Efficient proofs

Henk Barendregt
Nijmegen University
The Netherlands

- Borwein:

"Your obligations are high" (to developers of mathematical software)

"The 4CT has a non-traditional standard of rigor"

- Greuel:

"I only trust a machine that I built myself"

"Be aware of selling our soul to the devil of algebra [computing]"

- Joswig:

" ... it is often hard to verify whether a computer proof is correct or not"

- Cohen:

"All computational tasks (in number fields) are finished"

Babylonian: computing

Greek: proving

It took until the 19-th century until the two traditions fully came together

20-th century: a temporary separation

- Computer algebra: Babylonian
- Formalization of proofs: Greek

21-st century: synthesis is emerging

Computer Mathematics

Mathematical activity: defining, computing, proving

Mathematical assistant helps human user:

Representing arbitrary mathematical notions	(defining)
Manipulating these	(computing)
Proving results about them	(proving)

in an impeccable way

Eventually to assist humans to learn and develop mathematics

At present an interesting foundational problem

- Representing "computable" objects

  $\sqrt{2}$ becomes a symbol α

  $\alpha^2 - 2$ becomes 0 $\qquad$ $\alpha + 1$ cannot be simplified

- Representing "non-computable" objects

  Hilbert space H, again just a symbol

  $P(H) :=$ "*H is locally compact*" is not decidable

  But $\quad \vdash p :^1 P(H) \quad$ is decidable $\quad$ ($^1 p$ is a proof of $P(H)$)

But then we need formalized proofs

Aristotle

- The axiomatic method

objects	properties
primitive	axioms
defined	derived

defining   proving
computing

- The quest for logic

  try to chart reasoning

- Poof-checking vs theorem proving

ONTOLOGY (what objects do there exist?)

Classical mathematics (before the 19-th century)
only needed a few fixed spaces

Modern mathematics needs a wealth of new spaces
and ample energy is devoted to the construction of these

Set theory gives the illusion that all these spaces exist beforehand
but it has the virtue that it unifies all needed concepts in one framework

Type theory based on

- inductively defined data types with their
- recursively defined functions and closed under
- dependent products

is an interesting alternative

Inductive types (freely generated data types)

Natural numbers `nat`

```
nat := 0 | S(nat)
```

```
nat := 0:nat | S:nat→nat
```

Primitive recursion over `nat`: we postulate an f : nat,nat^k→nat such that

$$
\begin{aligned}
f(0, \vec{x}) &= g(\vec{x}) \\
f(S(n), \vec{x}) &= h(f(n, \vec{x}), n, \vec{x})
\end{aligned}
$$

for g : nat^k→nat, h : nat,nat,nat^k→nat.

For example

$$
\begin{aligned}
0 + x &= x & 0 * x &= 0 & 0! &= 1 \\
S(n) + x &= S(n + x) & S(n) * x &= (n * x) + x & (S(n))! &= n! * (n + 1)
\end{aligned}
$$

Other data type: (binary) trees.

```
tree := leaf:nat→tree | p:tree,tree→tree
```

Primitive recursion over trees: given g, h we postulate F such that

$$\begin{aligned} F(\text{leaf}(n), \vec{x}) &= g(n, \vec{x}) \\ F(p(t_1, t_2), \vec{x}) &= h(F(t_1, \vec{x}), F(t_2, \vec{x}), t_1, t_2) \end{aligned}$$

For example

$$\begin{aligned} \text{mirror}(\text{leaf}(n)) &= \text{leaf}(n) \\ \text{mirror}(p(t_1, t_2)) &= p(\text{mirror}(t_2), \text{mirror}(t_1)) \end{aligned}$$

No need to code such structures into numbers via the Chinese remainder theorem (Gödel)

Function space types

If A, B are types, then $A \rightarrow B$ is the type of functions from objects of type A into objects of type B.

$$\frac{a : A \quad f : (A \rightarrow B)}{(f\ a) : B} \quad \frac{f : (A \rightarrow B) \quad g : (B \rightarrow C)}{(g \circ f) : (A \rightarrow C)}$$

Dependent products

$$\frac{\Gamma, n{:}A \vdash B(n) : \mathit{type}}{\vdash \Pi_{n:A}.B(n) : \mathit{type}}$$

Functional abstraction

$$\lambda x.f(x)$$

stands for the function $x \longmapsto f(x)$. For example, $g \circ f = \lambda x.g(f(x))$

First order: rules for $\to$, & , $\vee$, $\neg$, $\Leftrightarrow$, $\forall x{\in}U$, $\exists x{\in}U$

Continuity: $\forall\epsilon{>}0\forall x\exists\delta\forall y.\ldots$, uniform continuity: $\forall\epsilon{>}0\exists\delta\forall x,y.\ldots$

Second-order: rules for $\forall X{\subseteq}U$, $\exists X{\subseteq}U$

An element x in a group G has torsion iff $\exists n{\in}\mathbb{N}.x^n = e$

This definition is not allowed in pure first order logic.

In second-order logic:

$$\forall X{\subseteq}G[x{\in}X \;\&\; (\forall y{\in}X.xy{\in}X) \;\Rightarrow\; e{\in}X]$$

Higher-order statements

$\mathcal{O}$ is a topology on X

There exists a topology on U such that F is continuous

Brouwer: Aristotelian logic is unreliable

It may promise existence without being able to give a witness

$$\vdash \exists n{\in}\mathbb{N}.P(n), \text{ but } \not\vdash P(0), \not\vdash P(1), \ldots$$

Heyting: charted Brouwer's logic

Gentzen: gave it a nice form

Example of such a P

$$P(n) \Leftrightarrow (n = 0 \ \& \ \mathrm{GRH}) \vee (n = 1 \ \& \ \neg\mathrm{GRH})$$

Logic: "Intuitionism has become technology" (Constable)

THEOREM-CLASSICAL [No effectiveness]
For every ideal $I = (p_1, \ldots, p_n)$ *in* $\mathbb{Q}[\vec{x}]$ *there is a Gröbner basis* P *of* I*.*

THEOREM-CLASSICAL [This does not give the theorem]
There is a Turing machine TM such that for every ideal $I = (p_1, \ldots, p_n)$ *in* $\mathbb{Q}[\vec{x}]$ *the result* $TM(p_1, \ldots, p_n) = P$ *is a Gröbner basis of* I*.*

THEOREM-CLASSICAL [We do not always want to be explicit]
Let $\mathrm{TM} = \langle\langle q_0, ..\rangle, \ldots\rangle$*. Then TM is a Turing machine and for every ideal* $I = (p_1, \ldots, p_n)$ *in* $\mathbb{Q}[\vec{x}]$ *the result* $TM(p_1, \ldots, p_n) = P$ *is a Gröbner basis of* I*.*

THEOREM-CONSTRUCTIVELY. [Effectiveness]
For every ideal $I = (p_1, \ldots, p_n)$ *in* $\mathbb{Q}[\vec{x}]$ *there is a Gröbner basis* P *of* I*.*

Building an intuitionistic library provides certified tools

$\Gamma \vdash A$ means: in context Γ one can derive A.

$$\frac{}{\Gamma \vdash A}\; A{\in}\Gamma$$

$$\frac{\Gamma \vdash A \quad \Gamma \vdash A{\to}B}{\Gamma \vdash B} \qquad \frac{\Gamma, A \vdash B}{\Gamma \vdash A{\to}B}$$

$$\frac{\Gamma \vdash A}{\Gamma \vdash \forall x.A}\; x \notin \Gamma \qquad \frac{\Gamma \vdash \forall x.A}{\Gamma \vdash A[x := t]}\; t \text{ is free in } A$$

A proof of $A{\to}B$ is an algorithm transforming proofs of A into those of B

A proof of $\forall x{:}D.A(x)$ is an algorithm transforming a $d{:}D$ into a proof of $A(d)$

Logic: propositions-as-types, proofs as lambda terms

$\Gamma \vdash p : A$ means: in context Γ one can derive A with proof p

$$\frac{}{\Gamma \vdash x : A} \; x{:}A \in \Gamma$$

$$\frac{\Gamma \vdash p : A \quad \Gamma \vdash q : A \to B}{\Gamma \vdash (q\ p) : B} \qquad \frac{\Gamma, x{:}A \vdash p : B}{\Gamma \vdash (\lambda x{:}A.p) : A \to B}$$

$$\frac{\Gamma \vdash p : A}{\Gamma \vdash (\lambda x{:}D.p) : \forall x{:}D.A}\; x \notin \Gamma \qquad \frac{\Gamma \vdash p : \forall x.A}{\Gamma \vdash (p\ t) : A[x := t]}\; t \text{ is free in } A$$

$\Gamma \vdash_L p : A$ is decidable,

where L is first, second or higher-order logic.

Proof-assistants: formalized proofs are difficult to obtain.

Assistance

Reliability?

The de Bruijn criterion: have a small checker.

A romantic proof

THEOREM. For all $d > 0$ and all $n \in \mathbb{N}$ there exist $q, r \in \mathbb{N}$ such that

$$d < n \;\&\; n = d * q + r.$$

PROOF. Given $0 < d \in \mathbb{N}$ write

$$P_d(n) := \exists q, r \in \mathbb{N}[r < d \;\&\; n = dq + r].$$

By applying course of value induction on n we show $\forall n \in \mathbb{N}.P(n)$. So let $n \in \mathbb{N}$ and assume

$$\forall m < n\ P(m) \qquad (ih)$$

in order to show $P(n)$. If $n < d$, we can take $q = 0, r = n$. If $n \geq d$, write $n' := n \dot{-} d$. Then $n' < n$ and hence by (ih)

$$P(n'). \qquad (H1)$$

Hence for some $q', r' \in \mathbb{N}$

$$n' = d * q' + r' \;\&\; r' < d. \qquad (H2)$$

Now take $q = q' + 1$ and $r = r'$. Then $r < d$ and $d * q + r = d * q' + r + d = n' + d = n$, so we are done. ■

A cool proof (*proof-object*)

PROPOSITION. $\forall d{:}\mathbb{N}[0 < d \;\Rightarrow\; \exists q, r{:}\mathbb{N}.(r < d) \,\&\, n = d * q + r]$

```
Proof.
[d:nnat; p:(lthan zero d)]
 [P:=[n:nnat]
      (EX q:nnat | (EX r:nnat | (lthan r d)/\n=(plus (times d q) r)))]
  (cv_ind P
    [n:nnat; ih:(before n P)]
     [H:=(ltgeq n d)]
      (or_ind (lthan n d) (geq n d)
        (EX q:nnat | (EX r:nnat | (lthan r d)/\n=(plus (times d q) r)))
        [H0:(lthan n d)]
         (ex_intro nnat
           [q:nnat](EX r:nnat | (lthan r d)/\n=(plus (times d q) r))
           zero
           (ex_intro nnat
             [r:nnat](lthan r d)/\n=(plus (times d zero) r) n
             (conj (lthan n d) n=(plus (times d zero) n) H0
               (eq_ind_r nnat (times zero d) [n0:nnat]n=(plus n0 n)
                 (req nnat n) (times d zero) (times_com d zero)))))
        [H0:(geq n d)]
         [n':=(monus n d)]
          [H1:=(ltm n d (leseq_trans one d n p H0) p)]
           [H2:=(ih n' H1)]
            (ex_ind nnat
              [q:nnat]
               (EX r:nnat | (lthan r d)/\nn=(plus (times d q) r))
              (EX q:nnat |
                (EX r:nnat | (lthan r d)/\n=(plus (times d q) r)))
              [q':nnat;
               H3:(EX r:nnat | (lthan r d)/\nn=(plus (times d q') r))]
               (ex_ind nnat
                 [r:nnat](lthan r d)/\n'=(plus (times d q') r)
                 (EX q:nnat | (EX r:nnat | (lthan r d)/\n=(plus (times d q) r)))
```

```
[r':nnat; H4:((lthan r' d)/\n'=(plus (times d q') r'))]
 (and_ind (lthan r' d) n'=(plus (times d q') r')
   (EX q:nnat | (EX r:nnat | (lthan r d)/\n=(plus (times d q) r)))
            [H5:(lthan r' d); H6:(n'=(plus (times d q') r'))]
              (ex_intro nnat
                 [q:nnat]
                    (EX r:nnat |
                     (lthan r d)/\n=(plus (times d q) r))
                      (suc q')
                       (ex_intro nnat
                         [r:nnat]
          (lthan r d)/\n=(plus (times d (suc q')) r) r'
             (conj (lthan r' d)
                n=(plus (times d (suc q')) r') H5
                    [H7:=(f_equal nnat nnat (plus d) n'
                       (plus (times d q') r') H6)]
          [H8:=(eq_ind_r nnat (plus (monus n d) d)
             [n0:nnat]n0=n (pdmon n d H0)
                      (plus d (monus n d))
                          (plus_com d (monus n d)))]
                             (eq_ind nnat (plus d n')
         [n0:nnat]n0=(plus (times d (suc q')) r')
            (eq_ind_r nnat
                   (plus d (plus (times d q') r'))
                      [n0:nnat]
                        n0=(plus (times d (suc q')) r')
                         (compute q' r' d) (plus d n') H7) n H8))))
                    H4) H3) H2) H)). QED
```

Coq: Arithmetic0

Coq: Course of value induction

Coq: Euclid

Facing problems: computations and intuition

How does one give formal proofs of

- Computations

$$(xy - x^2 + y^2)(x^3 - y^3 + z^3) = x^4y - xy^4 + xyz^3 - x^5 + x^2y^3 - x^2z^3 + y^2x^3 - y^5 + y^2z^3.$$

- Intuition

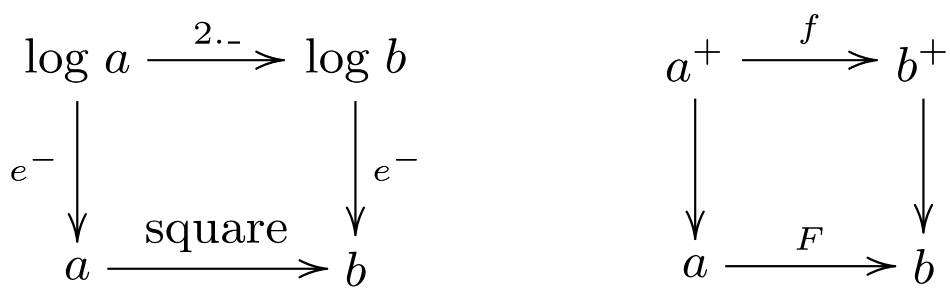
  Let $f : \mathbb{R} \to \mathbb{R}$ be defined by

$$f(x) = \frac{e^x + e^{-x}}{2} + e^{\sin^2 x} + e^{\cos^2 x}.$$

  Then f is continuous.

It is important to formally prove computations, not just for computational statements, but also for statements involving *intuition*

Facing problems: proving computational statements



Some computations f come for free: $f(a^+) = b^+$ is built into the system. This is the so-called Poincaré Principle:

If $f(a) = b$, via an external computation, then $a = b$ axiomatically.

The class $\mathcal{P}$ of f's for which this is postulated may vary, but usually contains the primitive recursive computations. Two extreme cases $\mathcal{P} = \emptyset$ and $\mathcal{P}$ is the class of all Computer Algebra definable maps do uccur.

Facing problems: Applying the Poincaré Principle

In our case the map + is not the logarithm but usually of a syntactic coding nature (reflection): $((x+y)^2)^+ = \texttt{sq}(\texttt{plus var}_\texttt{x}\ \texttt{var}_\texttt{y})$ and f is something like a **`simplify`** function on these syntactic expressions. The role of the exp function is played by the semantic function $[\![\ \ ]\!]$.

$$\begin{array}{ccc} \texttt{times(minus x y)(plus x y)} & \xrightarrow{\texttt{smpl.}} & \texttt{minus(sq x)(sq y)} \\ [\![\ ]\!]\downarrow & & \downarrow[\![\ ]\!] \\ (x-y)(x+y) & \xrightarrow{=_{\texttt{provably}}} & (x^2-y^2) \end{array}$$

In order to apply this freely one has to show

$$\forall e{:}L.[\![e]\!] = [\![\texttt{smpl e}]\!]$$

once and for all.

Facing problems: dealing with intuition

Goal to prove

$$A(t)$$

Pattern generalization

Strategy: write $t = f(s)$ with $s \in L$

and try to prove

$$\forall x \in L. A(f(x))$$

giving $A(f(s))$, hence $A(t)$.

This method is particularly powerful if combined with reflection.

Again we need to prove a computational equality $f(s) = t$.

CLAIM. *Let $g : \mathbb{R} \to \mathbb{R}$ be defined by*

$$g(x) = \frac{e^x + e^{-x}}{2} + e^{\sin^2 x} + e^{\cos^2 x}$$

Then g is continuous

PROOF. Use pattern generalization with

- A language L of expressions for continuous functions

- A valuation $[\![\ ]\!] : L \to (\mathbb{R} \to \mathbb{R})$

- $\forall t \in L. [\![t]\!]$ *is continuous*

- $[\![\ulcorner \lambda x. \frac{e^x + e^{-x}}{2} + e^{\sin^2 x} + e^{\cos^2 x} \urcorner]\!](x) = \frac{e^x + e^{-x}}{2} + e^{\sin^2 x} + e^{\cos^2 x}$

$A(g) :=$ 'g is continuous', $F(t) = [\![t]\!]$, $\forall t{:}L. A([\![t]\!])$ and $F(\ulcorner g \urcorner) = g$.

Facing problems: partial reflection

Proofs of computational equalities between rational expressions like

$$\forall x, y{\in}\mathbb{C}. x \neq y \;\&\; x \neq -y \;\Rightarrow\; \frac{1}{x+y} + \frac{1}{x-y} = \frac{2x}{x^2 - y^2}$$

are obtained by *partial reflection* and pattern generalization.

Formalized in Coq (intuitionistic proofs)

THEOREM 1.
[Formalization: Geuvers, Wiedijk, Zwanenburg, Pollack, Niqui]

Every non-constant polynomial $p(x)$ *over* $\mathbb{C}$ *has a* root x.

THEOREM 2. Collaboration between Coq and GAP
[Formalization: Oostdijk, Caprotti, Elbers]

The number 9026258083384996860449366072142307801963 *is a prime.*
(Based on little Fermat's theorem and Pocklington.)

THEOREM 3.
[Formalization: Capretta]

Correctness of the Fast Fourier Transform.

THEOREM 4.
[Formalization: Person, Théry]
Correctness of an efficient Gröbner base algorithm.

THEOREM 5.
[Formalization: Danos, Gonthier, Werner]
Main lemma for the four colour theorem.

For this, Coq needed an overdrive: compilation rather than interpretation

This compiler was proved correct in the simpler version of Coq

Cf.

• Human eye	Romantic proof
• Light microscope	Cool proof
• Electronic microscope	Hyper cool proof

Challenge: to formalize substantial parts of mathematics

Start with undergraduate mathematics. Needed

- Libraries: theories, algorithms.
- Tools for proving computations, intuitive steps.
- Interface: views, proof by clicking, library management.
- Make formalization as easy as writing down a proof in LaTeX, by developing a mathematical proof language (like Mizar has).
- International collaboration: Bologna database HELM.
- There is an *onto-logical* shift

  $\vdash_{\mathrm{PA}} \forall x.P(x)$

  $\vdash_{\lambda L} \forall x{:}\mathtt{nat}.P(x)$

  $\vdash_{\mathrm{ZF}} \forall x{\in}\omega.P(x)$

System	Underlying logic	Poincaré Principle
Mizar	ZF	$\emptyset$
HOL, Isabelle, Isar	HOL	$\emptyset$
[Automath]	λP	β
NuPrl	extensional TT	$\beta\delta\iota$
Agda	λP	$\beta\delta\iota$
Coq, Lego, Plastic	$\lambda P\omega$	$\beta\delta\iota$
ACL2, PVS	Prim. Rec. Arithmetic?	"Cas"

See ⟨`www.cs.kun./nl/~freek`⟩ for a longer (commented) list.

For references, see Barendregt, Cohen, Issac 2000,
Electronic Communication of Mathematics and the Interaction of Computer Algebras Systems and Proof Assistants,
Special Issue J. Symbolic Computing, 32 (2001), pp. 3–22.

Moral

The computer algebra community has done and is doing impressive work.

But the work will not be finished until the semantical content of mathematics

is fully encorporated.

There are modern functional programming languages with

- Every object is a typed function (no side effectes)
- Implementation is efficient
- Only meaningful combinations can be made
- Interaction can be programmed nicely with higher order functions
- The language has mobile code (dynamic linking)*

This makes it possible to bring down development time, during the design, debugging and maintenance. The mobile code makes possible distributed and parallel computing over the internet.

Clean (Nijmegen) *only in Clean
Haskell (Glasgow)