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 raised 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 [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 have 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* since they may require some user interaction and may help user to develop new proofs as well. 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, since they can be sufficiently applied in model checking and software verification.

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). As the continuation of that work, we propose more deep comparison of two aforementioned theorem provers – Coq and Isabelle – with respect to criteria s.a. expressiveness, computation power and usability.

1.2 Outline

This paper is organised as follows. Section 2 provides definition, basic properties and theoretical limitations of the formal systems. Section 3 covers general methods for automated reasoning. Section 4 enumerates possible application areas for automated theorem provers. Finally, Section 5 presents target comparison properties and the comparison itself.

2 Axiomatisation and Formal systems

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 completely 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 A logical 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 preserve 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. The proof system described above is called *Hilbert proof system*.

The formulae consists 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 $\Omega = \{\neg, \wedge, \vee, \rightarrow\}$,

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 $\bot$.

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 propositional variables and set of inference rules R. Then Γ is called:

- *consistent*, if no formula of the system contradicts another:
  $\nexists\phi \in \Gamma :\ \Gamma \vdash \phi \wedge \Gamma \vdash \neg\phi\ \Leftrightarrow\ \Gamma \nvdash \bot$;
- *complete*, if all truth 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:
  $\exists a \in A :\ A \vdash a$.

If the propositional variables have no restrictions on their form (i.e. they are 0-arity predicates), such logic is called *propositional logic*. However, if these variables are quantified on some 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 term in .

// TODO – finish it

Although the higher-order logics have stronger semantics than lower-order logics, they have ...

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

2.2 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. According to Church–Turing thesis, λ-calculus is equivalent to Turing machine in sense that any computable function can be expressed using this formalism.

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*):
  $(\lambda x.t[x])a \to_\beta t[x := a]$

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

2.3 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.

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].

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. The function that builds new type from another is called *type constructor*.

Calculus of Constructions
// TODO Dependent types hence type checking may become undecidable.

Inference rules for the calculus of constructions

1. $\dfrac{}{\Gamma \vdash P : T}$
2. $\dfrac{\Gamma \vdash A : K}{\Gamma, x : A \vdash x : A}$
3. $\dfrac{\Gamma, x : A \vdash B : K \qquad \Gamma, x : A \vdash N : B}{\Gamma \vdash (\lambda x : A.N) : (\forall x : A.B) : K}$
4. $\dfrac{\Gamma \vdash M : (\forall x : A.B) \qquad \Gamma \vdash N : A}{\Gamma \vdash MN : B[x := N]}$

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

2.4 Curry-Howard isomorphism

// TODO bla-bla, correspondence, isomorphism... 1934–1969

3 Methods for automated reasoning

// TODO (Paragraph is still in progress)
techniques in common words (and in introduced previously notation), e.g.:

- Clause rewriting Simplification - The concept of (conditional) term rewriting is introduced and its realization as the proof method simp is explained. (from http://isabelle.in.tum.de/cours
1/)
- Resolution
- Sequent Deduction
- Natural Deduction
- The Matrix Connection Method
- Term Rewriting (+lambda calculus)
- Mathematical Induction

4 Some applications of theorem provers

// TODO: think about merging information from this section to introduction.
// TODO (Paragraph is still in progress)
Describe possible applications of formal methods:

1. Interactive theorem proving: construct a formal axiomatic proof of correctness,
2. verifying that a mathematical statement is true.
3. verifying that a circuit description, an algorithm, or a network or security protocol meets its specification:
   - program verification (first-order logic),
   - distributed and concurrent systems (modal and temporal logics),
   - program specification (intuitionistic logic),
   - Model checking: reduce to a finite state space, and test exhaustively.

- hardware verification (higher-order logic),
- logic programming (Horn logic),
- and so on.

5 Comparison of some theorem provers

We have chosen for our comparison two automated proof assistants, Coq and Isabelle. 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. They both have rather large libraries that contain considerable amount of already proven theorems. Both systems can be used as functional programming languages since they allow to construct new data types and recursive functions.

Although, the key difference between these two systems is that they are based on different logical theories. Isabelle exploits higher order logic along with first-order set theory, while Coq operates with dependent types in higher order type theory, representing an implementation of intuitionistic logic.

5.1 The Coq theorem prover

// TODO (Paragraph is still in progress)

Gérard Huet, Thierry Coquand, Christine Paulin

The Coq is a computer tool for verifying theorem proofs. These theorems may concern usual mathematics, proof theory, or program verification

Coq is a formal proof assistant system. It provides a formal language to write mathematical definitions, executable algorithms and theorems together with an environment for semi-interactive development of machine-checked proofs [1]. Coq uses the Calculus of Construction, a higher-order formalism for constructive proofs in natural deduction style, developed by Thierry Coquand [12]. The Calculus of Construction can be considered as an extension of the Curry–Howard isomorphism.

program extraction : Haskell, Scheme and OCaml (add some core features of Coq ...)

(Coq has been used to formalize ...)

5.2 The Isabelle theorem prover

// TODO (Paragraph is still in progress)

// overview good at: https://pdfs.semanticscholar.org/95bf/a1bf0cbf5ae2c2a70daa13d4966143bd96f8.pdf

Main people behind the system: Larry Paulson, Tobias Nipkow, Markus Wenzel

is a theorem prover for various logics, including several first-order logics, Martin-Loef's

Type Theory, and Zermelo-Fraenkel set theory (https://pdfs.semanticscholar.org/95bf/a1bf0cbf5ae2c2a70d

The Isabelle is an interactive theorem prover, which relies on higher-order logic. It allows mathematical formulas to be expressed in a formal language and provides tools for proving those formulas in a logical calculus [2]. Isabelle's main proof method is a higher-order version of resolution, based on higher-order unification.

(add some core features of Isabelle ...)

(Isabelle has been used to formalize ...)

5.3 Joint comparison

//TODO: 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 ...

6 Results

// TODO (Paragraph is still in progress) conclusion of comparison

7 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] G. H. T. Coquard, "The calculus of constructions," 1986.