Comparison of Theorem Provers

Artem Yushkovskiy

`artem.yushkovskiy@aalto.fi`

Tutor: Stavros Tripakis

Abstract

The need for formal definition of the very basis of mathematics arose in the last century. The scale and complexity of mathematics, along with discovered paradoxes, revealed the danger of accumulating errors across theories. Although, according to Gödel's incompleteness theorems, it is not possible to construct a single formal system which will describe all phenomena in the world, being complete and consistent at the same time, that crisis has considerably improved philosophical views on mathematics. In addition, it gave rise to rather practical areas of logic, s.a. the theory of automated theorem proving. This is a set of techniques used to verify mathematical statements mechanically using logical reasoning. Moreover, it can be used to solve complex engineering problems as well, for instance, to prove the security properties of a software system or an algorithm. This paper compares two widespread tools for automated theorem proving, Coq [1] and Isabelle/HOL [2], with respect to the power of expressiveness and usability. For this reason, it firstly gives a brief introduction to the bases of formal systems and automated deduction theory, its main problems and challenges.

KEYWORDS: logic, formal method, proof theory, automated theorem prover, Coq, Isabelle.

1 Introduction

Nowadays, the search for foundations of mathematics has become one of the key questions in philosophy of mathematics, which eventually has an impact on numerous problems in modern life. Basically, this search has led to the development of *formal approach*, a methodology for manipulating the abstract essences according basic rules in a verifiable way. In other words, it is possible to follow the sequence of such manipulations in order to check the validity of each statement and, as a result, of a system at whole. Moreover, automating such a verification process can significantly increase reliability of formal models and systems based on them.

At present, a large number of tools have been developed to automate this process. Generally, these tools can be divided into two broad classes.

The first class contains tools pursuing the aim of validating the input statement (*theorem*) with respect to the sequence of inference transitions (user-defined *proof*) according to set of inference rules. Such tools are sometimes called *proof assistants*, their purpose is to help users to develop new proofs. The tools *Isabelle* [2], *Coq* [1], *PVS* [3] are well-known examples of such systems, which are commonly used nowadays.

The second class consists of tools that automatically *discover* the formal proof, which can rely either on induction, on meta argument, or on higher-order logic. Such tools are often called *automated theorem provers* so that they apply techniques of automated reasoning to find the proof. The systems *Otter* [4] and *ACL2* [5] are commonly known examples of such tools.

In this paper only systems of the first class were considered in order to test the usability of such systems.

1.1 Related work

A considerably extensive survey on theorem provers has been presented by F. Wiedijk [6], where fifteen 'state-of-the-art' systems for the formalization of mathematics were compared against various properties, in particular size of library with already proved lemmas, strength and expressiveness of underlying logic, size of proofs (the de Bruijn criterion) and level of automation (the Poincaré principle). We propose more deep comparison of two aforementioned theorem provers – Coq and Isabelle – with respect to criteria, s.a. expressiveness, computation power and usability. Section 2 provides definition, basic properties and theoretical limitations of the formal systems, while section 3 presents target comparison properties and the comparison itself.

2 Foundations of Formal Approach

The formal approach appeared in the beginning of previous century when mathematics experienced deep fundamental crisis caused by the need for a formal definition of the very basis. At that time, multiple paradoxes in several fields of mathematics have been discovered. Moreover, the radically new theories appeared just by modification of the set of axioms, e.g., reducing the parallel postulate of Euclidean geometry has lead to completely different non-Euclidian geometries s.a. Lobachevsky's hyperbolic geometry or Riemman's elliptic geometry, that eventually have a large number of applications in both natural sciences and engineering.

2.1 Definition of the Formal System

Let the *judgement* be an arbitrary statement. The *formal proof* of the formula ϕ is a finite sequence of judgements $(\psi_i)_{i=1}^n$, where each ψ_i is either an axiom A_i, or a formula inferred from the subset $\{\psi_k\}_{k=1}^{i-1}$ of previously derived formulae according the *rules of inference*. An *axiom* $A_i \in A$ is a judgement evidently claimed to be true. *A logical inference* is a transfer from one judgement (*premise*) to another (*consequence*), which preserves truth. In formal logic, inference is based entirely on the structure of those judgements, thereby, the result formal system represents the abstract model describing part of real world.

As an example of inference rule, the widely used *Modus ponens* (MP) rule can be considered: $\frac{A,\ (A \to B)}{B}$. In this standard notation the premises are enumerated above the horizontal line and consequencess – below it. We will use this notation further.

The formulae consist of *propositional variables*, connected with *logical connectives* (or logical operators) according to rules, defined by a formal language. The formulae, which satisfy such rules, are called *well-formed formulae* (wff). Only wff can form judgements in formal system. The propositional variable is an atomic formula which can be either true or false. The logical connective is a symbol in formal language that transforms one wff to another. Typically, the set of logical connectives is $\Omega = \{\neg, \wedge, \vee, \to\}$.

The formal system described above does not contain any restriction on the form of propositional variables, such logic is called *propositional logic*. However, if these variables are quantified on the sets, such logic is called *first-order* or *predicate logic*. Commonly, first-order logic has two quantifiers, the universal quantifier $\forall$ (means "for every"), and the existential quantifier $\exists$ (means "there exists"). Thereafter, the *second-order logic* extends first-order logic by adding quantifiers over second-order objects – relations defining the sets of sets. In turn, it can be extended by the *higher-order logic*, which contain quantifiers over the arbitrary nested sets, or *type theory*, which assigns a type for every expression in the formal language.

Let Φ be a set of formulae. Initially, it consists of only *hypotheses*, a priori true formulae,

which are claimed to be already proved. The notation $\Phi \vdash \phi$ means that the formula ϕ is *provable* from Φ, if there exists a proof that infers ϕ from Φ. The formula which is provable without additional premises (i.e. $\emptyset \vdash \phi$) is called *tautology* and denoted as $\vdash \phi$. The formula is called *contradiction* if $\vdash \neg\phi$. Obviously, all contradictions are equivalent in one formal system, hence they are denoted as $\perp$.

The formal system described above is called *Hilbert proof system*, it uses reasoning from *truth* statements, in contrast to *natural deduction systems* that use reasoning from *assumptions*. Although the difference between these two formal systems seems to be subtle, the latter can be used more as framework, allowing to build new systems on the logical base of pre-defined premises and formal proof rules.

2.2 Properties of Formal System

Let U be a set of all possible formulae, let $\Gamma = < A, V, \Omega, R >$ be a formal system with set of axioms A, set of propositional variables V, set of logical operators Ω, and set of inference rules R. Then Γ is called:

- *consistent*, if both formula and its negation can not be proved in the system:
  $\nexists \phi \in \Gamma : \Gamma \vdash \phi \wedge \Gamma \vdash \neg\phi \iff \Gamma \nvdash \perp$;

- *complete*, if all true statements can be inferred:
  $\forall \phi \in U : A \vdash \phi \vee A \vdash \neg\phi$;

- *independent*, if no axiom can be inferred from another:
  $\nexists a \in A : A \vdash a.$

In 1931, Kurt Gödel proved his First incompleteness theorem which states that any consistent formal system is incomplete. Later, in 1936, Alfred Tarski extended this result by proving his Undefinability theorem which states that the concept of truth cannot be defined in a formal system. In that case, modern tools, s.a. Coq, often restrict propositions to be either provable or unprovable, rather than true or false.

// TODO – finish it

Note, that the first-order logic is *undecidable*, so that there does not exist a decision algorithm which is sound, complete and terminating [TODO: REFERENCE].

2.3 Lambda-calculus

λ-*calculus* is a universal computation model invented by Alonzo Church in 1930s as a model for formalising the concept of effective computability. This formalism provides solid theoretical foundation for the family of functional programming languages [7]. In λ-calculus, functions are first-order objects, which means functions can be applied as arguments to other functions.

The central concept in λ-calculus is an *expression*, which can be defined as a subject for application the rewriting rules [8]. The basic rewriting rules of λ-calculus are listed below:

- Application: fa is the call of function f with argument a

- Abstraction: $\lambda x.t[x]$ is the function with formal parameter x and body $t[x]$

- Computation: Replace formal parameter by actual argument (β-*reduction*):
  $(\lambda x.t[x])a \rightarrow_\beta t[x := a]$

λ-calculus described above is called the *type-free* λ-calculus. By adding the types to expressions, one can construct more strong calculi and prove useful properties for them (e.g., non-termination ..... // TODO: finish it

2.4 Type Systems

A *type* is a collection of elements. In a type system, each element is associated with a type, which restricts set of possible operations with this element. Types can be thought as a structure of objects, that allows to reveal useful properties of the formal system. Type theory therefore serves as an alternative to classic set theory [9]. In our notation, the $a =_\tau b$ means that a equals b and both of them are of type τ. This notation is used for convenience to encode information about type to equality.

The function that builds new type from another is called *type constructor*. Such functions have been used long before type theories had been constructed formally, even in the 19th century Giuseppe Peano used type constructor S called the *successor* function, along with zero element 0, to axiomatise natural number arithmetic. Thus, number 3 can be constructed as $S(S(S(0)))$.

Simple Type Theory
The type can be defined declaratively, by assigning a label to set of values. Such types are called *simple types*, they can be useful to avoid some paradoxes of set theory, e.g., separating sets of individuals and sets of sets allows to avoid famous Russel's paradox [10]. Simple type theory can extend λ-calculus to a Higher-order logic through connection between formulae and expressions of type Boolean [11].

// == Pure Type System ?

Martin-Löf Type Theory
also known as *Intuitionistic type theory*, is based on the principles of mathematical constructivism which requires to find a way to "construct" an object in order to prove its existence. Therefore, an important place in intuitionistic type theory is hold by the *inductive types* which were constructed recursively using a basic type (zero) and successor function which defines "next" element.

Intuitionistic type theory also uses wide class of *dependent types*, whose definition depends on a value. For instance, the n-ary tuple is a dependent type that is defined by value of n. Although, type checking for such a system is an undecidable problem, as determining the equality of two arbitrary dependent types becomes to be tantamount to problem of inducing the equivalence of two non-trivial programs, which is undecidable in general case according to the Rice's theorem [12].

Calculus of Constructions

Another important constructive type theory is the Calculus of Constructions (CoC) developed by Thierry Coquand and Gérard Huet in 1985 [13]. It represents a natural deduction system which incorporates dependent types, polymorphism and type constructors.

// TODO: MORE DEFINITIONS HERE

The CoC has strong normalisation property, which means that every sequence of inference eventually terminates with an irreducible normal form. This property does not allow to define infinitely recursive structures and functions.

Inference rules for the Calculus of Constructions:

1. $$\frac{}{\Gamma \vdash P : T}$$

2. $$\frac{\Gamma \vdash A : K}{\Gamma, x : A \vdash x : A}$$

3. $$\frac{\Gamma, x : A \vdash B : K \qquad \Gamma, x : A \vdash N : B}{\Gamma \vdash (\lambda x : A.N) : (\forall x : A.B) : K}$$

4. $$\frac{\Gamma \vdash M : (\forall x : A.B) \qquad \Gamma \vdash N : A}{\Gamma \vdash MN : B[x := N]}$$

5. $$\frac{\Gamma \vdash M : A \qquad A =_\beta B \qquad B : K}{\Gamma \vdash M : B}$$

// TODO: add explanations to the inference rules or delete them

In 1990, Christine Paulin proposed the *Calculus of Inductive Constructions* (CIC) as an extension of Calculus of Construction by adding the Martin-Löf's primitive inductive definitions in order to perform efficient computation of the functions over inductive data types. This formalism lies behind the Coq proof assistant.

2.5 Curry-Howard isomorphism

// TODO – this is the most delicious idea among those described in this paper...

"proofs are programs" bla-bla correspondence, isomorphism, very interesting thing...

3   Comparison of some theorem provers

We have chosen for our comparison two automated proof assistants, Coq and Isabelle/HOL [1]. In general, they both work in similar way: given definition of a statement, they can either verify already written proof, or help user to develop such proof in an interactive fashion, so that the invalid proofs cannot be accepted. Both systems have rather large libraries with considerable amount of already proven lemmas and theorems; also, they can be used as functional programming languages as they allow to construct new data types and recursive functions, they have pattern matching, type inference and other features inherent for functional languages. Both tools are being actively developed: on the moment of writing this paper (autumn 2017), the latest versions were Coq 8.7.0 (stable) and Isabelle2017, both released in October 2017.

// TODO: too many 'boths'?..

// TODO: may I say "... inherent for functional languages" ?

Although, the key difference between these two systems is that they are based on different logical theories. Isabelle/HOL exploits higher order logic along with decidable non-dependent types, while Coq is based on Calculus of Inductive Constructions, which uses inductive and dependent types and represents an implementation of intuitionistic logic.

Since their first release, both Coq and Isabelle have already been used to formalize enormous amount of mathematical theorems, including those which have very large or even controversial proof: the Four colour theorem (2004), Lax-Milgram theorem (2017), etc. [14]. Also, the theorem provers have been successfully used for testing and verifying of software programs: general-purpose operating system kernel seL4 (2009) [15], the C standard (2015) [16], and others.

Both Coq and Isabelle have their own IDE to work in (gtk-based CoqIDE and jEdit Prover IDE respectively). They also have numerous of plugins for many popular IDEs, for instance, the Proof General [17] is a plugin for Emacs, which supports lots of proof assistants. During the work on this paper, we used the native IDEs of each proof assistant in order to minimize the impact of third-party tools to our research.

// The idea of comparison below is following: as we have described theoretical foundations so thoroughly, we need to illustrate each big idea with the code snippet for both theorem provers. I wish to place them into two-column table, side-by-side, but perhaps it'll reduce readability. I'll try.

---

[1]Roughly speaking, Isabelle is a core for an automated theorem proving which supports multiple logical theories: Higher-Order Logic (HOL), first-order logic theories s.a. Zermelo-Fraenkel Set Theory (ZF), Classical Computational Logic (CCL), etc. In this paper, we consider the Isabelle/HOL as the startpoint for exploring the power of the proof assistant.

3.1   The Coq theorem prover

Coq is a formal proof assistant system which has been developed at INRIA (Paris, France) since 1984. As a proof assistant system, Coq offers multiple interactive proof methods called *tactics* and decision algorithms for letting the user define new proof methods. A key feature of Coq is a capability of extraction of the verified program (in OCaml, Haskell or Scheme) from the constructive proof of its formal specification [1]. This facilitates using Coq as a tool for software verification.

During the proof process, Coq remembers its state, a set of *premises*, which are considered to be true, and set of *goals* (or subgoals), the statements to be proved. The proof consists of sequence of commands describing which tactic Coq should apply. Tactic may be thought as a pattern of reasoning, it can be already proved rule of inference, removing a hypothesis or introducing of a variable, application of the reasoning principle (s.a. induction), etc. Coq can be asked to try to find appropriate tactic from its collection in the mode *auto*.

In proofs, Coq combines two languages: *Gallina*, a purely functional programming language, and *Ltac*, a procedural language for manipulating the proof process. A statement for proof and structures it relies on are written in Gallina, and the tactics (inference rules) are written in Ltac. These tactics

// ?else?

// Not actually ready examples yet ...

```
1  Inductive nat : Type :=
2    | O : nat
3    | S : nat -> nat.
```

Example 3.1. Recursive function definition: factorial

```
1  ... example here with 'Fixpoint' and 'Inductive'
2  ...// fix x (v : V) : X := t Recursive function.
```

// note: Coq doesn't allow to define recursive functions without decreasing argument => always terminates

Example 3.2. Non-recursive function definition: factorial

```
1  // cheatsheet // fun x : X => t Non-recursive function.
2  ...
```

Example 3.3. Inductive data type definition: ???

```
1  ...
2  ...
```

Example 3.4. Propositional logic proof: de Morgan's law

```
1  Lemma DeMorgan_1 : (forall A B : Prop, ~(A /\ B) -> ~A \/ ~B).
```

```
2  Proof.
3  intros.
4  ...
5  Qed.
```

// ... A practical example: proof of correctness of an algorithm which sums n first members of arithmetic progression using formula $S_n = \frac{2a_1 + d(n-1)}{2} \cdot n$ through direct counting of this sum: $S_n = \sum_{k=0}^{n-1} (a_1 + d \cdot k)$.

Example 3.5. First-order logic proof: formula of the sum of n first members of arithmetic progression

```
1  Lemma  ...  ...
2  Proof.
3  ...
4  ...
5  ...
6  Qed.
```

3.2 The Isabelle/HOL theorem prover

Another widely used proof assistant is Isabelle. Developed by Larry Paulson in Technical University of Munich, it was released for the first time in 1986 (two years after the Coq's first release). It was built in a modular fashion – containing relatively small part of "pure" Isabelle, it has numerous basic theories that describe logic behind Isabelle. In particular, the theory of higher-order logic is implemented as Isabelle/HOL, and it is commonly used because of its expressivity and relative conciseness. As Coq, it combines several languages in its proofs: HOL as a functional programming language (which must be always in quotes), and the language for describing procedures for manipulating the proof. Unlike Coq, Isabelle supports more expressive style of proofs written in a declarative fashion in language Isar.

// IDE allows to randomly pick jump to any place in the syntax tree of the proof and view the state (in spite of Coq, where only the forward-backward operations are allowed).

// some words on termination checks in Isabelle (unlike Coq)

// ?else?

// Not actually ready examples yet ...

Example 3.6. Recursive function definition: factorial

```
1  ...
2  ...
```

Example 3.7. Non-recursive function definition: factorial

```
1  ...
2  ...
```

Example 3.8. Inductive data type definition: list

```
1  ...
2  ...
```

Example 3.9. Propositional logic proof: de Morgan's law

```
1  Lemma DeMorgan_1 : ( forall A B : Prop, ~(A /\ B) -> ~A \/ ~B).
2  Proof.
3  intros.
4  ...
5  Qed.
```

// Isabelle's feature: non-constructive logic!

Example 3.10. Propositional logic proof to the contrary

```
1  Lemma DeMorgan_1 : ( forall A B : Prop, ~(A /\ B) -> ~A \/ ~B).
2  Proof.
3  intros.
4  ...
5  Qed.
```

// Useful features: automatically find a model (SAT)

Example 3.11. Isabelle as an SAT solver

```
1  ... nitpick – finds a model
2  ...
```

// ... [Also] A practical example: proof of correctness of an algorithm which sums n first members of arithmetic progression using formula $S_n = \frac{2a_1 + d(n-1)}{2} \cdot n$ through direct counting of this sum: $S_n = \sum_{k=0}^{n-1}(a_1 + d \cdot k)$.

Example 3.12. First-order logic proof: formula of the sum of n first members of arithmetic progression

```
1  Lemma ... ...
2  Proof.
3  ...
4  ...
5  ...
6  Qed.
```

3.3   Results of comparison

//TODO: perhaps in table:

- expressiveness of logic used
- time of proving

- num of supporting theories

- set of techniques to prove automatically

- Volume of proof (as text)

- num of user interaction steps

- usability

- etc ...

4 Future work

// TODO (Paragraph is still in progress)

< in future, we want to apply this survey to software verification >

References

[1] "Coq proof assistant." `https://coq.inria.fr/`.

[2] "Isabelle, a generic proof assistant." `https://www.cl.cam.ac.uk/research/hvg/Isabelle/`.

[3] "Pvs specification and verification system." `http://pvs.csl.sri.com/`.

[4] W. McCune, "Otter and mace2," 2003. `https://www.cs.unm.edu/~mccune/otter/`.

[5] "Acl2: a computational logic for applicative common lisp." `http://www.cs.utexas.edu/users/moore/acl2/`.

[6] F. Wiedijk, "Comparing mathematical provers," in *MKM*, vol. 3, pp. 188–202, Springer, 2003.

[7] R. Rojas, "A tutorial introduction to the lambda calculus," *arXiv preprint arXiv:1503.09060*, 2015.

[8] H. P. Barendregt, "Introduction to lambda calculus," 1988.

[9] S. Thompson, *Type theory and functional programming*. Addison Wesley, 1991.

[10] A. D. Irvine and H. Deutsch, "Russell's paradox," *Stanford encyclopedia of philosophy*, 2008.

[11] L. C. Paulson, "A formulation of the simple theory of types (for isabelle)," in *COLOG-88*, pp. 246–274, Springer, 1990.

[12] H. G. Rice, "Classes of recursively enumerable sets and their decision problems," *Transactions of the American Mathematical Society*, vol. 74, no. 2, pp. 358–366, 1953.

[13] T. Coquand, *Une Théorie des Constructions*. PhD thesis, Université de Paris VII, 1995.

[14] F. Wiedijk, "Formalizing 100 theorems." `http://www.cs.ru.nl/~freek/100/`.

[15] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood, "sel4: Formal verification of an os kernel," in *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, SOSP '09, (New York, NY, USA), pp. 207–220, ACM, 2009.

[16] R. J. Krebbers, *The C standard formalized in Coq*. [Sl: sn], 2015.

[17] "Proof general." `https://proofgeneral.github.io/`.