# The quest for correctness

Henk Barendregt

Computing Science Institute
Nijmegen University
and
Department of Software Technology
CWI, Amsterdam

April 1, 1996

*Die Genauigkeit, Kraft und Sicherheit dieses* [*mathematischen*] *Denkens, die nirgends im Leben ihresgleichen hat, erfüllte ihm fast mit Schwermut.*[1]

R. Musil [1952]: Der Mann ohne Eigenschaften.

**Abstract**

Modern society has a strong need for reliable Information Technology. To warrant correct designs for hardware and software systems, there is a thorough methodology (specification, design based on subspecifications and composition of components, and correctness proofs). Because of the difficulty of making specifications and proofs, the success of this method has been only partial, mainly in the area of hardware design.

Presently there is emerging a new technology: Computer Mathematics. It consists of the interactive building of definitions, statements and proofs, such that it can be checked automatically whether the definitions are well-formed and the proofs are correct. Hereby the human user provides the intelligence and the system does part of the craftsmanship. Some forms of computer mathematics is already of use for the design of hardware systems. After the technology will be mature, it may become a tool for the development of mathematics comparable to systems of computer algebra, but with a scope and strength that is essentially beyond. Moreover, it probably will also be useful for the design of reliable software.

[1]The precision, strength and certainty of this [mathematical] thinking, that is unequaled in life, almost pervaded him with melancholy. R. Musil: The man without qualities, Ch. 28.

## 1. The problem

Once upon a time, money was by convention a substitute for gold. Rather than exchanging goods with goods or exchanging goods with gold it was more practical to exchange goods with money. The gold that was available in exchange for the otherwise worthless coins or paper notes was stored in banks. The central bank of the most powerful country had its reserve of gold stored in a well protected place: Fort Knox. Only powerful criminals or James Bond (with the aid of pretty girls) could enter such a place. About 25 years ago the link between money and gold was abandoned: since then money stands for itself. As a consequence Fort Knox lost its importance. Moreover, the flow of cash has been drastically reduced by means of several forms of electronic payment, and therefore possession has become equivalent to the right sequence of bits in some central computer of a bank. Even in a relatively peaceful country as The Netherlands such computers are stored in bunkers surrounded by a defense moat against tanks. But the James Bond of today or tomorrow will not need force or women to enter these places. Computer hacking using external connections makes the new Fort Knox vulnerable. Perhaps external connections can be limited or avoided (but what then is the use of a central computer?). There is, however, a more serious problem: the software of the central computer may be ill-designed (by accident or on purpose). Well-informed sources in the banking world admit that this is indeed a very worrisome danger.[2]

More examples can be given of the importance of correct systems. Simple products such as electric razors carry their own microprocessors and software. The manufacturers do not want to have bugs in these systems for obvious reasons. A more striking example is the following. Well-informed sources of a large airline company have stated that 'just 24 hours of failure of our worldwide reservation system will cause bankruptcy', because of missed orders. Even more important is the correctness of systems on which the safety of people depend: for example for the control of factories or air-traffic, or for medical or military applications.

These examples all show that correct systems—both the hardware and the software—are essential for the survival of a company. It is fair to state, that in this digital era correct systems for information processing are more valuable than gold. The remarkable thing is that in spite of our technologically advanced hardware, we still are by and large in the stone-age of software. By this is meant that it is very hard to produce correct software and also that it is difficult to predict the time it takes to produce it.

[2]The daily flow of money through computer networks is about $10^{12}$ US\$. Whereas in an average conventional bank robbery a couple of 1.000 US\$ is stolen, in an average computer bank robbery one does catch 1.000.000 US\$. It is classified information what is the frequency of these successful crimes. These are possible because of flaws in the design of systems.

## 2. Design methodology

Part of the problem is that the needed systems are very complex. How can one correctly design a system with millions of transistors or consisting of millions of lines of program code?

### 2.1. The Chinese box

A proposed solution is the following. The method is well-known, but usually not explicitly described. Given the task to construct a system, one should first transform the informal requirements into a more precise (formal) specification. Then one designs the system accordingly. Finally, one proves that the design satisfies the given specification. For this the specification and the design need to be formulated in one language.

This is an ambitious program; it works only if the following are available with full precision.

1. An expressive but intuitively understandable specification language;
2. an expressive design language with good tools to combine modules together;
3. a tool to verify proofs of formal statements.

If these three conditions are satisfied, then one can use the following construction methodology. Given a specification $S$, one wants to make a design $d$, such that one can verify

$$S(d),$$

i.e. that the program satisfies the specification. This can be done by providing

- specifications $S_1, \ldots, S_k$;
- a constructor F;
- a proof of the following statement (where the $d_1, \cdots, d_n$ are arbitrary)

$$S_1(d_1) \ \& \ \ldots \ \& \ S_k(d_k) \ \Rightarrow \ S(F(d_1, \ldots, d_k)).$$

So the problem of constructing $d$ satisfying $S$ is transformed into the $k$ (easier) problems of constructing $d_1, \ldots, d_k$ satisfying $S_1, \ldots, S_k$, respectively. If these $d_1, \ldots, d_k$ are available, then the required $d$ can be found:

$$d = F(d_1, \ldots, d_k),$$

see fig. 1. In order to find now the $d_1, \ldots, d_k$, the method is applied again to $S_1, \ldots, S_k$; and so on. In this way we obtain a 'Chinese box' containing several boxes, each in turn containing other boxes, etcetera, see fig. 2. An example of

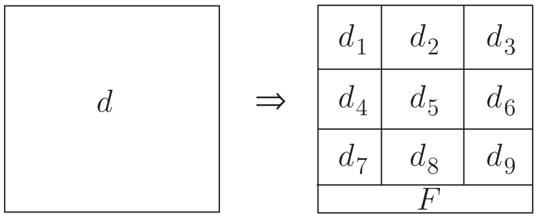


Fig. 1. Design methodology.

this is the following. Suppose we want a music reproducing device of high quality. This can be specified by stating that the difference between the actual sound and the reproduced sound is small (according to some appropriate measure). One way to obtain such a device is to buy a CD-player, a CD, an amplifier, boxes and wires, to have a current outlet, and then making the right connections. Each of the components can be specified. Now to obtain e.g. an amplifier, one needs a transformer, integrated circuits, etcetera, and make the right connections. In order to obtain a transformer, one needs a magnet, wire, etcetera. Etcetera.

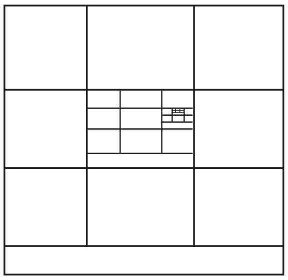

Fig. 2. The Chinese box.

The example not only shows that designing is a refined job, but also that making precise specifications is important as well. For example the CD should contain stored music according to a fixed coding scheme, otherwise the CD player cannot be constructed.

Of course the method has to be 'well-founded': the smallest boxes should not be empty. Indeed, for the design of software the smallest boxes will have to contain programs provided by the instruction set of the processor. These processors can be constructed following a similar design methodology for hardware out of smaller boxes. At the end it is *nature* that provides the final step: electrons in circuits that do their work.

### 2.2. Partial success

The design methodology given above is very general. It can be applied to many situations, in which complicated 'objects' have to be constructed. In the area of

IT it has provided a partial success for the construction of verified hardware. For more than a decade hardware design has been done according to the scheme specification/design/proof. The reason for this success is that hardware is relatively simple, being comparable to propositional logic. In this mathematical theory one can state and prove, for example, that '$B$ and $A$ or not $B$ implies $A$', in symbols

$$(B \ \& \ (A \vee \neg B)) \ \Rightarrow \ A. \tag{$*$}$$

Here the objects of interest are easy to specify as Boolean functions. Moreover, propositional logic is decidable. That is, proofs of correct statements can be generated automatically.

This description, however, is a simplification. For two reasons the situation with hardware design is more interesting. In the first place propositional logic, in spite of being decidable, is related to problems of high complexity (NP-complete and NP-hard). Methods like Binary Decision Diagrams have been developed to overcome this difficulty. It makes it possible to deal with propositions with large numbers of variables ($> 100$, and not just 2 as in ($*$)). Secondly, hardware is somewhat more complicated than propositional logic. This is caused on the one hand by real-time aspects of circuits, and on the other hand by multiple repetitions of patterns that can better be treated in a more powerful theory. As a result the theory and practise of hardware verification is a flourishing field, see Melham et al. [1994] for information and references.

### 2.3. A challenge

Also in the area of software design there have been efforts to prove correctness. But here matters are essentially more difficult. Software corresponds to predicate logic, with statements like *'there exists an $x$ such that for all $y$ one has $x \leq y$ implies that for all $y$ there exits an $x$ such that $x \leq y$'*; in symbols

$$\exists x \forall y . x \leq y \ \Rightarrow \ \forall y \exists x . x \leq y.$$

The difficulty is that the so called quantifiers 'for all' and 'there exists' range over potentially infinite sets, and therefore it comes as no surprise, that this theory is undecidable.

Another problem is to find the right granularity. The first introduction of correctness proofs in program design, by Hoare, was in connection with imperative programs in which continually a given state (the values of the variables) is modified. By formulating a suitable property of the state and proving that it is invariant under the modifications of the program, correctness proofs can be given. At the time this method was definite progress. But the method is of a granularity that is too fine to prove that large software systems are correct.

New programming paradigms such as functional programming, (see e.g. Plasmeijer and van Eekelen [1993] or Pountain [1994]), possibly including object oriented features, may become a good design tool to overcome this difficulty by using the methodology of the Chinese box. What is still lacking in the area of software design, is a good language for specification and the right tools for correctness proofs. This is one of the reasons, why we still are in the software crisis. In the next section we will discuss systems of so-called *computer mathematics* that may very well change this situation.

## 3. Computer mathematics

Because computer mathematics (CM) is a relatively new technology, presently in its second generation of prototypes, there is not yet a standard name for it. Alternative names are: 'systems for proof development', 'systems for theory development' and 'interactive theorem provers'. We have chosen the name 'computer mathematics' because of the analogy with systems for computer algebra (CA).

### 3.1. What is computer mathematics?

It is well-known that computers perform numerical computations. Since the sixties systems for computer algebra have been developed, that can represent exactly numbers like $\sqrt{\pi}$ and perform symbolic computations quite well.[3]

Systems for computer mathematics go an essential step beyond this. They can deal with arbitrary mathematical notions. For example, it is possible in these systems to represent exactly a Hilbert space or more complex mathematical structures. This is possible because one can formulate statements involving quantifiers ($\forall, \exists$) and predicates. Also the CM systems provide support for mathematical reasoning, for example by manipulating complex expressions. One fundamental difference between equational reasoning (both numerical and symbolic) and reasoning with predicates is that in many cases the former is decidable, whereas the latter usually is not. For this reason systems of computer mathematics are interactive, whereby the mathematician takes the lead.

Before going into more detail we first want to mention three important contributions of the Greek philosopher Aristotle (384-322 B.C.) to the field of computer mathematics. He established the following.

**Description of the axiomatic method.**

Aristotle distinguished *concepts* and *propositions*. Concepts need to be defined and propositions need to be proved[4]. But in order to have a proper

[3]Since several systems of CA state for example that $(\sqrt{x})^2 = x$ without requiring that $x \geq 0$, the diligent judgement of a mathematician remains necessary.

[4]A *proof* is a sequence of statements, such that each one is either an axiom or follows from previous ones by reason. A proof $p$ is *proving* $A$, if $A$ is the last statement in the sequence $p$. The discovery of proofs (attributed to Thales (625-545 B.C.)) is one of the gratest inventions of humanity. They occur in various degrees of precision (depending on the refinedness of the statements and the reasoning) and are the main foundation for the reliability of science.

start one needs *primitive concepts* and primitive propositions, the so called *axioms*. Not much later Euclid (around 300 B.C.) carefully based his famous book *Elements* on this axiomatic method.

**Formulation of the quest for logic.**
Using the axioms, primitive and defined concepts, one can prove propositions by means of steps motivated by intuitive reasoning. Aristotle wanted to provide a set of explicit rules (the so-called *syllogisms*), that is powerful enough for most intuitively valid proofs. Although his teacher Plato (427-347 B.C.) was against this program, Aristotle did succeed partially, by singling out a correct (but incomplete) set of syllogisms. But more important was, that Aristotle had the courage to state the problem of formalising reason.

**Distinction between proof-checking and theorem proving.**
Aristotle said, that if someone showed him a proof of a statement, then he would be able to verify the correctness of that proof (and thereby of that statement relative to the axioms). If, however, he would be asked to prove a given theorem (of which a proof existed, but that was not given to him), then he would not always be able to do so.

The philosopher and mathematician G.W. Leibniz (1646-1717) went further, by expecting more from formalisms and machines. He wanted to construct

- a language $L$, in which arbitrary problems could be formulated;
- a machine[5], that could determine the correctness of statements in $L$.

But, as was hinted at by Aristotle (and proved later by Turing), such automated deduction is not possible in general. An actual system of computer mathematics is less powerful than Leibniz would have wanted. It consists of a user interface in which the user can construct a so-called (mathematical) context:

- primitive notions, axioms and defined notions;

furthermore one can formulate

- statements;

and for some of these statements one can construct interactively

- proofs.

[5]It is interesting that around 1700 the belief in machines was such, that Leibniz had in mind to ask as first question: 'Does God exist?'.

The computer will verify whether the definitions are well-formed and whether the proofs are correct. Such definitions and proofs need to be given in a fully formal way, otherwise they cannot be verified mechanically. A fully formal proof is called a *proof-object*.

It is clear, how much this technology is related to the three ideas of Aristotle. His program to find a complete system of logic, was completed by Frege (in 1879, more than 2200 years after the original quest), building upon work of Leibniz, Boole and Peirce. (Frege did start with a formalisation, but unfortunately used an inconsistent system.) Soon after, Russell and Whitehead gave fully formalised version of small fragments of consistent mathematics (*Principia Mathematica,* 1910). Based on this work are the fundamental results stating that arithmetic is essentially incomplete (Gödel, 1931) and undecidable (Turing, 1936). For practical purposes, however, Russell and Whitehead's system is not adequate for full formalisation. This is so, because the system does not contain names, which causes actual theories to become exponentially large; moreover, there is a need for substitution instances of theorems, which in *Principia* were indicated informally.

The idea of machine verified proofs originated[6] with the Dutch mathematician N.G. de Bruijn, who in 1968 designed for this purpose a family of languages generically called Automath. See Nederpelt et al. [1994]. Inspiration for this came from the meaning of the logical connectives, as put forward by Brouwer and Heyting. (The ideas are also related to work of Gentzen, Church and Howard.)

As was pointed out by Aristotle proof search is essentially more difficult than proof-checking. By the definition of proof, automated proof verification is always possible, while automated deduction is not. Nevertheless, for special areas of interest there are good systems for automated deduction. For example, the geometry prover of Chou and Wu, see Bundy [1994], p. 393, can derive automatically Morley's theorem concerning triangles in which the three angles are trisected. But this is a statement in a decidable theory. Another example is concerned with predicate logic. Although this theory is undecidable, one can derive automatically a class of tautologies of predicate logic, that are more difficult than those used in most mathematical texts.

In spite of the remark of Aristotle that proof-checking is different from theorem proving, systems for CM usually incorporate both. The reason is that in a pure proof-checker it is very boring to write down a formal proof. On the other hand fully automated theorem proving is impossible. In several theorem provers one needs to give a sequence of lemmas, in order that an interesting result can be proved. So there is a spectrum between proof checkers and theorem provers.

[6]Added in print: R. Boyer pointed out to me that also in McCarthy [1962] automated proof-checking was introduced and the first steps were made. In fact McCarthy's provident paper is rather close to the present one. An essential difference is that the use of type theory (see below) and natural deduction proofs is not discussed by McCarthy. In type theory one can not only efficiently formalise proofs but also mathematical notions.

### 3.2. Why computer mathematics?

Automated proof-checking and the development of formal definitions, statements and proof-objects is important for the following reasons.

**Correctness.**

Both in the field of mathematics and in that of system design there is a need for correctness. In the two disciplines the problems are somewhat different.

In mathematics the body of knowledge has become very large and complicated. Verification by humans of proofs of complicated theorems is done in the so-called sociological way: the proof is divided into parts, that are checked by different specialists. A prime example is the result about the structure of the simple finite groups, with a proof of more than 20.000 pages. If one succeeds in generating formal proof-objects for such complicated theorems, then the computer verification will add to the acceptance of these (methodological considerations will be given below). But this is an extreme example. It is also important to have automatic proof verification for less spectacular results. If a mathematician sees a useful but unknown result, with a proof of, say, 20 pages long, then it saves time to know beforehand whether the result is correct or incorrect. Moreover, it is useful to have mathematical theorems in a verified library, because the size of the collection of results is so rapidly growing, that it has become impossible for one person to understand all of it or even just know about all of it. In this way arbitrary details can be looked up and be combined.[7]

In the field of systems design (hardware and software) the problem of verification is different from that in usual mathematics. Here most proofs are long but intellectually not stimulating. This implies that machine verification is essential. As mentioned in section 2 this is done successfully for hardware design, but not yet in software design because of the lack of specification and proof tools. I expect, that when the technology of CM will be mature, it will have a strong influence on software design.

**Support.**

Systems of CM may facilitate the construction of large proofs (both in mathematics and in system design). These constructions can be done top-down or bottom-up. One may leave some lemmas unproven or even concepts undefined; the necessary details can be filled in later. Of course even without a system of CM this can be done. But the help of such a system consists

[7] It should be admitted that formalising mathematics is difficult. This is not so much because of the length of proofs but because of their depth. Also standard mathematical proofs contain jumps that are not formal, but are clear to an artisan in the field. It is fair to say, that at present mathematical proofs can be verified better by a human than by a computer, because of the mentioned difficulty in formalisation. In §§3.3, 3.4 it is indicated how this may change.

in verifying the well-formedness of definitions and the correctness of proofs. Moreover, the systems keep a record of those details that are still left out. Another support by systems of CM consists in generating formal proofs from so-called *tactics*, to be discussed in the next subsection.

**Program extraction.**
If a statement of the form '$\forall x \exists y \ldots$' is proved, then this often gives rise to an algorithm that finds the $y$ in terms of $x$. If the proof is given formally, then the algorithm can be extracted automatically in the form of a program, see e.g. Parent [1994].

**Education.**
As it is a fact, that in several 'civilised' countries the notion of proof is not taught anymore at the level of high-school, it becomes necessary that university students of mathematics, science and technology get acquainted with them as soon as possible. Interactive systems for proof development will be of definite help, notably because such systems are patient. Moreover, the proofs can be found only, if one understands what one is doing.

**Cultural value.**
Suppose, that with the support of a CM system writing verified mathematics is not much more difficult than writing an intuitively correct paper in TEX, then a new standard of precision may emerge. By building a library of verified results, mathematics may be protected against corruption in times that the subject is not cultivated anymore (as essentially happened in the middle ages). In Bundy [1994], pp. 238-251, a dramatic but non-Utopian plea for building such a library is formulated as the *QED manifesto.* One quotation: '[building such a library is] *of significant cultural character. .... Like the great pyramids, the effort required (especially early on) may be great; but the rewards can be even more staggering than this effort'*.

**Foundational interest.**
It is an interesting challenge to see, whether it is possible to build systems of CM, such that it does not require too much effort to construct proof-objects. In this respect de Bruijn has as thesis, that in a proper system of CM the length of a formal proof or required tactics is just a constant factor times the length of a complete intuitive proof. Experience so far is in favour of this thesis. For the first generation prototypes the factor is about 30; for the second generation that uses tactics it is about 10. Also it is of interest to study, in which class of formal systems proofs can be represented well (set theories vs. type theories; other systems).

There may be a methodological objection to the idea of computer verification. If a mathematical statement is verified for its correctness by a computer, are we willing to believe that statement? It could be the case that there is a mistake in the design of the verifying program. This question has a satisfactory answer. If the verification is warranted by a proof-object that is made public and that is verifyable by a relatively simple method (by a program consisting of a few pages), then one can recheck the statement locally, i.e. on a PC with ones own personal proof-checker. Under these conditions of repeatability, one can trust the correctness of the statement at least as much as (or even more than) the safety of a bridge over which one is going to walk.

### 3.3. Formal systems for CM

Systems of computer mathematics with portable proof-objects (as required by the quest for reliability discussed above) are to be done in a formal system $T$ that should enjoy the following list of properties.

1. $T$ is *adequate*. This means that the usual mathematical concepts, statements and proofs can be expressed (in a natural way) formally in $T$. Adequacy requires the following particulars.
   (a) Adequacy for *defining*. The system $T$ has sufficiently rich concepts and allows the introduction of names.
   (b) Adequacy for *reasoning*. The usual logical deductions occurring in mathematics are representable as proof-objects of $T$.
   (c) Adequacy for *computing*. Symbolic (and numerical) computations are possible in $T$. Also equational reasoning is possible in $T$.
2. $T$ is *faithful*. This means that $T$ is conservative in the sense that if it states that $P$ is a proof of statement $S$ (in context $\Gamma$), then the intuitive statement $S$ is provable in ordinary mathematics relative to the corresponding context.
3. $T$ is *efficient* (for the machine). The verification of the well-formedness of a definition and the correctness of a proof can be verified in a feasible way.
4. $T$ is *practical* (for the human user). Writing mathematics in $T$ is not much more difficult than writing it in informal language.

Following the ideas in the languages of the Automath family, now a wide spectrum of systems of *type theory* are used as formal system $T$. See Nederpelt et al. [1994], 229-247 for a discussion. The basic idea is explaned in fig. 3. The most simple of these are called Pure Type Systems (PTSs), see Barendregt [1992]. Under influence of D. Scott and P. Martin-Löf inductive types and extra reduction rules are added to the formal systems. The resulting extensions are called Type Systems (TSs). See Martin-Löf [1984] and Paulin-Mohring [1993] for a description.

Type theories are formal systems in which there is a natural way to represent statements and proofs. In fact, it seems more natural to encode mathematics in these systems than in the more conventional set theory. The reason is, that type theory has a natural way to use many-sorted logic (in order to deal with structures like vector spaces, in which there are vectors and scalars belonging to different 'sorts'); a natural way to formalise second and higher order logic (to reason about properties of propositions, or properties of properties of propositions; in this way one can formulate the notion of 'infinite'). To make a variation on a statement of Laplace[8], we can say that 'we do not need the hypotheses of set theory'. Moreover, set theory sometimes gives rise to unnatural questions. For example $\emptyset \in \sqrt{2}$? In type theory such questions cannot be formulated.

One important aspect of (P)TSs is that some proof steps (of definitional nature) do not need to be given explicitly. Suppose we have proved $P_1$ & $P_2 \Rightarrow Q$. Now define $P \equiv P_1$ & $P_2$. Then we have, of course, $P \Rightarrow Q$. In a (P)TS the same proof-object for $P_1$ & $P_2 \Rightarrow Q$ works also for $P \Rightarrow Q$. This $P$ and $P_1$ & $P_2$ are said to be *definitionally equal* and share the same inhabitants (by a rule of PTSs). In TSs more equalities hold in this way. For example there exists a term Sq (for squaring), such that for a natural number like 3 one has $\mathrm{Sq}\ (3) = 9$ definitionally. So a proof of $A(9)$ is also a proof of $A(\mathrm{Sq}(3))$. This is the essential difference between TSs and PTSs.

Now we will discuss how the list of requirements apply to type theories.

1(a). As was already mentioned, type theories are strong enough to represent most mathematical reasoning. In addition to this adequacy, the extra data types available in TSs make representations more easy in these systems. It is known also in programming that extra data types makes life much more easy. Surprisingly many concepts in mathematics are related to inductively defined data types (for example the notion of polynomial).

1(b). Although proofs of most tautologies (syllogisms) of predicate logic, that are needed in mathematics as deduction steps, can be found automatically by resolution methods, this does not mean that the problem of formalising logical steps is solved. The reason is that one needs names in mathematics. Now even if

$$\mathrm{name}_1 \;\Rightarrow\; \mathrm{name}_2$$

is a tautology, this is so, only after the names are replaced by the proper expression they stand for. This so-called 'unfolding' should not be done fully, because then the expressions become exponentially large. So the problem boils down to deciding what names have to be unfolded.

1(c). Equational reasoning is not yet incorporated in a feasible way in TSs. Systems of CA can produce valid equations. Several ways of doing this in CM systems are being studied: the *believing* way, in which the CM system just accepts equations produced by a CA system; the *skeptic* in which the CA system is forced to give evidence (a proof-object) for each statement it sends to the CM system;

[8]Napoleon remarked to Laplace that in his work 'Mécanique Céleste' he did not mention the author of the universe. Laplace answered: 'Sire, I did not need that hypothesis.'

and finally the *autarkic* way in which the CM system learns to do equational reasoning by encorporating some verified term rewriting techniques.

Each statement $A$ of informal (but precise) mathematics can be translated as a formal statement $(A)$ in logic. A mathematical context $\Gamma$ consists of a set of axioms and definitions. When we write these formally, we obtain a formal context $(\Gamma)$. Predicate logic is such that we can derive $(A)$ from $(\Gamma)$ exactly when $A$ is informally provable from $\Gamma$.

Type theory goes one step further. A statement $A$ is transformed into the type (i.e. collection)

$$[A] = \text{ the set of proofs of } A.$$

So $A$ is provable if and only if $[A]$ is 'inhabited' by a proof $p$. Now a proof of $A \Rightarrow B$ consists (according to the Brouwer-Heyting interpretation of implication) of a function having as argument a proof of $A$ and as value a proof of $B$. In symbols

$$[A \Rightarrow B] = [A] \rightarrow [B].$$

Similarly

$$[\forall x{\in}A.Px] = \Pi x{:}A.[Px],$$

where $\Pi x{:}A.[Px]$ is the cartesian product of the $[Px]$, because a proof of $\forall x{\in}A.Px$ consists of a function that assigns to each element $x{\in}A$ a proof of $Px$. In this way proof-objects become isomorphic with so-called *natural deduction proofs*. Using this interpretation a proof of $\forall y{\in}A.Py \Rightarrow Py$ is $\lambda y{:}A\lambda x{:}Py.x$. Here $\lambda x{:}A.B(x)$ denotes the function that assigns to input $x{\in}A$ the output $B(x)$.

Verifying whether $p$ is a proof of $[A]$, boils down to verifying whether in the given context the type of $p$ is equal to $[A]$.

Fig. 3. The essence of proof-checking using type theory.

2. Unfortunately faithfulness is only known for several adequate PTSs and not for the corresponding more practical TSs. It is conjectured, however, that the right TSs are faithful.

3. As to efficiency, both for the PTSs and the TSs correctness is decidable by a simple program. But the verification is in general not feasible in these systems. It is, however, an empirical fact, that if formal definitions and proofs in these type theories come from definitions and proofs understood by a human, then the verification of correctness is feasible.

4. Even the stronger TSs are not yet practical. What is needed, is a kind of higher language (in the sense that FORTRAN and PASCAL are higher programming languages than assembler) that is convenient to express mathematics but

that can be translated easily to the more low-level language of (P)TSs. Such a language is called by de Bruijn a *mathematical vernacular.*

### 3.4. Implementations

The first serious prototype system of CM was the Automath proof-checker build in 1970, see Nederpelt et al. [1994], pp. 783-804. In this system the proof-object had to be constructed by hand. It required a *mathematical monk* to formalise a non-trivial part of mathematics. In the second generation prototypes the proof-objects are generated via so-called tactics. A tactic is a sequence of commands that the user can give to the system; from this a proof-object can be compiled automatically. Suppose, for example, that one has to create a formal proof for

$$\forall x, y{\in}A\ [P(x, y) \Rightarrow Q(x)] \tag{+}$$

(the 'goal') from a certain context. Then one 'pushes a button' and the system adds to the context that $x, y{\in}A$ and the assumption $P(x, y)$. Now the goal is to prove $Q(x)$ from the extended context. As soon as this is done the system provides a proof for (+). See fig. 4 for an example of tactics. Not shown are the answers of the system after each statement made by the user. These answers consists of new (simpler) goals—as described above—so that the human does not get lost while designing the proof of (+).

These tactics do not constitute a vernacular. The reason is that the tactics are close to the syntactical structure of the formal proof, rather than to the mathematical idea of the informal proof.

```
(*Theorem Drinkers'_principle.*)
Goal ({P:Prop} P \/ ~P) ->
     {Cafe : Type} {w:Cafe} {Drunk : Cafe -> Prop}
     Ex [x:Cafe] (Drunk x) -> {x:Cafe} Drunk x;
==============================================
  intros EM ___;
  Refine EM ({x:Cafe} Drunk x);
  Intros _; Refine ExIntro; Refine w;
  intros _; Immed;
  intros; Refine EM (Ex [x:Cafe] ~(Drunk x));
  intros; Refine H1; intros; Refine ExIntro; Refine t;
  intros _; Refine H2 H3;
  intros;  Refine H;
  intros; Refine EM (Drunk x);
  intros; Immed;
  intros; Refine H1;
  Refine ExIntro; Refine x; Immed;
Save Drinkers'_principle;
```

Fig. 4. Tactics.

The principal second generation prototype systems for CM with portable proof-objects are NuPrl, see Constable et al. [1986], Coq, see Dowek et al. [1993], and Lego, see Luo et al. [1992]. These systems have as extra features the following.

1. tactics;

2. term rewriting;

3. (some form of) automated deduction.

Tactics make it much easier for the human user to construct a proof-object. NuPrl was the first system based on type theory using tactics. Many theorems (mathematically relatively simple ones) are proved using it.

Encorporation of term-rewriting makes it more easy to deal with symbolic and other forms of computation. In particular the autarkic way of incorporating CA can make use of this facility.

Automated deduction based on resolution solves some of the logical steps to be made. As pointed out before, diligent use of unfolding definitions is necessary. In some versions of Coq this can be done by 'clicking' on the name. Lego has some 'automated' unfolding, necessary for ease of use. One of the newer features of Coq is the automatic translation of a proof-object into a proof in natural language, see Coscoy et al. [1995].

THEOREM (Drinkers' principle). *Let $U$ be a non-empty set and let $Q$ be a predicate on $U$. Then*

$$\exists x{\in}U.[Q(x) \rightarrow \forall y{\in}U.Q(y)].$$

PROOF. We distinguish two cases.
Case 1. $\forall y.Q(y)$. Then an arbitrary $x{\in}U$ makes the implication true.
Case 2. $\neg\forall y.Q(y)$. Then $\neg Q(x_0)$ for some $x_0{\in}U$. Now take $x = x_0$, to make the implication vacuously true. ■

Fig. 5a.

```
Theorem  Drinkers' principle.
Statement
 (∀P: Prop. P ∨ ¬ P) ⇒
  ∀U: Set.  U ⇒ ∀Q: U ⇒ Prop. ∃x: U.  (Q  x) ⇒ ∀x0: U. (Q  x0).
Proof
 λH': ∀P: Prop. P ∨ ¬ P.  λU: Set.  λH'0: U.  λQ: U ⇒ Prop.
  (or_ind
      ∀x0: U. (Q  x0)  ¬ (∀x0: U. (Q  x0))  ∃x: U.  (Q  x) ⇒ ∀x0: U. (Q  x0)
      λH'1: ∀x0: U. (Q  x0).
       (ex_intro  U  λx: U.  (Q  x) ⇒ ∀x0: U. (Q  x0)  H'0  λH'2: (Q  H'0). H'1)
      λH'1: ¬ (∀x0: U. (Q  x0)).
       (or_ind
           ∃x: U.  ¬ (Q  x)  ¬ (∃x: U.  ¬ (Q  x))  ∃x: U.  (Q  x) ⇒ ∀x0: U. (Q  x0)
           λH'2: ∃x: U.  ¬ (Q  x).
            (ex_ind
                U  λx: U. ¬ (Q  x)  ∃x: U.  (Q  x) ⇒ ∀x0: U. (Q  x0)
                λx: U.  λE: ¬ (Q  x).
                 (ex_intro
                     U  λx0: U.  (Q  x0) ⇒ ∀x1: U. (Q  x1)  x
                     λH'3: (Q  x). (False_ind  ∀x0: U. (Q  x0)  (E  H'3)))  H'2)
           λH'2: ¬ (∃x: U.  ¬ (Q  x)).
            (ex_intro
                U  λx: U.  (Q  x) ⇒ ∀x0: U. (Q  x0)  H'0
                λH'3: (Q  H'0).  λx: U.
                 (or_ind
                     (Q  x)  ¬ (Q  x)  (Q  x)  λH'4: (Q  x). H'4
                     λH'4: ¬ (Q  x).
                      (False_ind  (Q  x)  (H'2  (ex_intro  U  λx0: U. ¬ (Q  x0)  x  H'4)))
                     (H'  (Q  x))))  (H'  ∃x: U.  ¬ (Q  x)))  (H'  ∀x0: U. (Q  x0))).
```

Fig. 5b.

```
Theorem  Drinkers' principle.
Statement
 (∀P: Prop. P ∨ ¬ P) ⇒
  ∀U: Set.  U ⇒ ∀Q: U ⇒ Prop. ∃x: U.  (Q  x) ⇒ ∀x0: U. (Q  x0).
Proof
 Assume  ∀P: Prop. P ∨ ¬ P  (H')
     Let  U  be a set
     Consider an arbitrary element  H'0  in  U
         Let  Q  :  U ⇒ Prop
         Specializing  H'  to  ∀x0: U. (Q  x0),  we get
         (∀x0: U. (Q  x0)) ∨ ¬ (∀x0: U. (Q  x0))
         So we have two cases :
         a)  Assume  ∀x0: U. (Q  x0)  (H'1)
                 Assume  (Q  H'0)  (H'2)
                     We have  H'1
                 We have proved  (Q  H'0) ⇒ ∀x0: U. (Q  x0)
                 We have found an element  x  that verifies  (Q  x) ⇒ ∀x0: U. (Q  x0),
                 namely  H'0
         b)  Assume  ¬ (∀x0: U. (Q  x0))  (H'1)
                 Specializing  H'  to  ∃x: U.  ¬ (Q  x),  we get
                 (∃x: U.  ¬ (Q  x)) ∨ ¬ (∃x: U.  ¬ (Q  x))
                 So we have two cases :
                 a)  Assume  ∃x: U.  ¬ (Q  x)  (H'2)
                         Choose an element  x  in  U  such that  ¬ (Q  x)  (E)
                         Assume  (Q  x)  (H'3)
                             From  H'3  and  E,  we deduce  a contradiction
                             So, this case cannot happen
                         We have proved  (Q  x) ⇒ ∀x0: U. (Q  x0)
                         We have found an element  x0  that verifies
                         (Q  x0) ⇒ ∀x1: U. (Q  x1),  namely  x
                 b)  Assume  ¬ (∃x: U.  ¬ (Q  x))  (H'2)
                         Assume  (Q  H'0)  (H'3)
                             Consider an arbitrary element  x  in  U
                                 Specializing  H'  to  (Q  x),  we get  (Q  x) ∨ ¬ (Q  x)
                                 So we have two cases :
                                 a)  Assume  (Q  x)  (H'4)
                                         We have  H'4
                                 b)  Assume  ¬ (Q  x)  (H'4)
                                         We have  H'4
                                         We have found an element  x0  that verifies
                                         ¬ (Q  x0),  namely  x
                                         so, from  H'2,  we deduce  a contradiction
                                         So, this case cannot happen
                                 We have  (Q  x)  in both cases
                         We have proved  (Q  H'0) ⇒ ∀x: U. (Q  x)
                         We have found an element  x  that verifies
                         (Q  x) ⇒ ∀x0: U. (Q  x0),  namely  H'0
                 We have  ∃x: U.  (Q  x) ⇒ ∀x0: U. (Q  x0)  in both cases
         We have  ∃x: U.  (Q  x) ⇒ ∀x0: U. (Q  x0)  in both cases
 We have proved
 (∀P: Prop. P ∨ ¬ P) ⇒
  ∀U: Set.  U ⇒ ∀Q: U ⇒ Prop. ∃x: U.  (Q  x) ⇒ ∀x0: U. (Q  x0).
```

Fig. 5c.

Fig. 5. Various forms of proofs of Smullyans 'Drinkers' principle': a. the informal proof; b. the proof-object; c. the proof-object translated back into natural language.

In fig. 5 one can see for Smullyans 'Drinkers' principle'[9] an informal proof (in

[9]The principle states that in a room with people there always is at least one person, such

'my best mathematical style'), the proof-object, and finally a translation of that formal proof into natural language. The formal proof is obtained through an interactive session in which the user provides tactics to the machine, see fig. 4.

Although the translated proofs in natural language are somewhat 'stiff', these may turn out to be useful for the construction of a vernacular. The reason is that seeing a formal proof-object does not easily give understanding, while a proof in natural language does. In particular this is so, when sufficiently many details that are obvious to a human are elided.

### 3.5. Related work

There are several systems for CM based on a different methodology. Not all of these have portable proof-objects, and therefore one has to have a belief in the design of them. Nevertheless, these systems are rather interesting.

A system of CM based on some form of TS, but without proof-objects is Isabelle, see Paulson [1994]. This system has a good module for term rewriting, which is important for equational reasoning.

Systems of CM not based on TSs are HOL, see Gordon et al. [1994], the Boyer-More theorem prover, see Boyer et al. [1988], OTTER, see Wos et al. [1992], and MIZAR, see Rudnicki [1992]. HOL is based on higher order logic and has been used for hardware verification. The Boyer-More theorem prover is based on a formal system called *primitive recursive arithmetic*. Because this system is relatively weak—it has no quantifiers and only states universal propositions—there are more strategies for automated proof search for it, than for the stronger theories. OTTER is based on the resolution method and is able to find many proofs of tautologies in predicate logic used in intuitive mathematical proofs. MIZAR is based on set theory formulated in predicate logic. Many theorems (again relatively simple ones) have been proved in these systems.

## 4. Conclusion

Systems for computer mathematics are very promising. Nevertheless, presently they still have some weak points. There is a need for a good vernacular to make formalising more natural; there is a need for a good way to handle symbolic computations; and finally for the TSs used one needs to prove the faithfulness for the formalisations.

I expect, that within a decade systems for CM are more mature. In particular they will include (or use) the power of systems for CA to deal with equations. Then CM will be essentially stronger than CA, because of the fact that statements can be proved. (Working with CA systems one may overlook necessary side-conditions.) Two interesting uses are probable. One in the field of interactive

that if that person starts to drink, then everybody in the room starts to drink.

development of mathematics and one in the field of software design.

The interactive development of mathematics does not imply 'Death of proof' or the end of human involvement with mathematics, as some have claimed. On the contrary, both proofs and the the ingenuity of the user will play an essential role in computer mathematics and its applications. Proofs are essential, because without them there is no warranted correctness; humans are essential, because otherwise proofs cannot be found.

Limited experience with systems of CM have shown that the phase of defining concepts is very essential. Once sufficient experience is obtained with handling complicated notions, I expect applications to specification and correctness of software systems. A necessary condition is, that software is written in a modular way, as is possible in e.g. functional languages. Some researchers express doubts, that the design methodology of the Chinese box will be sufficient to produce correct software. They do believe, however, in program extraction from verified proofs. In any case, precise specifications and proofs will be important.

**Acknowledgements**

I thank the following persons for useful information: G. Barthe, M. Bezem, R. Boyer, A. Cohen, R. Constable, H. Elbers, H. Geuvers, G. Huet, H. Meijer, R. Plasmeijer, R. Platek, R. Pollack, M. Ruys, F. Vaandrager and H. Wupper.

## References

BARENDREGT, H.P.

[1992] Typed Lambda calculi, in: *Handbook of Logic in Computer Science*, S. Abramsky et al. (Eds.), Oxford University Press, 117-309.

BARENDREGT, H.P. and T. NIPKOW (EDS.)

[1994] Types for Proofs and Programs, Lecture Notes in Computer Science, 806, Springer, Berlin.

BOYER, R.S. and J.S. MOORE

[1988] *A computational logic*, Academic Press, New York.

BUNDY, A. (ED.)

[1994] Automated Deduction—CADE-12, Lecture Notes in Artificial Intelligence, 814, Springer, Berlin.

CONSTABLE, R. ET AL.

[1986] *Implementing Mathematics with the NuPrl Proof Development System,* Prentice Hall, London.

COSCOY, Y., G. KAHN and L. THÉRY

[1995] Extracting text from proofs, in: *Typed Lambda Calculus and Applications*, M. Dezani-Ciancaglini (Eds.), Lecture Notes in Computer Science 902, Springer, Berlin, 109-123.

DOWEK, G. ET AL.

[1993] The Coq Proof Assistant User's Guide—Version 5.8. Projet Formel, INRIA, Rocquencourt.

GORDON, M.J.C. and T.F. MELHAM

[1993] *Introduction to HOL: a theorem proving environment for higher order logic*, Cambridge University Press, Cambridge.

LUO, Z. and R. POLLACK

[1992] LEGO proof development system: User's manual, Technical Report ECS-LFCS-92-211, Computer Science Department, University of Edinburgh.

MARTIN-LÖF, P.

[1984] *Intuitionistic Type Theory*, Studies in Proof Theory, Bibliopolis, Napoli.

MCCARTHY, J.

[1962] Computer Programs for Checking Mathematical Proofs, In: *Recursive Function Theory, Proceedings of a Symposium in Pure Mathematics*, V, American Mathematical Society, Providence, RI, 219-227.

MELHAM, T.F. and J. CAMILLERI (EDS.)

[1994] Higher Order Logic Theorem Proving and its Application, Lecture Notes in Computer Science, 859, Springer, Berlin.

MUSIL, R.

[1952] *Der Mann ohne Eigenschaften,* Rowohlt, Hamburg.

NEDERPELT, R.P., J.H. GEUVERS and R.C. DE VRIJER (EDS.)

[1994] Selected Papers on Automath, Studies in Logic 133, North-Holland, Amsterdam.

PARENT, C.

[1994] Certified programs in the system Coq—The program tactic, in: H Barendregt et al. [1994], 291-312.

PAULIN-MOHRING, C.

[1993] Inductive Definitions in the Sytem Coq—Rules and Properties, in: *Typed Lambda Calculus and Applications,* M. Bezem et al. (Eds.), Lecture Notes in Computer Science 664, Springer, Berlin, 328-345.

PAULSON, L.

[1994] *Isabelle: A generic theorem prover*. Lecture Notes in Computer Science 828, Springer, Berlin.

PLASMEIJER, R. and M. VAN EEKELEN

[1993] *Functional Programming and Parallel Graph Rewriting*, Adison Wesley, New York.

POUNTAIN, D.

[1994] Functional programming comes of age, in: *Byte,* August 1994, 183-184.

RUDNICKI, P.

[1992] An overview of the MIZAR project. Available by anonymous FTP from menaik.cs.ualberta.ca as `pub/Mizar/Mizar_Over.tar.Z`.

WOS, L., R. OVERBEEK, R. LUSK and J. BOYLE

[1992] *Automated Reasoning: Introduction and Applications*, McGraw-Hill, New York.