used by the proof assistant community, and may be a different format is needed, but it is clear that mechanisms for the exchange of formal content between proof assistants will be needed and will be developed. For example, in the Flyspeck project, different systems are used for different parts of the proof; in the end these will have to be glued together somehow.

At this moment we see a lot of activity in the field of *Mathematical knowledge management* MKM, which aims at developing tools and techniques for the management of mathematical knowledge and data. As a matter of fact, the research in proof assistants is just a part of the MKM field, and so is the study of mathematical exchange formats. Other interesting research topics in this field that will have impetus on the use and development of proof assistants are the study of *interactive mathematical documents* and *mathematical search*.

Ideally, one would like to extract a mathematical document from a formalization, but things are not that simple (see (Asperti *et al* 2006a) for an example study). The outcome is a quite direct 'pretty printed' translation of the computer code, containing too many details. It is possible to suppress some of the details, so we only see the most important ones, but it is hard to say in general what's important. Combining the wish for 'interactive mathematical documents' with the idea of 'literate proving', researchers have developed environments where one can edit a standard mathematical document and at the same time do a formalization 'underneath'. A good vehicle for that is the TexMacs system (TeXmacs) that allows the editing of a mathematical document (in a wysiwyg LaTeX-like style) and at the same time interact with another computer program. In the system tmegg (Geuvers & Mamane 2006), this is used to write a mathematical document with a Coq-formalization underneath: special commands open an interaction with the Coq system, whose output is rendered within the TexMacs document. A similar approach, combining TexMacs with the Omega proof tool is developed in (Wagner *et al* 2008).

Mathematical search is different from ordinary string based search because one also wants to search for mathematical structure and also modulo 'mathematical equivalence'. An example is searching for a lemma that can prove a statement like $x^2 \leq 2^x$, which may be proved from a lemma like $\forall y > 1(x^y \leq y^x)$, but also from a lemma like $\forall n \in \mathbb{N}(x^{n+1} < (n+1)^x)$. Such a search requires incorporating some of the semantics. The Whelp system (Asperti *et al* 2006b) is aiming in this direction.

5. Conclusion

In the present paper, we have given an overview of the issues related to the mechanical verification of mathematical proofs using proof assistants. We have given some history, focussing on the underlying techniques and ideas to ensure the correctness of proofs that have been machine checked. Also we argued that formalizing proofs is not just a scientifically challenging and interesting activity, but it also has useful applications that we believe will be further extended in the future. At this moment, the community of people formalizing mathematics is still relatively small and distributed over various communities that are each connected to their own proof assistant system.

So, what needs to be done to improve proof assistant technology and make it more widely spread? According to me, we have to work on the following.

- Develop proof assistants further, working towards a simple basic technology that can be easily used. At this moment there are many different proof assistants. Basically, that is very good: competition leads to new ideas and improvement, by taking over and improving upon the ideas of others. Important points that need further development are: proof automatization and interfaces.
- Develop a joint platform for the exchange of ideas and mathematical content between proof assistant system and to the 'outside world' of interested scientists and users of mathematics.
- Big formalizations give feedback on the basis of those. The systems only get better if they are really used and their shortcoming are made explicit.
- Build up a basic library, where special care has to be taken of coherence, usefulness and documentation. To which extent is the library useful for a newbie who wants to formalize a theorem using the basic mathematical results of the library?

- Applications using the library. Can we use the proof assistant and its library when modelling and developing and verifying a new product, like a network protocol or the software operating a robot arm?

On the basis of this work, a QED like system may arise in due course of time. The largest risk is that one expects miracles to happen quickly. In this context it is interesting to make an estimate of the amount of work that is involved in creating a formalized library of mathematics. A well-motivated computation of Wiedijk (Wiedijk 2005) estimates that it requires about 140 men a year to formalize the standard bachelor curriculum of mathematics. That is a lot and it exceeds the research budgets of one university by far. That doesn't mean it is impossible. Developments like Linux and Wikipedia show that using a distributed, well-organized set-up one may achieve a lot.

I would like to thank F. Wiedijk, C. Kaliszyk, P. Corbineau, J. McKinna and the anonymous referees for the various comments and discussions on the paper.

References

ACL2: A Computational Logic/Applicative Common Lisp. http://www.cs.utexas.edu/ moore/acl2/

Agda: An interactive proof editor. http://agda.sf.net

Boyer R S, Moore J S 1998 *A Computational Logic Handbook*. Second Edition, Academic Press

Aigner, Ziegler M G 2004 *Proofs from THE BOOK*, 3rd ed., Springer

Alur R, Dang T, Ivancic F 2006 Predicate abstraction for reachability analysis of hybrid systems. In: *ACM Transactions on Embedded Computing Systems (TECS)* 5(1): 152–199

Asperti A, Geuvers H, Loeb I, Mamane L, Sacerdoti Coen C 2006 An Interactive Algebra Course with Formalised Proofs and Definitions. In: *Proceedings of the Fourth Conference Mathematical Knowledge Management, MKM 2005* LNAI 3863, Springer 315–329

Asperti A, Guidi F, Sacerdoti Coen C, Tassi E, Zacchiroli S 2006 A content based mathematical search engine: Whelp. In: *Types for Proofs and Programs International Workshop 2004*, J-C Filliatre, C Paulin-Mohring, B Werner, (eds), LNCS 3839, Springer 17–32

Avigad J, Donnelly K, Gray D, Raff P 2007 A formally verified proof of the prime number theorem. *ACM Transactions on Computational Logic* 9(1:2)

Aydemir B, Bohannon A, Fairbairn M, Foster J, Pierce B, Sewell P, Vytiniotis D, Washburn G, Weirich S, Zdancewic S 2005 Mechanized Metatheory for the Masses: The PoplMark Challenge. In *Theorem Proving in Higher Order Logics*, LNCS 3603, Springer 50–65. Also on (POPLmark)

Barendregt H 2003 Towards an Interactive Mathematical Proof Language. In: *Thirty Five Years of Automath*, Ed. F Kamareddine, Kluwer 25–36

Barendregt H, Geuvers H 2001 Proof Assistants using Dependent Type Systems. In: A Robinson, A Voronkov, (eds) *Handbook of Automated Reasoning* (Vol. 2), Elsevier 1149–1238 (chapter 18)

Barras B 1999 Auto-validation d'un vérificateur de preuves avec familles inductives. Ph.D. thesis, Université Paris 7

Blazy S, Dargaye Z, Leroy X 2006 Formal verification of a C compiler front-end. In: *FM 2006: Int. Symp. on Formal Methods*, LNCS 4085, Springer 460–475

De Bruijn N 1983 Automath, a language for mathematics, Department of Mathematics, Eindhoven University of Technology, TH-report 68-WSK-05, 1968. Reprinted in revised form, with two pages commentary. In: *Automation and Reasoning, vol. 2, Classical papers on computational logic 1967–1970*, Springer Verlag 159–200

Constable R L, Allen S F, Bromley H M, Cleaveland W R, Cremer J F, Harper R W, Howe D J, Knoblock T B, Mendler N P, Panangaden P, Sasaki J T, Smith S F 1986 *Implementing Mathematics with the Nuprl Development System*. (NJ: Prentice-Hall)

Corbineau P, Kaliszyk C 2007 Cooperative Repositories for Formal Proofs. In *Towards Mechanized Mathematical Assistants*, M Kauers *et al* (eds), LNCS 4573, Springer 221–234

Corbineau P 2007 A declarative proof language for the Coq proof assistant. In *Types for Proofs and Programs*, LNCS 4941

The Coq proof assistant, http://coq.inria.fr/

Cruz-Filipe L, Geuvers H, Wiedijk F 2004 C-CoRN, the Constructive Coq Repository at Nijmegen. In: A Asperti, G Bancerek, A Trybulec, (eds) *Mathematical Knowledge Management, Proceedings of MKM 2004*, LNCS 3119, Springer 88–103

Fitch F B 1952 *Symbolic Logic, An Introduction*, The Ronald Press Company

The Flyspeck project, http://www.math.pitt.edu/ thales/flyspeck/

Geuvers H, Mamane L 2006 A Document-Oriented Coq Plugin for TeXmacs. In: Libbrecht P, editor, *Proceedings of the MathUI workshop at the MKM 2006 conference*, Wokingham, UK, http://www.activemath.org/ paul/MathUI06/

Gonthier G 2004 A computer-checked proof of the Four Colour Theorem, http://research.microsoft.com/ gonthier/4colproof.pdf

Gordon M J C 2006 From LCF to HOL: a short history. *Proof, language, and interaction: essays in honour of Robin Milner* (Cambridge, MA, USA: MIT Press) 169–185, 2000

Gordon M J C, Melham T F 1993 *Introduction to HOL: A theorem proving environment for higher order logic*. Cambridge University Press

Gordon M J C, Milner R, Wadsworth C P 1979 *Edinburgh LCF: A Mechanised Logic of Computation*, volume 78 of Lecture Notes in Computer Science. Springer-Verlag

Hales Th 2007 A proof of the Kepler conjecture. In: *Annals of Mathematics* 162: 1065–1185

Hales Th 2007 Jordan's Proof of the Jordan Curve Theorem. In *From Insight to Proof Festschrift in Honour of Andrzej Trybulec* (eds) R Matuszewski, A Zalewska, Studies in Logic, Grammar and Rhetoric 10(23)

Halmos P 1985 *I want to be a Mathematician: An Automathography*, Springer 410

Harper R, Honsell F, Plotkin G 1993 A framework for defining logics. *Journal of the ACM* 40: 194–204

Harrison J 1996 A Mizar Mode for HOL, *Proceedings, of Theorem Proving in Higher Order Logics, TPHOLs'96*, Turku, Finland, Lecture Notes in Computer Science 1125, Springer 203–220

Harrison J 2000 Formal verification of IA-64 division algorithms, in: M Aagaard, J Harrison, (eds), *Theorem Proving in Higher Order Logics, TPHOLs 2000* LNCS 1869, Springer 234–251

Harrison J 2006 Towards self-verification of HOL Light, U Furbach, N Shankar (eds), *Proceedings of the Third International Joint Conference IJCAR 2006*, Seattle, WA, LNCS 4130, 177–191

The HOL Light theorem prover http://www.cl.cam.ac.uk/ jrh13/hol-light/

HOL4, http://hol.sourceforge.net/

Howard W A 1980 The formulae-as-types notion of construction, in J Seldin, R Hindley (eds), *to H.B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, Boston, MA, Academic Press, 479–490, (original manuscript from 1969)

Isabelle, http://isabelle.in.tum.de/

Jaśkowski S 1967 On the rules of suppositional formal logic. In Storrs McCall, editor, *Polish Logic 1920–1939* 232–258. (Oxford: Clarendon Press)

Kaliszyk C 2007 Web Interfaces for Proof Assistants, *Electr. Notes Theor. Comput. Sci.* 174(2): 49–61

Kaufmann M, Manolios P, Moore J S 2000 *Computer-Aided Reasoning: An Approach* Kluwer Academic Publishers

Knuth D 1992 *Literate Programming*, Center for the Study of Language and Information, 1992, xvi+368pp. CSLI Lecture Notes, no. 27, Stanford, California

Kohlhase M 2000 OMDoc: Towards an Internet Standard for the Administration, Distribution and Teaching of mathematical Knowledge, in *Proceedings of Artificial Intelligence and Symbolic Computation*, Springer LNAI

Korniłowicz A 2005 Jordan curve theorem. *Formalized Mathematics* 13(4): 481–491
Letouzey P 2003 A new extraction for Coq. *Proceedings of the TYPES Conference 2002*, LNCS 2626, Springer-Verlag 200–219
Lyaletski A, Paskevich A, Verchinin K 2004 Theorem Proving and Proof Verification in the System SAD. In: *Mathematical Knowledge Management, Third International Conference*, Białowieza, Poland, Proceedings, LNCS 3119, 236–250
Luo Z, Pollack R 1992 LEGO Proof Development System: User's Manual, LFCS Technical Report ECS-LFCS-92-211
Magnusson L, Nordström B 1994 The ALF proof editor and its proof engine, In *Types for Proofs and Programs*, H Barendregt, T Nipkow (eds), LNCS, 806: 213–237
Martin-Löf P 1984 *Intuitionistic type theory*, Napoli, Bibliopolis
Matuszewski R, Rudnicki P 2005 Mizar: the first 30 years, *Mechanized mathematics and its applications* 4(1): 3–24
McCarthy J 1962 Computer programs for checking mathematical proofs, In *Recursive Function Theory*, Proceedings of Symposia in Pure Mathematics, volume 5. Americal Mathematical Society
Mizar Home Page, http://www.mizar.org/
Muñoz C, Dowek G 2005 Hybrid Verification of an Air Traffic Operational Concept, in: *Proceedings of IEEE ISoLA Workshop on Leveraging Applications of Formal Methods, Verification, and Validation*, Columbia, Maryland
Necula G, Lee P 1996 Proof-Carrying Code, Technical Report CMU-CS-96-165, November 1996. (62 pages) http://www.cs.berkeley.edu/ necula/Papers/tr96-165.ps.gz
Nederpelt R P, Geuvers H, de Vrijer R C 1994 (editors), *Selected Papers on Automath*, Volume 133 in Studies in Logic and the Foundations of Mathematics, North-Holland, Amsterdam, pp 1024
Nordström B, Petersson K, Smith J 1990 *Programming in Martin-Löf's Type Theory* Oxford University Press
The OpenMath Society, http://www.openmath.org/
OMDoc, A Standard for Open Mathematical Documents, http://www.mathweb.org/omdoc/
Paulin-Mohring Ch 1989 Extracting $F\omega$'s programs from proofs in the Calulus of Constructions, *Sixteenth Annual ACM Symposium on Principles of Programming Languages*, ACM, Austin
Pollack R 1998 How to believe a machine-checked proof. In G Sambin and J Smith, (eds), *Twenty-Five Years of Constructive Type Theory*. OUP
The POPLmark Challenge.s http://fling-l.seas.upenn.edu/ plclub/cgi-bin/poplmark
PVS Specification and Verification System http://pvs.csl.sri.com/
The QED Manifesto, in: *Automated Deduction - CADE 12*, LNAI 814, Springer, 1994, 238–251
Scott D 1993 A type-theoretical alternative to ISWIM, CUCH, OWHY. *TCS*, 121: 411–440, 1993. (Annotated version of a 1969 manuscript)
Scott D 1970 Constructive validity. *Symposium on Automatic Demonstration* (Versailles, 1968), Lecture Notes in Mathematics, Vol. 125, Springer, Berlin 237–275
TeXmacs, http://www.math.u-psud.fr/ anh/TeXmacs/TeXmacs.html
The Twelf Project, http://twelf.plparty.org/
Verchinine K, Lyaletski A, Paskevich A, Anisimov A 2008 On correctness of mathematical texts from a logical and practical point of view. In: *Intelligent Computer Mathematics, AISC/Calculemus/MKM 2008*, Birmingham, UK, LNCS 5144, Springer 583–598
Wagner M, Dietrich D, Schulz E 2008 Authoring Verified Documents by Interactive Proof Construction and Verification in Text-Editors, In *Intelligent Computer Mathematics, AISC/Calculemus/MKM 2008*, Birmingham, UK, LNCS 5144, Springer
Wenzel M, Paulson L C, Nipkow T 2008 The Isabelle Framework. In: O Ait-Mohamed, editor, *Theorem Proving in Higher Order Logics*, TPHOLs 2008, invited paper, LNCS 5170 Springer
Wenzel M 2006 Isabelle/Isar — a generic framework for human-readable proof documents. In R Matuszewski and A Zalewska, (eds) *From Insight to Proof — Festschrift in Honour of Andrzej Trybulec, Studies in Logic, Grammar, and Rhetoric* 10(23), University of Białystok

Wiedijk F 2005 Estimating the Cost of a Standard Library for a Mathematical Proof Checker, http://www.cs.ru.nl/ freek/notes/index.html
Wiedijk F 2004 Formal Proof Sketches, In *Types for Proofs and Programs: Third International Workshop, TYPES 2003*. In: S Berardi, M Coppo, F Damiani, (eds), Springer, LNCS 3085 378–393
Wiedijk F 2006 (ed.) *The Seventeen Provers of the World*, Springer LNAI 3600
Wiedijk F 2008 Formalizing the 'top 100' of mathematical theorems http://www.cs.ru.nl/ freek/100/index.html