Journal of Automated Reasoning **28:** 321–336, 2002.
© 2002 *Kluwer Academic Publishers. Printed in the Netherlands.*

Autarkic Computations in Formal Proofs

HENK BARENDREGT and ERIK BARENDSEN
Computing Science Institute, University of Nijmegen, Toernooiveld 1, 6525 ED Nijmegen, The Netherlands. e-mail: {henk,erikb}@cs.kun.nl

(Received: 1 December 1999)

Abstract. Formal proofs in mathematics and computer science are being studied because these objects can be verified by a very simple computer program. An important open problem is whether these formal proofs can be generated with an effort not much greater than writing a mathematical paper in, say, LATEX. Modern systems for proof development make the formalization of reasoning relatively easy. However, formalizing computations in such a manner that the results can be used in formal proofs is not immediate. In this paper we show how to obtain formal proofs of statements such as $\texttt{Prime}(\underline{61})$ in the context of Peano arithmetic or $(x+1)(x+1) = x^2 + 2x + 1$ in the context of rings. We hope that the method will help bridge the gap between the efficient systems of computer algebra and the reliable systems of proof development.

Key words: formal proof, computer algebra, proof development, autarkic cmputation.

1. The Problem

Ordinary mathematical exposition is informal but precise: one speaks of *informal rigor*. Formal mathematics, on the other hand, consists of definitions, statements, and proofs having such a complete level of detail that its correctness (relative to a context in which the primitive notions and axioms are introduced) can be verified by computer. Using certain systems of type theory, such formalizations assume a canonical form. The use of computer-verified proofs is discussed by, for example, McCarthy et al. (1962), de Brujin (1970, 1990), Constable (1986), Bundy (1984), Barendregt (1996), and Cohen (1996). An important question is whether it is feasible to construct fully formalized proofs. Feasibility is meant here in a sense stricter than in computer science: A mathematician should be able to generate correct formal mathematics with an effort comparable to writing an article in, say, (La)TeX. Systems for proof development are interactive programs that help the user to generate formal mathematics. Such systems are capable of exactly representing arbitrary mathematical notions (such as an infinite-dimensional Hilbert space) and ask the human user to give hints for a proof of a property formulated using these notions. The detailed formal proof will be produced when all requests are fulfilled.

An important group of systems for proof checking is based on type theory, all derived from the AUTOMATH family of Brujin (1970). These systems have so-called canonical public proof objects that can be verified locally by other groups.

Modern prototype systems for proof development based on type theory are, for example, Coq (see Coquand and Huet (1988)) and LEGO (see Luo and Pollack (1992)).

Systems of computer algebra can deal very successfully with certain parts of mathematics, namely, symbolic equational reasoning. But systems of computer algebra do not have a notion of proof. One could imagine these systems extended to include proofs of the equations that are claimed. But systems for computer algebra are essentially equational; they cannot deal with arbitrary quantifiers. In principle, systems of computer mathematics (henceforth: CM) are far more powerful.[⋆]

In case studies it has been noticed (see, e.g., Ruys (1999)) that although the representation of logical reasoning is relatively straightforward, attempts to include equational reasoning run into problems. This may sound surprising because systems of computer algebra are very good at such tasks. The reason for the difficulty is that the formalization of equational reasoning in systems of proof development that first comes to mind proceeds via first-order predicate logic with equality. The axiom or rule that the operations are compatible with equality (the congruence axiom) states

$$x = x' \;\Rightarrow\; x + y = x' + y \;\&\; y + x = y + x'.$$

If we now want to show that, for example,

$$x = x' \;\Rightarrow\; z + ((x + y) + z) = z + ((x' + y) + z),$$

several of these steps are required. This causes formal proofs of simple equations to be quadratic in the size of the terms involved. This difficulty can be avoided by formally proving “metamathematical” results like

$$x = x' \;\Rightarrow\; \forall C[\,]\, C[x] = C[x'].$$

The two technologies, computer algebra and proof development, are comparable to Babylonian and Greek mathematics. The former is much better at algorithmic computing but does not have a notion of proof. The latter does have a notion of proof but stumbles over equational reasoning. The reason that, say, Euclid has difficulties with simple algebraic equations is that these are first translated into geometric propositions that only then are proved. (It is said that the problems with incommensurable magnitudes is one of the factors that led the Greeks to do algebra via geometry.) The power of algebra is that it permits the direct manipulation of the syntactic expressions themselves.

We emphasize that in algebra not only is the meaning of expressions in an equation important, but also the expressions themselves, that is, their syntactic

[⋆] An interesting intermediate system is the Boyer–Moore theorem prover (1988). This system uses open formulas of primitive recursive arithmetic that are tacitly universally quantified; statements involving alternations of quantifiers such as $\forall x \exists y A(x, y)$ are translated into their Skolem form $A(x, f(x))$.

form. More explicitly, if we have an equation like $t = s$ (for example, $2 + 2 = 4$), then the message is not only that $[\![2 + 2]\!] = [\![4]\!]$ (which, in the example, becomes $SSSS0 = SSSS0$). The importance of $s = t$ is that the expressions "t" and "s" when evaluated yield the same result. Goethe has criticized mathematics by stating that $2 + 2 = 4$ is trivial because 4 is just another name for $2 + 2$.[⋆] His criticism can be refuted, among other ways, by the above argument.

In this paper we sketch methods for merging the two technologies of computer algebra and proof development. We hope that this will soon lead to the emergence of what can properly be called systems of computer mathematics.

2. Computations and Proofs: Methodologies

Computations are required in proofs in several situations. On such situation arises when we want to prove formal versions of the following intuitive statements:

(1) $[\sqrt{45}] = 6$, where $[r]$ is the largest integer $\leq r$;
(2) `Prime`(61);
(3) $(x + 1)(x + 1) = x^2 + 2x + 1$.

A way to keep proof objects from growing too large is to employ a principle introduced by Poincaré. Poincaré (1902), p. 12, stated that an argument showing that $2 + 2 = 4$ "is not a proof in the strict sense, it is a verification" (actually he claimed that an arbitrary mathematician would agree with this remark).

We call this the Poincaré principle. In the AUTOMATH project of de Bruijn, the following interpretation was given to this principle: If p is a proof of $A(t)$ and $t =_R t'$, then the same p is also a proof of $A(t')$. Here R is a notion of reduction consisting of ordinary β reduction and δ-reduction in order to deal with the unfolding of definitions. Since $\beta\delta$-reduction is not too complicated to be programmed, the type systems enjoying this interpretation of the Poincaré principle still satisfy the de Bruijn criterion.[⋆⋆]

Several styles for incorporating computations in formal proofs may be distinguished. We designate these styles as follows:

- accepting,
- skeptical, and
- autarkic.

[⋆] According to Goethe: "Mathematics has the completely false reputation of yielding infallible conclusions. Its infallibility is nothing but identity. Two times two is not four, but is just two times two, and that is what we call four for short. But four is nothing new at all. And thus it goes on and on in its conclusions, except that in the higher formulas the identity fades out of sight."

[⋆⋆] The reductions may sometimes cause the proof-checking to have an unacceptable time complexity. We have that p is a proof of A iff $\texttt{type}(p) =_{\beta\delta} A$. Because the proof is coming from a human, the necessary conversion path is feasible, but to find it automatically may be hard. The problem probably can be avoided by enhancing proof objects with hints for a reduction strategy.

In the *accepting* style, the CM system does no computation but consults as an oracle a system of computer algebra. The result is then accepted by incorporating it as an extra axiom. For example, in a ring one has

$$(x+1)(x-1) = x^2 - 1$$

because a system of computer algebra says so.

In the *skeptical* style, again the CM system asks a system of computer algebra to make a computation. This time, however, the system of computer algebra is required to be "verbose," that is, to provide a trace of the computation. For example, in a ring one has $(x+1)(x+1) = x^2 + 2x + 1$ because

$$\left.\begin{array}{rcl}(x+1)(x+1) & = & (x+1)x + 1.(x+1)\\ & = & (x.x + 1.x) + (1.x + 1.1)\\ & = & (x.x + x) + (x+1)\\ & = & (x.x + (x + (x+1))\\ & = & (x.x + ((x+x)+1)\\ & = & (x^2 + 2x + 1).\end{array}\right\} \qquad (*)$$

From such a trace a proof object can easily be generated.

In the *autarkic* style a CM system will have to do the computation on its own.

Until systems for computer mathematics are better developed, the accepting style can be defended on pragmatic grounds. But we claim that in the long run the accepting style is methodologically unsatisfactory (a computer algebra system may contain bugs or a side condition may be left out).

The skeptical style is advocated by some researchers in computer algebra as superior to the autarkic style. The argument for this position is that the autarkic style requires a proof of global correctness (equality chains like $(*)$ hold for all possible traces), whereas the skeptical style requires only local correctness (the equality chain $(*)$ is valid).

Contrary to this opinion we claim that the autarkic style is both more efficient and hardly more expensive than the skeptical style. The reason for the greater efficiency is that in general the incorporation of $(*)$ in a proof object is quite expensive (computations may be long). This can be avoided, however, by what can be called the Poincaré principle, stating that once a (symbolic) algorithm is proved to be sound, its use is not a mathematical proof but a "verification." Extending the use of the Poincaré principle, Scott and Martin-Löf emphasized that steps of ι-conversion for inductive types (see below) should likewise not be registered in proof objects. Moreover, to establish the local correctness of a trace like $(*)$ is often almost all that is needed to establish the global correctness of a (symbolic) algorithm. Since such reductions are not too hard to program, the resulting proof checking still satisfies the de Bruijn criterion.

3. Autarkic Direct Computations

From now on let $\Gamma \vdash A : B$ denote derivability of $A : B$ in context Γ in a typical (formal system underlying a) CM system such as Coq or LEGO. To be specific, one may think of the calculus of constructions λC extended with inductive types with, as conversion, $R = \beta\iota$. This means that one works in the PTS specified by

$\mathcal{S}$	$*, \Box$
$\mathcal{A}$	$* : \Box$
$\mathcal{R}$	$(*, *), (*, \Box), (\Box, *), (\Box, \Box)$

(Barendregt, 1992) and extended with inductive types like

$$\mathsf{Nat} = (\mu X : *)(\text{zero} : X|\text{suc} : X{\to}X)$$

having as induction/recursion principle $R = \mathsf{Nat\text{-}elim}$ satisfying

$$R : \forall P : \mathsf{Nat}{\to} * [P(\text{zero}) \to \forall x{:}\mathsf{Nat}.[P(x){\to} P(\text{suc}(x)] \to \forall x{:}\mathsf{Nat}.P(x)];$$

and in context $P : \mathsf{Nat}{\to}*$, base:$P(\text{zero})$, step : $\forall x{:}\mathsf{Nat}.[P(x){\to} P(\text{suc}(x)]$

$$RP \text{ base step zero } \to_\iota \text{ base};$$
$$RP \text{ base step } (\text{suc}x) \to_\iota \text{ step } x(RP \text{ base step } x).$$

This means that one has the following assumptions built in:

$$\mathsf{Nat}{:}*, \text{zero}{:}\mathsf{Nat}, \text{suc}{:}\mathsf{Nat}{\to}\mathsf{Nat}$$

and

$$R : \forall P : \mathsf{Nat}{\to} * [P(\text{zero}) \to \forall x{:}\mathsf{Nat}.[P(x){\to} P(\text{suc}(x)] \to \forall x{:}\mathsf{Nat}.P(x)].$$

The constant $R = \mathsf{Nat\text{-}elim}$ of the given type states that one assumes the scheme of induction. By introducing the notion of reduction $\to_\iota$ this $\mathsf{Nat\text{-}elim}$ also works as an operator for primitive recursion (of higher type). Note that the ι-reduction steps preserve typing. Also, one has the conversion rule stating that

$$\Gamma \vdash A : B,\ B =_R B',\ \text{and}\ \Gamma \vdash B' : s\ \Rightarrow\ \Gamma \vdash A : B'.$$

Here R is the notion of reduction $\beta\delta\iota$, where δ stands for definitional expansion. The denotation $\Gamma \vdash B$ stands for $\Gamma \vdash p : B$, for some p.

DEFINITION 3.1. Let Γ be a context of our type system.

(i) A Γ-*set* is a term A such that

$$\Gamma \vdash A : *.$$

(ii) A (k-ary) Γ-*function* on A is a term F such that

$$\Gamma \vdash F : A^k{\to}A,$$

where $A^0 \to B = B$ and $A^{k+1} \to B = A \to (A^k \to B)$.

(iii) Let A be a Γ-set. A (k-ary) Γ-*relation* over A is a term R such that

$$\Gamma \vdash R : A^k \to * .$$

(iv) Let $\mathbf{A}$ be a set. Then $\mathbf{A}$ is *representable* in context Γ iff there is a Γ-set A and for each $a \in \mathbf{A}$ a term $\underline{a}$ such that

$$\Gamma \vdash \underline{a} : A$$
$$a = b \;\Leftrightarrow\; \underline{a} =_{\beta\iota} \underline{b}.$$

We say that $\mathbf{A}$ is represented by A in Γ.

(v) Let $\mathbf{A}$ be represented by A in Γ. A function $f : \mathbf{A}^k \to \mathbf{A}$ is λ-computable (in Γ) if there exists a Γ-function F on A such that for all $\vec{a} \in \mathbf{A}$

$$F\underline{\vec{a}} =_{\beta\iota} \underline{f(\vec{a})}.$$

In what follows $\mathbf{A}$ is represented by A in Γ.

A technique for handling an equation like $[\sqrt{45}] = 6$ in an autarkic manner is to use the Poincaré principle extended to the reduction relation $\twoheadrightarrow_\iota$ for primitive recursion on the natural numbers. Operations like $f(n) = [\sqrt{n}\,]$ are primitive recursive and hence are lambda definable (using $\twoheadrightarrow_{\beta\iota}$) by a term, say F, in the lambda calculus extended by an operation for primitive recursion R satisfying

$$R\;A\;B\,\ulcorner 0 \urcorner \to_\iota A$$
$$R\;A\;B\,(\mathtt{succ}\,x) \to_\iota B\,x\,(R\;A\;B\;x).$$

Then, as

$$\ulcorner 6 \urcorner = \ulcorner 6 \urcorner$$

is formally derivable, it follows from the Poincaré principle that the same is true for

$$F \ulcorner 45 \urcorner = \ulcorner 6 \urcorner$$

(with the same proof object), since $F\ulcorner 45 \urcorner \twoheadrightarrow_{\beta\iota} \ulcorner 6 \urcorner$. For a function computed by a term F, there is a proof requirement stating that this is done correctly. For example, in this case the requirement is

$$\forall n\;(F\,n)^2 \;\le\; n \;<((F\,n)+1)^2,$$

because that implies

$$\forall n\;(F\,n) \;\le \sqrt{n} \;\;<(F\,n)+1,$$

so

$$\forall n\;(F\,n) \;=\; [\sqrt{n}].$$

Such a proof requirement needs to be formally proved, but only once; after that, reductions like

$$F\ulcorner n \urcorner \twoheadrightarrow_{\beta\iota} \ulcorner[\sqrt{n}]\urcorner$$

can be used freely many times.

DEFINITION 3.2. Let $\boldsymbol{R} \subseteq \boldsymbol{A}^k$ be a k-ary relation on $\boldsymbol{A}$.

(i) $\boldsymbol{R}$ is called *definable* in Γ iff for some term k-ary Γ-relation R over $\boldsymbol{A}$ one has

$$\boldsymbol{R}(a_1, \ldots, a_k) \;\Leftrightarrow\; \Gamma \vdash R\, \underline{a_1} \ldots \underline{a_k}.$$

In this case we say that $\boldsymbol{R}$ is *defined* by R.

(ii) $\boldsymbol{R}$ is called *strongly definable* in Γ iff $\boldsymbol{R}$ is defined in Γ by R and moreover

$$\text{not } \boldsymbol{R}(a_1, \ldots, a_k) \;\Leftrightarrow\; \Gamma \vdash \neg R\, \underline{a_1} \ldots \underline{a_k}.$$

In this case we say that $\boldsymbol{R}$ is strongly defined by R.

DEFINITION 3.3. Let $\boldsymbol{R} \subseteq \boldsymbol{A}^k$ be a k-ary relation on $\boldsymbol{A}$. We say that $\boldsymbol{R}$ is λ-*computable* in Γ iff there is a k-ary Γ-relation $\widetilde{R}$ on $\boldsymbol{A}$ such that $\Gamma \vdash \widetilde{R} : A \rightarrow \mathtt{Bool}$ and one has

$$\boldsymbol{R}(a_1, \ldots, a_k) \;\Leftrightarrow\; \widetilde{R}\, \underline{a_1} \ldots \underline{a_k} =_{\beta_\iota} \mathsf{True}.$$

In this case we say that $\boldsymbol{R}$ is λ-*computed* by $\widetilde{R}$.

As a typical example we will show how formal proofs of statements like

$$\mathtt{Prime}(\underline{61})$$

or

$$\neg\mathtt{Prime}(\underline{60})$$

can be obtained in the context of Peano arithmetic. The method applies to arbitrary primitive recursive predicates.

One can construct a lambda-defining term $K_{\mathtt{Prime}}$ for the characteristic function of the predicate $\mathtt{Prime}$. This term should satisfy the following statement

$$\begin{aligned} \forall n\ \ [(&\mathtt{Prime}\, n \;\leftrightarrow\; K_{\mathtt{Prime}}\, n = \ulcorner 1 \urcorner)\ \& \\ &(K_{\mathtt{Prime}}\, n = \ulcorner 0 \urcorner \;\vee\; K_{\mathtt{Prime}}\, n = \ulcorner 1 \urcorner)], \end{aligned}$$

which is the proof requirement.

PROPOSITION 3.4. *Let $\boldsymbol{R}$ be a relation on $\boldsymbol{A}$. If $\boldsymbol{R}$ is λ-computable, then $\boldsymbol{R}$ is definable.*

Proof. Suppose that $\boldsymbol{R}$ is λ-computed by $\tilde{R}$. Then

$$\boldsymbol{R}(a_1, \ldots, a_k) \Leftrightarrow \vdash \tilde{R}\,\underline{a}_1 \ldots \underline{a}_k = \underline{\mathsf{True}}.$$

Hence $\boldsymbol{R}$ is represented by $R \equiv (\lambda x_1 \ldots x_k . \tilde{R}\,\underline{x}_1 \ldots \underline{x}_k = \underline{\mathsf{True}})$. □

Given a λ-computable relation $\boldsymbol{R}$, its defining term $\tilde{R}$ is less canonical than its representative R. We mean that a notion like being a prime number is fixed for ages, but the way to test primality depends on mathematical progress. More explicitly, while the predicate `Prime` is defined once and for all by

$$\mathtt{Prime}(x) \Leftrightarrow x > 1 \;\&\; [\forall y < x . y \mid x \Rightarrow y = 1],$$

possible algorithms $K_{\mathtt{Prime}}$ have undergone substantial mathematical improvements. This intensional aspect will be seen in the following definition.

DEFINITION 3.5. Let $\boldsymbol{R}$ be a definable relation on $\boldsymbol{A}$ with representing term R. We say that $\boldsymbol{R}$ is *provably λ-computable* iff there is a term $\tilde{R}$ that λ-defines $\boldsymbol{R}$ and

$$\vdash \forall x_1 \ldots x_k\; [R\, x_1 \ldots x_k \leftrightarrow \tilde{R}\, x_1 \ldots x_k = \underline{\mathsf{True}}].$$

Moreover, in this case we say that $\boldsymbol{R}$ is $R, \tilde{R}$ provably λ-computable. Most relations $\boldsymbol{R}$ come with a canonical representing term R. We will usually take this Γ-relation R as our starting point. In the above situation we say that R is provably λ-computable by $\tilde{R}$.

Likewise, for sets we often start directly from a syntactical representation A (a Γ-set) and leave the underlying set $\boldsymbol{A}$ implicit.

DEFINITION 3.6. Let A be a Γ-set, and let R be an n-ary predicate on A (in Γ).

(i) We say that R has a *proof generator* (in context Γ) iff there is some pseudoterm $T(x_1, \ldots, x_n)$ such that for all $\vec{a} \in A$ one has

$$\boldsymbol{R}(\vec{a}) \Leftrightarrow \Gamma \vdash T(\underline{\vec{a}}) : R(\underline{\vec{a}}).$$

(ii) R has a *strong proof generator* (in context Γ) iff both R and $\neg R$ have a proof generator (in context Γ). The needed pseudoterms may be different.

PROPOSITION 3.7. *Let A be a set in Γ. Let R be an n-ary predicate on A that is provably λ-computable by $\tilde{R}$ in context Γ. Then R has a strong proof generator in context Γ.*

Proof. We know that for some p_0 one has

$$\Gamma \vdash p_0 : \forall \vec{x}{:}A\, [R(\vec{x}) \leftrightarrow \tilde{R}(\vec{x}) = \underline{\mathsf{True}}].$$

Assume $R(\vec{a})$. Then $\tilde{R}\underline{\vec{a}} =_{\beta\iota} \underline{\mathsf{True}}$. We have

$$\Gamma \vdash p_0\underline{\vec{a}} : [R(\underline{\vec{a}}) \leftrightarrow \tilde{R}(\underline{\vec{a}}) = \underline{\mathsf{True}}],$$

hence

$$\Gamma \vdash \texttt{snd}(p_0\underline{\vec{a}}) : [R(\underline{\vec{a}}) \leftarrow \tilde{R}(\underline{\vec{a}}) = \underline{\mathsf{True}}].$$

Now

$$\Gamma \vdash \texttt{refl} : \forall x{:}\mathsf{Boole}.x = x,$$
$$\Gamma \vdash \texttt{refl}\ \underline{\mathsf{True}} : \underline{\mathsf{True}} = \underline{\mathsf{True}}.$$

Hence by the Poincaré principle and the fact that $\tilde{R}\underline{\vec{a}} =_{\beta\iota} \underline{\mathsf{True}}$ we have

$$\Gamma \vdash \texttt{refl}\ \underline{\mathsf{True}} : \tilde{R}\underline{\vec{a}} = \underline{\mathsf{True}},$$
$$\Gamma \vdash \texttt{snd}(p_0\underline{\vec{a}})(\texttt{refl}\underline{\mathsf{True}}) : R(\underline{\vec{a}})$$

We have now proved for appropriate T that for all $\vec{a} \in A$

$$\boldsymbol{R}(\vec{a}) \ \Rightarrow \ \Gamma \vdash T(\underline{\vec{a}}) : R(\underline{\vec{a}}).$$

The proof that for appropriate T' one has for all $\vec{a} \in A$

$$\text{not } \boldsymbol{R}(\vec{a}) \ \Rightarrow \ \Gamma \vdash T'(\underline{\vec{a}}) : \neg R(\underline{\vec{a}})$$

is similar.

To prove

$$\Gamma \vdash T(\underline{\vec{a}}) : R(\underline{\vec{a}}) \Rightarrow \boldsymbol{R}(\vec{a}),$$

we note that by the Generation lemma one has

$$\Gamma \vdash \texttt{refl}\ \underline{\mathsf{True}} : P = \underline{\mathsf{True}} \qquad \Leftrightarrow \qquad P =_{\beta\iota} \underline{\mathsf{True}}.$$

Suppose not $\boldsymbol{R}(\vec{a})$. Then $\tilde{R}\underline{\vec{a}} =_{\beta\iota} \underline{\mathsf{False}}$, so by the Church–Rosser property $\tilde{R}\underline{\vec{a}} \neq_{\beta\iota} \underline{\mathsf{True}}$, so $\Gamma \nvdash \texttt{refl}\ \underline{\mathsf{True}} : \tilde{R}\underline{\vec{a}} = \underline{\mathsf{True}}$. Hence (from the Generation lemma again)

$$\Gamma \nvdash \texttt{snd}(p_0\underline{\vec{a}})(\texttt{refl}\underline{\mathsf{True}}) : R(\underline{\vec{a}}).$$

The proof for the negative case is similar. □

Oostdijk (1996) presents a program that, for each primitive recursive predicate P, constructs the lambda-defining term K_P of its characteristic function and the proof of the adequacy of K_P. The resulting computations for $P = \texttt{Prime}$ are not efficient, because a straightforward (nonoptimized) translation of primitive recursion is given and the numerals (represented numbers) used are in a unary (rather than n-ary) representation; but the method is promising. Elbers (1996) gives a more efficient ad hoc lambda definition of the characteristic function of `Prime`, using Fermat's little theorem about primality. Also, the required proof requirement has been given.

4. Autarkic Two-Level Computations

First we show that certain binary relations have a proof generator. The method then applies to convertibility relations on terms in a complete term-rewriting system.

DEFINITION 4.1. Let R be a Γ-relation on A.

(i) R is *provably symmetric* if

$$\Gamma \vdash \Pi x,y{:}A.\ Rxy \to Ryx.$$

(ii) R is *provably transitive* if

$$\Gamma \vdash \Pi x,y,z{:}A.\ Rxy \to Ryz \to Rxz.$$

DEFINITION 4.2. Let $\boldsymbol{\varphi} : \boldsymbol{A} \to \boldsymbol{A}$.

(i) $\boldsymbol{\varphi}$ is an *indicator* for $\boldsymbol{R}$ if for all $a, b \in \boldsymbol{A}$

$$\boldsymbol{R}(a,b) \;\Leftrightarrow\; \boldsymbol{\varphi}(a) = \boldsymbol{\varphi}(b).$$

(ii) $\boldsymbol{\varphi}$ is said to be a *selector* for $\boldsymbol{R}$ if $\boldsymbol{\varphi}$ is an indicator for $\boldsymbol{R}$ and moreover for all $a \in \boldsymbol{A}$

$$\boldsymbol{R}(a, \boldsymbol{\varphi}(a)).$$

In what follows, $\boldsymbol{R}$ is represented by R in Γ.

DEFINITION 4.3. *R has a selector* if there exists a λ-computable indicator $\boldsymbol{\varphi}$ for $\boldsymbol{R}$ such that

$$\Gamma \vdash \Pi x{:}A.\ Rx(\varphi x).$$

THEOREM 4.4. *Let R be provably symmetric and transitive. If R has a selector, then there exists a proof generator for R.*

Proof. Let $\boldsymbol{\varphi}$ be a λ-computable indicator for $\boldsymbol{R}$ with $\Gamma \vdash \Pi x : A.\ Rx(\varphi x)$. Let $a, b \in \boldsymbol{A}$. Then one has

$$\begin{aligned} \boldsymbol{R}(a,b) &\;\Leftrightarrow\; \boldsymbol{\varphi}(a) = \boldsymbol{\varphi}(b) \\ &\;\Leftrightarrow\; \underline{\boldsymbol{\varphi}(a)} =_{\beta\iota} \underline{\boldsymbol{\varphi}(b)} \\ &\;\Leftrightarrow\; \varphi\,\underline{a} =_{\beta\iota} \varphi\,\underline{b}. \end{aligned}$$

Moreover, note that

$$\Gamma \vdash R\,\underline{a}(\varphi\,\underline{a}) \tag{2}$$

and

$$\Gamma \vdash R(\varphi\,\underline{b})\underline{b}. \tag{3}$$

(For (2) use symmetry of R.) Now suppose $\boldsymbol{R}(a,b)$. Then $R\,\underline{a}(\varphi\,\underline{a}) = \beta\iota R\,\underline{a}(\varphi\,\underline{b})$, so $\Gamma \vdash R\,\underline{a}(\varphi\,\underline{b})$ by (1) and the equality rule. Therefore $\Gamma \vdash R\,\underline{a}\,\underline{b}$ from (2) and transitivity. The proof object corresponding to the above argument is

$$\begin{aligned} &\mathrm{Trans}_R\ \underline{a}(\varphi\,\underline{b}) \\ &\qquad (\mathrm{Ind}\,\underline{a}) \\ &\qquad (\mathrm{Symm}_R\ \underline{b}(\varphi\,\underline{b})(\mathrm{Ind}\,\underline{b})) \\ &\equiv T[\underline{a}, \underline{b}] \quad \text{for short,} \end{aligned}$$

where Symm_R, Trans_R are the proof objects corresponding to symmetry and transitivity of R, and Ind is an inhabitant of $\Pi x : A.\, Rx(\varphi x)$. Now we have established

$$\boldsymbol{R}(a,b) \Rightarrow\ \vdash T[\underline{a},\underline{b}] : R\,\underline{a}\,\underline{b}.$$

Using the Generation lemma and the Church–Rosser property, one can show

$$\text{not } \boldsymbol{R}(a,b) \Rightarrow \Gamma \nvdash T[\underline{a},\underline{b}] : R\,\underline{a}\,\underline{b}$$

by an argument similar to the one in the proof of Proposition 3.7. □

The proof of a statement like $(x+1)(x+1) = x^2+2x+1$ corresponds to a symbolic computation. This computation takes place on the syntactic level of the formal terms. A function g acts on syntactic expressions satisfying

$$g(\,(x+1)(x+1)\,) = x^2+2x+1$$

that we want to lambda define. While $x+1 : \mathbb{N}$ (in context $x{:}\mathbb{N}$), the expression on a syntactic level represented internally satisfies $\ulcorner x+1 \urcorner : \mathtt{term}(\mathbb{N})$, for the suitably defined inductive type $\mathtt{term}(\mathbb{N})$. After introducing a reduction relation $\twoheadrightarrow_\iota$ for primitive recursion over this data type, one can use techniques similar to those of Section 3 in order to lambda define g by, say, G so that

$$G\ulcorner(x+1)(x+1)\urcorner \twoheadrightarrow_{\beta\iota} \ulcorner x^2+2x+1\urcorner.$$

To finish the proof, one needs to construct a self-interpreter E such that for all expressions $p : \mathbb{N}$ one has

$$\mathsf{E}\ulcorner p\urcorner \twoheadrightarrow_{\beta\iota} p$$

and prove the proof requirement for G, which is

$$\forall t\mathtt{term}(\mathbb{N})\ \mathsf{E}(G\,t) = \mathsf{E}\,t.$$

It follows that

$$\mathsf{E}(G\ulcorner(x+1)(x+1)\urcorner) = \mathsf{E}\ulcorner(x+1)(x+1)\urcorner.$$

Now since

$$\begin{aligned}
\mathsf{E}(G\ulcorner(x+1)(x+1)\urcorner) &\twoheadrightarrow_{\beta\iota} \mathsf{E}\ulcorner x^2+2x+1\urcorner\\
&\twoheadrightarrow_{\beta\iota} x^2+2x+1\\
\mathsf{E}\ulcorner(x+1)(x+1)\urcorner &\twoheadrightarrow_{\beta\iota} (x+1)(x+1),
\end{aligned}$$

we have by the Poincaré principle

$$(x+1)(x+1) = x^2+2x+1.$$

This method can be applied to many algebraic manipulations.

DEFINITION 4.5. Let Σ be a many-sorted signature, and let Γ be a context. Then Γ *extends* Σ (notation $\Gamma \supseteq \Sigma$) if Γ contains declarations of sorts $A : \star$ and operations $f : \vec{A} \to A$ according to Σ, and besides these the only declarations of variables in types $\Pi\vec{z} : \vec{B}.A$ (with A a sort in Σ) are of the form $x : A$.

For simplicity, we will focus on signatures with one sort A.

DEFINITION 4.6. Let $\Gamma \supseteq \Sigma$ be a context. The set of Γ-*elements* of A is defined by

$$A(\Gamma) = \{a \in \mathrm{NF} \mid \Gamma \vdash a : A\}.$$

The idea is to consider the elements of $A(\Gamma)$ explicitly as syntactic entities. This involves *representing* $A(\Gamma)$ in Γ, such that syntactic operations like unravelling of terms in the signature of rings

$$\mathrm{left}(a + b) = a,$$
$$\mathrm{right}(a + b) = b$$

are λ-*computable*. Note that representing $a \in A(\Gamma)$ by a itself does not work. Instead, we will translate the inductive definition of Σ-terms into type theory.

We focus on syntactic expressions in one variable $x : A$. This facilitates the representation; the method can easily be extended to expressions over more than one variable. In the case of rings, this would naturally lead to the formalization of multivariate polynomials.

DEFINITION 4.7. (i) The inductive type of A-*terms* (notation TERM_A) is generated by the constructors

$$\mathsf{X} : \mathsf{TERM}_A$$

and for each operation f in Σ, say, $f : A \to A \to A$,

$$\mathsf{f} : \mathsf{TERM}_A \to \mathsf{TERM}_A \to \mathsf{TERM}_A.$$

(ii) For each context $\Gamma \supseteq \Sigma$ we set

$$\mathsf{TERM}_A(\Gamma) = \{t \mid \Gamma \vdash t : \mathsf{TERM}_A\}.$$

Now we can translate $A(\Gamma)$ into $\mathsf{TERM}_A(\Gamma)$ and vice versa.

DEFINITION 4.8. (i) The function Quote : $A(\Gamma) \to \mathsf{TERM}_A(\Gamma)$ is defined by

$$\mathrm{Quote}(x) \equiv \mathsf{X},$$
$$\mathrm{Quote}(f\vec{a}) \equiv \mathsf{f}\,\mathrm{Quote}(\vec{a}).$$

We usually abbreviate Quote(a) by $\ulcorner a \urcorner$.

(ii) The *interpretation function* $\mathrm{E} : \mathsf{TERM}_A(\Gamma) \to A(\Gamma)$ is defined inductively by

$$\mathrm{E}(\mathsf{X}) \equiv x,$$
$$\mathrm{E}(\mathsf{f}\,\vec{t}) \equiv f\,\mathrm{E}(\vec{t}).$$

Now we can represent the set $A(\Gamma)$ as the type TERM_A by setting

$$\underline{a} \equiv \ulcorner a \urcorner : \mathsf{TERM}_A$$

for each $a \in A(\Gamma)$. Note that this choice indeed satisfies the requirements in Definition ref.

Remark 4.9. The operations left and right are λ-computable w.r.t. the representations $\underline{a}$.

LEMMA 4.10. *The interpretation function* E *can be represented in* λ*-calculus: there is a term* $\underline{\mathrm{E}}$ *such that*

$$\vdash \underline{\mathrm{E}} : \mathsf{TERM}_A \rightarrow A$$

such that for all $a \in A(\Gamma)$

$$\underline{\mathrm{E}}\,\underline{a} =_{\beta_\iota} a.$$

Proof. Easy, following the inductive specification of E using the recursor on TERM_A. □

In some cases one can take advantage of the fact that operations of a syntactic nature are λ-computable w.r.t. $\underline{t}$.

DEFINITION 4.11. Let R be a Γ-relation on A.

(i) The *syntactic variant* of R (notation $\widehat{R}$) is defined by

$$\widehat{R} \equiv \lambda s, t : \mathsf{TERM}_A.\, R(\underline{\mathrm{E}}\,s)(\underline{\mathrm{E}}\,t).$$

(ii) R is said to have a *syntactic proof generator* if $\widehat{R}$ has a proof generator.

Note that if T is a proof generator for $\widehat{R}$, then for all $a, b \in A(\Gamma)$

$$\Gamma \vdash R\,a\,b \;\Leftrightarrow\; \Gamma \vdash T[\ulcorner a \urcorner \ulcorner b \urcorner] : R\,a\,b.$$

This can be used for relations R with a Knuth–Bendix-like characterization. Consider, for example, the equality relation on rings (signature: sort A and operations $+$, $\cdot$, 0, 1). Since this equality can be characterized by an effective normalization procedure (rewriting $1 + 0$ to 1, for example) the syntactic variant of equality has an indicator, say, Normalize:

$$\Gamma \vdash a = b \;\Leftrightarrow\; \text{Normalize}(a) \equiv \text{Normalize}(b).$$

This is even provably so, since the function Normalize can be λ-computed and proven correct. Hence $\widehat{=}$ has a proof generator, say, T.

Now we can handle statements like

$$\Gamma \vdash \Pi x : A.\, x^2{+}2x{+}1 = (x{+}1)(x{+}1)$$

by checking the validity of

$$\Gamma, x{:}A \vdash T[\ulcorner x^2{+}2x{+}1 \urcorner, \ulcorner (x{+}1)(x{+}1) \urcorner] : x^2{+}2x{+}1 = (x{+}1)(x{+}1).$$

5. Conclusion

At this point one has not yet reached the goal of being able to formalize mathematics with an effort of the same order of magnitude as writing it in, say, LaTeX. To approach this goal, one needs to combine the tools of computer algebra (CA – for efficient computations) and proof development and verification (PDV – for reliable statements).

A pragmatic approach is to construct an interface between a system for CA and one for PDV. This approach is followed in systems such as Isabelle, HOL, and PVS. These systems provide a convenient way to represent logic and computations. For the verification of software, particularly in the case of protocols that are relatively small, this approach definitely increases reliability, compared with verification by hand. However, the resulting hybrid systems do not satisfy the "de Bruijn criterion" of providing statements verifiable by a small program (that can be checked by hand). As a matter of fact, some systems for CA contain bugs or are not always careful with side conditions for the validity of an equation.

A higher degree of reliability will be obtained by integrating systems for CA and PDV into one system for computer mathematics with proof objects verifiable by a small program that can be checked by hand. This can be achieved in at least two ways: beginning with a system for PDV or with a system for CA.

In the first method, one can start with a system for PDV, such as Coq or LEGO, specify computable functions used in systems for CA, and implement these with the necessary proof objects that show that the implementations are correctly constructed. This is the approach of the present paper. As with many algorithms of CA implemented as a term rewrite system, the work to be done is as follows. Suppose one has functions f_1, f_2, say, from a set P of polynomials to P. In a CA system the values $f_1(p)$, $f_2(p)$ are obtained as follows.

$$\boxed{\begin{array}{l} p \to_1 \ldots \to_1 p^{1-nf} \equiv f_1(p); \\ p \to_2 \ldots \to_2 p^{2-nf} \equiv f_2(p). \end{array}}$$

Here $\to_1$, $\to_2$ are certain notions of reduction on P computing the functions f_1, f_2, respectively. This is to be replaced by

$$\boxed{\begin{array}{l} F_1 p \to_{\beta\delta\iota} \ldots \to_{\beta\delta\iota} f_1(p); \\ F_2 p \to_{\beta\delta\iota} \ldots \to_{\beta\delta\iota} f_2(p). \end{array}}$$

If the $\to_1$, $\to_2$ are seen as special-purpose machines, then the transition to $\to_{\beta\delta\iota}$ is analogous to the transition from special-purpose machines to computing in a universal machine using software (the F_1, F_2).

The proof requirements consist of specifications for f_1, f_2 in the form of

$$\forall p.S_1(p, F_1 p); \forall p.S_2(p, F_2 p).$$

The analogy to a universal machine is not perfect because not all computable functions can be represented by $\beta\delta\iota$-reduction.[⋆]

The other way to obtain an integrated system for CM is to start with a system for CA and then to extend it with PDV tools. This method is proposed by Buchberger in his Theorema project. Moreover, in this project use is made of the Poincaré principle: "If an algorithm is proved correct, then it may be used many times in proofs to yield locally correct computations, without having to verify each instance."

Basically the two methods are complementary and will result in more or less equivalent systems for CM. We prefer the first method starting from a system for PDV, because it is carried out in a formal system and it is clear what work needs to be done. The approach starting with a system for CA probably needs more work, since the implemented algorithms have to be partly redesigned in order to make them fit into some type system.

The technology of CM is sometimes criticized as follows: By Gödel's incompleteness theorem there is no formal system that is strong enough to formalize all of mathematics. Therefore a system for CM based on, say, some pure type system like the calculus of constructions seems too limited.

As a reply we envisage a system for CM with two parameters[⋆⋆] that determine the logical and computational strength, respectively. The first parameter is the specification A of the pure type system in which one works. For example, one can set this parameter to PRED (first-order predicate logic), PRED2 (second-order predicate logic), PREDω (higher-order predicate logic), or many other systems; see Barendregt (1992). The second parameter determines the notion of reduction R for which the conversion rule (and therby Poincaré's principle) is valid. For example, it can be set to $\beta\delta$, $\beta\delta\iota$, or $\beta\delta\iota Y$. Another possibility is to add primitive numerals with a reduction relation A determined by the tables of addition and multiplication. Indeed, arithmetic operations are done more efficiently on the ALU (algorithmic logic unit) of a CPU than via defined lambda term numerals (even if these are represented as, say, binary numbers).

[⋆] For this reason one may, as in ML, want to add a fixed-point operator Y of type $\forall\alpha.(\alpha\rightarrow\alpha)\rightarrow\alpha$ and the reduction behavior

$$Yf \rightarrow_Y f(Yf).$$

The proof objects then become candidate proof objects that are reducing, and as soon as the Y have disappeared, a real proof object is obtained. Besides making the language universal for computations, the use of Y has the advantage that certain search procedures (say, for numbers) may be used in proofs. For example, this seems useful for a formalization of the proof of the four-color theorem.

[⋆⋆] Controlled by a "joystick."

Acknowledgments

The research in this paper has been supported by the Netherlands Computer Science Research Foundation (SION) with financial support of the Netherlands Organization for Scientific Research (NWO). We thank professor Martin Davis and the referee for improving the English text.

References

Aspers, W. A. M., de Bakker, J. W., ten Hagen, P. J. W., Hazewinkel, M., van der Houwen, P. J., and Nieland, H. M. (eds.): *Images of SMC Research 1996*, Stichting Mathematisch Centrum, Amsterdam, 1996.

Barendregt, H. P.: Lambda calculi with types, in S. Abramsky, D. M. Gabbay, and T. S. E. Maibaum (eds.), *Handbook of Logic in Computer Science*, Vol. 2, Oxford University Press, 1992, pp. 117–309.

Barendregt, H. P.: The quest for correctness, in Aspers et al., 1996, pp. 39–58.

Boyer, R. S. and Moore, J S.: *A Computational Logic*, Academic Press, New York, 1988.

de Bruijn, N. G.: The mathematical language AUTOMATH, its usage and some of its extensions, in M. Laudet, D. Lacombe, and M. Schuetzenberger (eds.), *Symposium on Automatic Demonstration*, INRIA, Versailles, Lecture Notes in Comput. Sci. 125, Springer-Verlag, Berlin, 1970, pp. 29–61. Also in Nederpelt et al., 1994.

de Bruijn, N. G.: Reflections on Automath 1990. Also in Nederpelt et al., 1994, pp. 201–228.

Bundy, A. (ed.): *CADE-12*, Lecture Notes in Comput. Sci. 814, Springer-Verlag, Berlin, 1984.

Cohen, A. M.: Computers: (Ac)counting for mathematical proofs, in Aspers et al., 1996, pp. 15–38.

Constable, R. L.: *Implementing Mathematics with the Nuprl Proof Development System*, Prentice-Hall, Englewood Cliffs, New Jersey, 1986.

Coquand, T. and Huet, G.: The calculus of constructions, *Information and Computation* **76** (1988), 95–120. Available at URL : http://pauillac.inria.fr/coq/coq-eng.html.

Elbers, H.: Personal communication, 1996.

Luo, Z. and Pollack, R.: The LEGO proof development system: A user's manual, Technical Report ECS-LFCS-92-211, University of Edinburgh, 1992. LEGO system available via URL http://www.dcs.ed.ac.uk/packages/lego/.

McCarthy, J.: *LISP 1.5 Programmer's Manual*, MIT Press, Cambridge, MA, 1962.

Nederpelt, R. P., Geuvers, J. H., and de Vrijer, R. C. (eds.): *Selected Papers on Automath*, Studies in Logic 133, North-Holland, Amsterdam, 1994.

Oostdijk, M.: *Proof by Calculation*, Master's thesis 385, Universitaire School voor Informatica, University of Nijmegen, 1996.

Poincaré, H.: *La Science et l'Hypothèse*, Flammarion, Paris. 1902.

Ruys, M. P. J.: *Studies in Mechanical Verification of Mathematical Proofs*, Dissertation, University of Nijmegen, 1999.