Introduction to Computer-Aided Proofs and LCF Approach

Viktor Kuncak, EPFL

`https://lara.epfl.ch/w/fv`

Hilbert's Program and The Origin of Formal Sciences

Is there an error in a complex mathematical proof?

How can we be sure there is none?

Hilbert's Program and The Origin of Formal Sciences

Is there an error in a complex mathematical proof?
How can we be sure there is none?

If we require proof to be a *formal proof*, this question can be checked by a computer, or any person that can follow precise instructions!

Precise axioms: Euclid. Inference rule: Frege, logicians around 1900s

The origin of this idea (as well as the use of the word "formal"), used in *formal methods* and *formal verification*, is **Hilbert's program**.

Attempts to make foundations of analysis precise using set theory faced paradoxes (e.g. for $S = \{x \mid x \notin x\}$, does $S \in S$?) that allow us to prove everything.
The idea of (W) David Hilbert was: define precisely axioms and inference rules about mathematics. Existence of proofs can be expressed using sequences of formulas, which can also be encoded in number theory (a sequence of bits is a natural number). The hope: use number theory to prove that other axioms do not lead to contradiction.

Proof in a Proof System $(\mathscr{F}, \mathrm{Infer})$ as a Statement about Numbers

Definition

Given $(\mathscr{F}, \mathrm{Infer})$ where $\mathrm{Infer} \subseteq \mathscr{F}^* \times \mathscr{F}$ a **proof** in $(\mathscr{F}, \mathrm{Infer})$ is a finite sequence of inference steps $S_0, \ldots, S_m \in \mathrm{Infer}$ such that, for each S_i where $0 \leq i \leq m$, for each premise P_j of S_i there exists $0 \leq k < i$ such that P_j is the conclusion of S_k.

$$\begin{array}{rll} S_0 : & ((), & C_0) \\ & \ldots & \\ S_k : & ((\ldots\ldots\ldots), & \mathbf{P_j}) \\ & \ldots & \\ S_i : & ((\ldots, \mathbf{P_j}, \ldots), & C_i) \end{array}$$

Proof in a Proof System $(\mathscr{F}, \mathsf{Infer})$ as a Statement about Numbers

Definition

Given $(\mathscr{F}, \mathsf{Infer})$ where $\mathsf{Infer} \subseteq \mathscr{F}^* \times \mathscr{F}$ a **proof** in $(\mathscr{F}, \mathsf{Infer})$ is a finite sequence of inference steps $S_0, \ldots, S_m \in \mathsf{Infer}$ such that, for each S_i where $0 \le i \le m$, for each premise P_j of S_i there exists $0 \le k < i$ such that P_j is the conclusion of S_k.

$$\left.\begin{array}{rll} S_0: & ((), & C_0) \\ & \ldots & \\ S_k: & ((\ldots\ldots\ldots), & \mathbf{P_j}) \\ & \ldots & \\ S_i: & ((\ldots, \mathbf{P_j}, \ldots), & C_i) \end{array}\right\} \text{sequence of bits, a very big natural number}$$

Peano arithmetic (a first-order logic theory of $+$ and $*$) can express statement that a number represents a proof in a given proof system: putting together symbols can be expressed using multiplication, addition, and other number-theoretic operations.

Incompleteness (Which is Overstated)

(W) Kurt Gödel showed in 1931 his incompleteness theorem, implying that this aspect of Hilbert's program does not work: a non-contradictory system that can express complex enough axioms is not even able to prove its own proof system as consistent, let alone a more complex proof system.
A part of Gödel's incompleteness theorem can be restated by saying that **provability** in a given axiom system is a recursively **enumerable** property, and performs diagonalization to show that there are theorems that are not provable, for analogous reason that there are enumerable problems that are not decidable. The proof is similar to the 1936 proof of (W) Alan Turing but is phrased in terms of arithmetic (Gödel numbering of formulas) and predates Turing's proof.

Certain sets (including sets of true formulas in arithmetic) are not enumerable. Any consistent proof system will under-approximate the truth using some enumerable set; there will be true arithmetic statements that do not have proofs in a given proof system ("incompleteness"). But, in practice, we have expressive enough systems for which we can prove the useful theorems. The hard part is finding proofs, not making axioms!

Computer-Checked Proofs and Automath

Logicians and mathematicians developed precise formal definitions and theorems for much of the useful mathematics. When computers came, efforts started to write programs that check formal proofs and search for them.
Automath project from 1960ies in Netherlands, led by (W) Nicolaas Govert de Bruijn, constructed a system for checking proofs (represented as lambda calculus terms).
`https://www.win.tue.nl/automath/`

In 1970ies a PhD thesis of L.S. van Benthem Jutting checked a famous textbook by Landau, which axiomatizes natural numbers then uses them to construct integer, rational, and real numbers (using Dedekind cuts).

Almost no automation - just checking of proofs.
Very dense proofs span hundreds of kilobytes per chapter.
`http://www.cs.ru.nl/~freek/aut/`

Edmund Landau: Grundlagen der Analysis

“Ich hoffe, nach **jahrzehntelanger Vorbereitung** diese Schrift so abgefaßt zu haben, daß **ein normaler Student sie in zwei Tagen lesen kann**. Und dann darf er sogar (da er die formalen Regeln ja schon von der Schule her kennt) den ganzen Inhalt bis auf das Induktionsaxiom und den *Dedekind*schen Hauptsatz vergessen.
Wenn aber gar dem einen oder anderen Kollegen der anderen Richtung die Sache so leicht erscheint, daß er sie in seinen Anfängervorlesungen (auf dem folgenden oder irgend einem anderen Wege) bringt, würde ich ein Ziel erreicht haben, auf das ich in größerem Umfange nicht zu hoffen wage.

Berlin, den 28. Dezember 1929.

Edmund Landau.”

Edmund Landau: Grundlagen der Analysis

Axiom 1: 1 *ist eine natürliche Zahl.*

D. h. unsere Menge ist nicht leer; sie enthält ein Ding, das 1 (sprich: Eins) heißt.

Axiom 2: *Zu jedem x gibt es genau eine natürliche Zahl, die der Nachfolger von x heißt und mit x' bezeichnet uerden möge.*

Bei komplizierten x wird die Zahl, um deren Nachfolger es sich handelt, eingeklammert, wenn sonst ein Mißverständnis zu befürchten ist. Entsprechendes gilt im ganzen Buch bei $x+y$, xy, $x-y$, $-x$, x^y u. dgl.

Aus

$$x = y$$

folgt also

$$x' = y'.$$

Axiom 3: *Stets ist*

$$x' \neq 1.$$

D. h. es gibt keine Zahl mit dem Nachfolger 1.

Dedekind Cuts (Reals as Sets of Rationals), Complex Numbers

Operations on natural numbers through successor function.
Integers as equivalence classes of pairs of natural numbers.
Rational numbers as equivalence classes of pairs of natural numbers.
Real numbers as certain sets of rational numbers.

...

Dedekindscher Hauptsatz.

Satz 205: *Gegeben sei irgend eine Einteilung aller reellen Zahlen in zwei Klassen mit folgenden Eigenschaften.*
1) *Es gibt eine Zahl der ersten Klasse und eine Zahl der zweiten Klasse.*
2) *Jede Zahl der ersten Klasse ist kleiner als jede Zahl der zweiten Klasse.*
Dann gibt es genau eine reelle Zahl Ξ*, so daß jedes* $\eta < \Xi$ *zur ersten, jedes* $\eta > \Xi$ *zur zweiten Klasse gehört.*

...

Complex numbers as equivalence classes of pairs of real numbers.

7k lines of latex. 220 KB $\rightarrow$ 68 KB gzipped

Formalization in Automath (From Last Chapter). PhD thesis of Jutting

```
@[r:real][s:real][t:real][n:nis(t,0)]
+v0
t1:=tr3is(real,ts(t,ts(r,ov(s,t,n))),ts(ts(r,ov(s,t,n)),t),ts(r,ts(ov(s
-v0
lemma6:=satz204g(ts(r,s),t,ts(r,ov(s,t,n)),n,t1".v0"):is(ts(r,ov(s,t,n)
+*v0
n@t2:=tris(real,ts(t,pl(ov(r,t,n),ov(s,t,n))),pl(ts(t,ov(r,t,n)),ts(t,o
-v0
n@lemma7:=satz204g(pl(r,s),t,pl(ov(r,t,n),ov(s,t,n)),n,t2".v0"):is(pl(o
r@[n:nis(r,0)]
lemma8:=satz204b(r,r,n,ov(r,r,n),1rl,satz204c(r,r,n),satz195(r)):is(ov(
lemma9:=ore2(is(r,0),is(ov(0,r,n),0),satz192c(r,ov(0,r,n),satz204c(0,r,
r@[i:is(r,m0(r))]
+*v0
i@[p:pos(m0(r))]
t3:=<isneg(r,m0(r),i,satz176f(r,p))>pnotn(m0(r),p):con
```

10k lines of formal proof. 750 KB text → 172 KB gzipped

Illustration of Foundational Construction: Integers

Let $\mathbb{N}$ be the set of natural numbers with zero.
Define relation $\sim$ on pairs $(a,b),(c,d) \in \mathbb{N}^2$ by

$$a+d=c+b$$

Intuition: pair (a,b) represents $a-b$ and (c,d) pair $c-d$, so the above corresponds to $a-b=c-d$, but we do not use $-$ to define it.
Theorem: the relation $\sim$ is reflexive, symmetric, transitive. We call equivalence classes $(a,b)_{/\sim}$ **integers** and denote their set by $\mathbb{Z}$.
An equivalence class of (a,b) with $a \geq b$ also contains the pair $(n,0)$ where $n=b-a$ and we denote this equivalence class by n. An equivalence class of (a,b) with $a \leq b$ also contains the pair $(0,n)$ where n is $b-a$ and we denote it by $-n$.

Operations and Congruence

If we define addition by $(a,b)+(c,d)=(a+c,b+d)$, then $\sim$ is a congruence relation with respect to $+$ (i.e. $\sim$ behaves like equality):

$$(a,b)\sim(a',b') \ \wedge \ (c,d)\sim(c',d') \ \ \rightarrow \ \ (a,b)+(c,d) \ \sim \ (a',b')+(c',d')$$

Thus, we can define an operation on equivalence classes such that

$$(a,b)_{/\sim}+(c,d)_{/\sim}=(a+c,b+d)_{/\sim}$$

Similarly, we can define

$$(a,b)\cdot(c,d)=(ac+bd,ad+bc)$$

for which $\sim$ is also a congruence.

Construction for rationals from integers is analogous.
Construction for real numbers requires sets of rational numbers and is more complex.
Construction of complex numbers is again easy.

LCF Approach: Proof System as an Abstract Data Type

LCF = Logic for Computable Functions

Logic for reasoning about computable (partial!) functions based on **domain theory** developed by (W) Dana Scott. A system for writing formal proofs about such functions.

Michael J. C. Gordon, Robin Milner, L. Morris, Malcolm C. Newey, Christopher P. Wadsworth: A Metalanguage for Interactive Proof in LCF. POPL 1978: 119-130

- Theorem is an abstract data type that stores a formula.
- We cannot create a theorem out of an arbitrary formula (constructor is private)
- The only ways to create formulas is to invoke public functions that:
  - create axiom instances given some parameters, or
  - take theorems as arguments and return new theorems (inference rules)

For any program (even one that we cannot prove to terminate), if it computes a value v:Theorem, then v is a theorem of the proof system.

Key Idea of Abstract Data Types

One of the key ideas in formal specification of programs.
A foundation of object-oriented programming constructs.

Claim: If a language is **type safe** and has **private constructors and fields**, then we can build data structure with operations that maintain **invariants** on the private state, without assuming anything about clients except that they are type checked.

Binary Search Tree Data Structure as an Abstract Data Type

```
sealed abstract class Tree
case class Leaf private() extends Tree
case class Node private(left: Tree, value: BigInt, right: Tree)
    extends Tree // users outside cannot invoke Node(t1,t2)
def emptySet(): Tree = Leaf() // public
def insert(tree: Tree, value: BigInt): Node = // public
  tree match {
    case Leaf() ⇒ Node(Leaf(), value, Leaf())
    case Node(l, v, r) ⇒ (if (v < value) {
      Node(l, v, insert(r, value))
    } else ...
  }

def insert(tree: Tree, value: BigInt): Boolean = // public
  // we can be sure that tree is sorted, regardless of client
    ...
```

A Propositional Proof System as an Abstract Data Type (LCF Approach)

```scala
final class Theorem private(val formula: Formula) {
  def modusPonens(pq: Theorem, p: Theorem): Theorem = pq.formula match{
    case Implies(pf, qf) if p.formula == pf ⇒ Theorem(qf)
    case _ ⇒ throw new Exception("Illegal use of modusPonens.")
  }
  /** Axiom `P ⇒ Q ⇒ P`. */
  def addImplies(p: Formula, q: Formula): Theorem =
    Theorem(Implies(p, Implies(q, p)))

  /** Axiom `(P ⇒ Q ⇒ R) ⇒ (P ⇒ Q) ⇒ P ⇒ R`. */
  def impliesDistr(p: Formula, q: Formula, r: Formula): Theorem =
    Theorem(Implies(Implies(p, Implies(q, r)),
                    Implies(Implies(p, q), Implies(p, r))))
  ...
}
```

Proof of $P \Rightarrow P$

```
// proof system: in this package we can build a theorem out of formula
 def modusPonens(pq: Theorem, p: Theorem): Theorem = pq.formula match{
   case Implies(pf, qf) if p.formula == pf ⇒ Theorem(qf) ... }
  /** Axiom `P ⇒ Q ⇒ P`. */
 def addImplies(p: Formula, q: Formula): Theorem =
   Theorem(Implies(p, Implies(q, p)))
  /** Axiom `(P ⇒ Q ⇒ R) ⇒ (P ⇒ Q) ⇒ P ⇒ R`. */
 def impliesDistr(p: Formula, q: Formula, r: Formula): Theorem =
   Theorem(Implies(Implies(p, Implies(q, r)),
                   Implies(Implies(p, q), Implies(p, r))))
} // end of package defining axioms and rules

// outside, Theorem is opaque and we can only invoke above functions
def impliesRefl(p: Formula): Theorem = // returns Theorem(Implies(p,p))
  modusPonens(impliesDistr(p, Implies(p,p), p),
              modusPonens(addImplies(p, Implies(p, p)), addImplies(p, p)))
```

Which Languages for Writing Proofs?

"Our present point of view is that neither a straightforward proof-checker (laborious and repetitive to use) nor an automatic theorem-prover (inefficient because of general search) is satisfactory. What is required is a framework in which a user can both design his own partial proof strategies (where he can find them) and execute single steps of proof (where he needs to)." Gordon et al (POPL'78)

We need:

Symbolic computation: manipulate theorem trees (at least as good as LISP)

Type safety: so we cannot cast Formula into Theorem. Support abstract data types.

- convenient to have type inference, so code is not much less reusable than LISP

Higher-order functions: help us produce tactics and tacticals that automate increasingly complex steps.

Exceptions: when you use a function in wrong way, throw exception as opposed to doing checks everywhere

Result: Greatest Step in Modern Programming Language Design

The resulting proposed language was called meta-language (ML) because it is a language for writing programs that compute theorems (as opposed to language of of LCF formulas).

Invented generic types.
Standardized into Standard ML and branched into CAML (categorical ML), then OCaml (its variants F# (Microsoft), Reason (Facebook)) and influenced Haskell.

Influenced Scala as well: Java went back to Scheme. M.Odersky and P. Wadler added generics; at EPFL Odersky created Scala.

Follow-Up LCF-Style Proof Assistants

M. Gordon was on the LCF paper and, with L. Paulson created their own variants in Cambridge (UK).
Partial functions of domain theory fell out of fashion:
"The need to deal with the associated "bottom" value (⊥) tended to clutter proofs. Mike thought it could go away temporarily. (It never came back.)"[1]
This led to the a family of HOL systems

- higher-order logic based on simply-typed lambda calculus
- modulo lambda calculus notation, dates back to (W) Bertrand Russel's Principia Mathematica (1910)

Simpler logic, but still using tactics in ML.
HOL4 is still actively used: recently, a formally proved a correctness of a compiler for ML, as well as just-in-time compilers (CakeML project).
Today, main ML implementation is Poly/ML in Cambdidge (UK) Computing Lab.

[1]Lawrence C Paulson FRS: *Michael John Caldwell Gordon (FRS 1994). 28 February 1948–22 August 2017*

John Harrison: HOL Light

John Harrison, `https://www.cl.cam.ac.uk/~jrh13/`, graduated from Cambridge and went to work in industry (Intel) where he developed **HOL Light** in ocaml and verified correctness of microprocessors (especially floating-point units). Now at Amazon.

Wrote an amazing book with ocaml code, see `https://www.cl.cam.ac.uk/~jrh13/atp/`

HOL Light was also used to e.g. verify (W) Kepler conjecture (packing oranges in supermarket), one of the most complex formal proofs ever completed.

Logical Frameworks

To construct proofs with rules directly in an LCF system, we have to either go forward, or add new rules to go backward. Instead, it may be convenient to have a language for rules. This leads us to logical frameworks.
Examples:

- LF, ELF: formulas are types, proofs are terms of those types
- Isabelle (formulas are formulas in higher-order minimal logic (meta-logic), inference rules are implications)

(W) Isabelle (proof assistant) was used to define many concrete logics, e.g.

- LCF
- FOL
- ZF set theory - mainstream foundation of mathematics
- HOL (like in the HOL system of HOL4 and HOL Light)

Lawrence Paulson: Isabelle Proof Assistant

Lawrence C. Paulson: Isabelle: the next 700 theorem provers. Logic and Computer Science, 361–386. 1990.

Use LCF-style framework to define inference rules for minimal higher-order logic (a variant of intuitionistic higher-order logic):

- implication (expresses inference rules)
- quantification (expresses bound variables)
- higher-order unification (takes care of equality and instantiation)
- the main inference rule is a higher-order variant of resolution

Define each concrete logic as a set of axioms in Isabelle's meta-logic
Meta implication $\Rightarrow$ denotes rules, object-level $\rightarrow$ denotes connective in formulas.
The most developed among the logics is Isabelle/HOL (similar to one in HOL4 system)
The group of Prof. Tobias Nipkow (TUM) has been the center of many practical development of Isabelle system in recent years.

More Topics

Review of syntax and model theoretic semantics of first-order logic.

Encoding in Scala of the proof Hilbert-style proof system as used in code by John Harrison to accompany the textbook “Practical Logic and Automated Reasoning”.

Presentation of (Fitch-style) natural deduction rules for (intuitionistic) first-order logic with equality from page 2 of “Old Introduction to Isabelle” by Lawrence Paulson, 9 June 2019.