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, such as 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 in recent years.

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*, they apply techniques of automated logical reasoning to develop 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, such as 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, such as 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 consequences – 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 a 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 contains negation $\neg$, conjunction $\wedge$, disjunction $\vee$, and implication $\to$ operators, although the combination of negation operator with any other of aforementioned operators will be already functionally complete (i.e., any formula can be represented with the usage of these two logical connectives).

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 $\bot$.

The formal system described above is called *the 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:
  $\nexists \phi \in \Gamma : \ \Gamma \vdash \phi \wedge \Gamma \vdash \neg\phi \ \Leftrightarrow \ \Gamma \nvdash \bot$;
- *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, such as 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. The more strong calculi can be constructed by using the types of expressions to the system, for which some useful properties can be proven (e.g., termination or memory safety) [9].

2.4 Type Systems

A *type* is a collection of elements. In a type system, each element is associated with a type, which defines a basic structure of it and restricts set of possible operations with the element. This allows to reveal useful properties of the formal system. Therefore, type theory serves as an alternative to the classic set theory [10]. 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 a 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 [11]. Simple type theory can extend λ-calculus to a Higher-order logic through connection between formulae and expressions of type Boolean [12].

Martin-Löf Type Theory

The Martin-Löf type theory, also known as the *Intuitionistic type theory*, is based on the principles of mathematical constructivism, that require explicit definition of the way of "constructing" an object in order to prove its existence. Therefore, an important place in

intuitionistic type theory is held by the *inductive types*, which were constructed recursively using a basic type (zero) and successor function which defines "next" element.

The Intuitionistic type theory also uses a 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 the value of n. However, the type checking for such a system is an undecidable problem since determining of the equality of two arbitrary dependent types turns to be tantamount to a problem of inducing the equivalence of two non-trivial programs (which is undecidable in general case according to the Rice's theorem [13]).

Calculus of Constructions

Another important constructive type theory is the Calculus of Constructions (CoC) developed by Thierry Coquand and Gérard Huet in 1985 [14]. 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.

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 [15]. 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] as they both are widely used tools for theorem proving.

"There are theorems of CPC that are unprovable in IPC, but not vice versa, so the latter logic is strictly weaker. "

[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 such as 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 this 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 (such as 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 // TODO !!!

// "Coq's built-in logic is very small: the only primitives are Inductive definitions, universal quantification (forall), and implication (->), while all the other familiar logical connectives — conjunction, disjunction, negation, existential quantification, even equality — can be encoded using just these."

// In Coq, "Implications are functions"

// The `False` in Coq is a type without a constructor: "Inductive False : Prop := ." "Intuition: False is a proposition for which there is no way to give evidence." => show the proof of 'ex falso quodlibet' ?

"Inductive True : Prop := I : True." ("a proposition for which it is trivial to give evidence" => always true proposition)

// ?else?

3.2 The Isabelle/HOL theorem prover

Isabelle was developed by Larry Paulson in Technical University of Munich as a successor of HOL theorem prover [16]. Isabelle was released for the first time in 1986 (two years after the Coq's first release). It was built in a modular manner, i.e., it has relatively small core, which can be extended by 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. Similarly to 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)

// `A ⟹ B ⟹ C` means `A ⟹ (B ⟹ C)` and `[| A; B |] ⟹ C`, where `A` and `B` are premises, `C` is conclusion.

// HOL: Higher-order = functions are values, too

// termination of computation. see:

// ?else?

3.3 Common features

In general, both Coq and Isabelle work in a 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; in addition, 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.

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, such as Four colour theorem (2004), Lax-Milgram theorem (2017), and other important theorems [17]. Moreover, the theorem provers have been successfully used for testing and verifying of software programs, including the general-purpose operating system kernel seL4 (2009) [18], the C standard (2015) [19], and others.

Both Coq and Isabelle have their own Integrated Development Environment (IDE) to work in (gtk-based CoqIDE and jEdit Prover IDE, respectively). In general, both native IDEs of these theorem provers provide the facility for interactive executing scripts step-by-step while preserving the state of proof (*environment*), which for each step describes the set of premises along with already proved statements (*context*) and the set of statements to be proven (*goals*). However, Isabelle's native IDE allows to change the proof state arbitrarily, in contrast to the CoqIDE, which provides only the capability of switching the proof state to the next or previous statement only. Alternatively, both considering theorem provers have numerous of plugins for many popular IDEs, for instance, the Proof General [20] is a plugin for Emacs, which supports numerous 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.

// for example, addition in Isabelle is defined as :

```
fun add :: "nat ⇒ nat ⇒ nat" where
  "add 0 n = n" |
  "add (Suc m) n = Suc(add m n)"
```

3.4 Differences

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.

Intuitionistic Logic: excl middle is forbidden => not not P no longer implies P and (p or not p) is no longer provable.

// DIFFERENCES BTW CLASSIC AND INTUITIONISTIC LOGIC see slide page 101 (and book by prof.)

// Existence proofs are often non-constructive

// tactics = inference rules

3.5 Example Proofs

//Let's prove some propositional tautologies here

Propositional Intuitionistic Logic in Coq

// in '(*' brackets - comment

```
Check False. (* output: Prop : Type *)
Print False. (* output: Inductive False : Prop :=  *)
Print True.  (* output: Inductive True : Prop :=  I : True *)
```

Figure 1. Definition of basic propositional terms in Coq

<tauto - the tactic to automatically prove propositional tautologies (need definition of tautology)> //Let's prove that $\vdash (P \vee \neg P)$

```
Theorem DeMorgan_Auto_Coq : forall P Q : Prop,
    ~P /\ ~Q <-> ~(P \/ Q).
Proof.
    tauto.
Qed.
```

<but Coq's logic does not include the axiom of excluded middle:>

```
Lemma ExcludedMiddle_AutoFails_Coq:
```

```
    forall P, P \/ ~P.
Proof.
    try tauto.  (* tauto fails in Coq's intuitionistic logic *)
Abort.
```

// Note, that Coq's logic can be extended by assuming the excluded middle fact as an axiom (we use that in the proof ??? of nth number of arith progr)

Nonetheless, <idea: Assuming NOT(A OR NOT A), you first prove A, then find a contradiction.>

```
Lemma DoubleNegatedExcludedMiddle_Coq:
    forall P : Prop, ~~(P \/ ~P).
Proof.
    unfold not.     (* apply ~P ==> P -> False *)
    intros P f.     (* move premises to the set of hypotheses *)
    apply f.        (* replace the goal with premise of implication in f *)
    right.          (* apply disjunction elimination inference rule *)
    intro P_holds. (* move P to the set of hypotheses *)
    apply f.        (* replace the goal with premise of implication in f *)
    left.           (* apply disjunction elimination inference rule *)
    exact P_holds. (* match the goal with one of the hypotheses *)
Qed.
```

// so, in this way intuitionistic logic can model classic logic (?)

In Isabelle, vice versa: it can model intuitionistic logic by not using the excluded middle law (see theory IFOL [REFERENCE]):

Still, double negation is easily proved in classic logic:

//Figure **??** shows the proof of De Morgan's law in Coq system. The law statement is being proved for all propositions of type `Prop` <which is ...> ... . For arguments of type `bool` the proof is trivial.

//Let's prove that $\neg(P \vee Q) \leftrightarrow (\neg P \wedge \neg Q)$ (* this simple propositional formula can be easily solved by the tactic tauto *)

```
Theorem DeMorganPropositional_Coq:
    forall P Q : Prop, ~(P \/ Q) <-> ~P /\ ~Q.
Proof.
    intros P Q. unfold iff.
    split.
    - intros H_not_or. unfold not. constructor.
      + intro H_P. apply H_not_or. left. apply H_P.
      + intro H_Q. apply H_not_or. right. apply H_Q.
    - intros H_and_not H_or.
      destruct H_and_not as [H_not_P H_not_Q].
      destruct H_or as [H_P | H_Q].
```

```
      + apply H_not_P. assumption.
      + apply H_not_Q. assumption.
Qed.
```

Figure 2. Proof of propositional logic tautology in Coq: the de Morgan's law for propositions

//For Isabelle, same proof:

```
lemma DeMorganPropositional_Isabelle: "(¬ (P ∧ Q)) = (¬ P ∨ ¬ Q)"

apply (rule iffI)       (* split equality into two subgoals *)
(* "Forward" subgoal: 1. ¬(P ∧ Q)  ⟹  ¬ P ∨ ¬ Q *)
apply (rule classical) (* 1. ¬ (P ∧ Q)  ⟹  ¬ (¬ P ∨ ¬ Q)  ⟹  ¬ P ∨ ¬ Q *)
apply (erule notE)      (* 1. ¬ (¬ P ∨ ¬ Q)  ⟹  P ∧ Q *)
apply (rule conjI)      (* 1. ¬ (¬ P ∨ ¬ Q)  ⟹  P; 2. ¬ (¬ P ∨ ¬ Q)  ⟹  Q *)
apply (rule classical) (* 1. ¬ (¬ P ∨ ¬ Q)  ⟹  ¬ P  ⟹  P *)
apply (erule notE)      (* 1. ¬ P  ⟹  ¬ P ∨ ¬ Q *)
apply (rule disjI1)     (* 1. ¬ P  ⟹  ¬ P *)
apply assumption        (* 1. (solved). 2. ¬ (¬ P ∨ ¬ Q)  ⟹  Q *)
apply (rule classical) (* 2. ¬ (¬ P ∨ ¬ Q)  ⟹  ¬ Q  ⟹  Q *)
apply (erule notE)      (* 2. ¬ Q  ⟹  ¬ P ∨ ¬ Q *)
apply (rule disjI2)     (* 2. ¬ Q  ⟹  ¬ Q *)
apply assumption        (* 2. (solved) *)
(* "Backward" subgoal: 3. ¬ P ∨ ¬ Q  ⟹  ¬ (P ∧ Q) *)
apply (rule notI)    (* 3. ¬ P ∨ ¬ Q  ⟹  P ∧ Q  ⟹  False *)
apply (erule conjE) (* 3. ¬ P ∨ ¬ Q  ⟹  P  ⟹  Q  ⟹  False *)
apply (erule disjE) (* 3. P  ⟹  Q  ⟹  ¬P  ⟹False; 4. P  ⟹  Q⟹  ¬Q⟹  False *)
apply (erule notE, assumption)+  (* 3. (solved); 4. (solved) *)

done
```

<very hardly provable stmt 2**sqrt(2) irrational....>

First-order Logic Proofs

// see slide 53 from

```
Lemma DeMorganQuantified_Coq:
    forall P:A -> Prop, ~(forall x:A, P x) -> exists x: A, ~P x.
Proof.
    unfold not.             (* unfold negation *)
    intros P H_notall.      (* move premises to the set of hypotheses *)
    apply NNPP.             (* apply ~~P ==> P *)
    unfold not.
    intro H_not_notexist.
    cut (forall x:A, P x).       (* add new goal from the goal's premise *)
    exact H_notall.
    intro x.
    apply NNPP.
    unfold not.
```

```
    intros H_not_P_x.
    apply H_not_notexist.
    exists x.
    exact H_not_P_x.
Qed.
```

For Isabelle, see here

```
lemma DeMorganQuantified_Isabelle:
    assumes "¬ (∀x. P x)"
    shows "∃x. ¬ P x"
    proof (rule classical)
        assume "∄x. ¬ P x"
        have "∀x. P x"
        proof
            fix x show "P x"
            proof (rule classical)
                assume "¬ P x"
                then have "∃x. ¬ P x" ..
                with <∄x. ¬ P x> show ?thesis by contradiction
        qed
    qed
    with <¬(∀x. P x)> show ?thesis by contradiction
qed
```

Intuitionistic Type Theory

<here bool the datatype, which is inductively defined as a `Set` => can use destruct, which expands the definition of inductive type>

```
Check bool.  (* output: bool : Set *)
Check false. (* output: false : bool *)
Print bool.  (* output: Inductive bool : Set :=  true : bool | false : bool *)

(* define macroses: *)
Notation "a || b" := (orb a b).
Notation "a && b" := (andb a b).

Theorem DeMorganBoolean_Coq:
    forall a b: bool, negb (a || b) = ((negb a) && (negb b)).
Proof.
    intros a b.
    destruct a; simpl; reflexivity.
Qed.
```

Figure 3. Propositional logic proof: de Morgan's law for booleans

//for Isabelle, see p.10 'Polymorph. types', e.g. prove "Suc = Suc"

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

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

Figure 4. Recursive function definition: factorial

// "Syntactic restriction on recursive calls on term": // Coq doesn't allow to define recursive functions without decreasing argument => always terminates

```
...
...
```

Figure 5. Inductive data type definition: ???

// ... 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: $S_n = \sum_{k=0}^{n-1} (a_1 + d \cdot k)$.

proofs by simplification (`omega` in Coq)

```
Require Import Coq.omega.Omega.
Require Coq.Logic.Classical.

Fixpoint range_sum (n: nat) : nat :=
match n with
    | O => 0
    | S p => range_sum p + (S p)
end.
Compute range_sum 3.

Lemma range_sum_successor_lemma: forall n: nat,
    range_sum (n + 1) = range_sum n + (n + 1).
Proof.
    intros.
    induction n.
    - simpl; reflexivity.
    - simpl; omega.
Qed.

Theorem SimpleArithProgressionSumFormula_Coq: forall n,
    2 * range_sum n = n * (n + 1).
Proof.
    intros.
    induction n.
    - simpl; reflexivity.
    - rewrite -> Nat.mul_add_distr_l.
```

```
      rewrite -> Nat.mul_1_r.
      rewrite -> (Nat.mul_succ_l n).
      rewrite <- (Nat.add_1_r n).
      rewrite -> range_sum_succ.
      omega.
Qed.
```

Isabelle:

```
fun range_sum :: "nat ⇒ nat"
  where "range_sum n = (\<Sum>k::nat=0..n . k)"
value "range_sum 10"

theorem SimpleArithProgressionSumFormula_Isabelle: "2 * (range_sum n) = n * (n + 1)"
proof (induct n)
    show "2 * range_sum 0 = 0 * (0 + 1)" by simp
  next
    fix n have "2 * range_sum (n + 1) = 2 * (range_sum n) + 2 * (n + 1)" by simp
    also assume "2 * (range_sum n) = n * (n + 1)"
    also have "\<dots> + 2 * (n + 1) = (n + 1) * (n + 2)" by simp
    finally show "2 * (range_sum (Suc n)) = (Suc n) * (Suc n + 1)" by simp
qed
```

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

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

Figure 6. Propositional logic proof to the contrary

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

```
... nitpick - finds a model
...
```

Figure 7. Isabelle as an SAT solver

// ... [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: $S_n = \sum_{k=0}^{n-1}(a_1 + d \cdot k)$.

```
Lemma ... ...
Proof.
...
...
...
Qed.
```

Figure 8. First-order logic proof: formula of the sum of n first members of arithmetic progression

3.6 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 ...

Coq requires more mathematical background to even understand manuals (not just tactics, but rather deep knowledge of lambda-calculus to understand different types of reduction, functional programming and inductively created types ... ...)

Isabelle has more powerful meta-language syntax to operate tactics: it allows to use the functional combinators of tactics called tacticals for building up complex tactics from simpler ones. Common tacticals perform sequential composition, disjunctive choice, iteration, or goal addressing, which make Isar the Turing-complete language in the domain of logic <TODO: say better (?)>. [REFERENCE: implementaion.pdf in Isabelle Jedit]. In comparison, Coq allows to use only the composition of tactics. NOTE!!! Coq allows combinations of tactics as well, see:

Moreover, Isar allow more relaxed syntax in sense that its proof may look more like mathematical proof because of writing the goals during the proof (e.g., '`assume "¬ P x" then have "∃x. ¬ P x"`') and using the connectors that can be read conveniently by human (e.g., '`then`' abbreviates '`from this`', '`hence`' expands to '`then have`', etc.). In contrast, proofs written in Coq look more like programs written in imperative programming language: the sequence of states need to be executed in order to check how the directives (application of tactics) change the state of proof. On the other hand, the brevity of Gallina language may let the experienced user, that knows the syntax and tactics well, spend less time for writing the proof in Coq rather than in Isabelle.

Isabelle is very well documented, all its manuals and documentation is compiled as a set of pdf files and is supplied with Isabelle setup bundle. Coq's manuals were in web, so numerous of files were inaccessible or even lost.

4 Future work

// TODO (Paragraph is still in progress)

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

5 Appendix

```
Require Import Coq.omega.Omega.
Require Coq.Logic.Classical.

Fixpoint nth_member (a_0 d n: nat) : nat :=
match n with
| 0 => a_0
| S p => d + (nth_member a_0 d p)
end.
Compute nth_member 3 2 4.

Theorem nth_member_formula : forall a_0 d n: nat,
(nth_member a_0 d n) = a_0 + d * n.
Proof.
intros.
induction n; simpl.
- rewrite -> Nat.mul_0_r.
rewrite -> Nat.add_0_r.
reflexivity.
- induction n; rewrite -> IHn.
+ rewrite -> Nat.mul_0_r.
rewrite -> Nat.add_0_r.
rewrite -> Nat.mul_1_r.
rewrite -> Nat.add_comm.
reflexivity.
+ rewrite <- Nat.add_1_l.
rewrite -> (Nat.mul_succ_r d (1 + n)).
omega.
Qed.
```

Figure 9. First-order logic proof: formula of the sum of n first members of arithmetic progression

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] H. Barendregt, W. Dekkers, and R. Statman, *Lambda calculus with types*. Cambridge University Press, 2013.

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

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

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

[13] 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.

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

[15] C. Paulin-Mohring, "Introduction to the calculus of inductive constructions," 2015.

[16] "Hol, interactive theorem prover." https://hol-theorem-prover.org/.

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

[18] 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.

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

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