Comparison of Two Theorem Provers: Isabelle/HOL and Coq

Artem Yushkovskiy[1,2] and Prof. Stavros Tripakis[1]

[1]Department of Computer Science,
Aalto University (Espoo, Finland)

[2]Department of Computer System Design and Security,
ITMO University (St. Petersburg, Russia)

Autumn 2017

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, 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, Isabelle/HOL [1] and Coq [2], with respect to expressiveness, limitations and usability. For this reason, it firstly gives a brief introduction to the bases of formal systems and automated deduction theory, their main problems and challenges, and then provides detailed comparison of most notable features of the selected theorem provers with support of illustrative proof examples.

KEYWORDS: proof assistants, Coq, Isabelle/HOL, logics, proof theory, formal method, classical logic, intuitionistic logic, usability.

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. As a result, the *formal approach* was developed as a new methodology for manipulating the abstract essences 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 (defined by logic). Such tools are sometimes called *proof assistants*, their purpose is to help users to develop new proofs. The tools *Isabelle* [1], *Coq* [2], *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 automatically. 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.

This paper is organised as follows. Section 2 describes basic foundations of logic necessary for understanding theorem provers. In particular, Section 2.1 provides formal definition, Sections 2.2–2.4 describe different types, basic properties and theoretical limitations of formal systems. Section 3 presents the comparison itself and provides the illustrative examples of different kinds of proofs in both considering systems.

1.1 Related work

A considerably extensive survey on theorem provers has been presented by F. Wiedijk [6], where fifteen most common systems for the formalization of mathematics were compared against various properties, in particular, size of supporting libraries, expressiveness of underlying logic, size of proofs (the de Bruijn criterion) and level of automation (the Poincaré principle). Another notable work was presented by D. Griffioen and M. Huisman [7], in which two theorem provers, PVS and Isabelle/HOL, were deeply compared with respect to numerous important aspects, such as properties of used logic, specification language, user interface, etc. This paper proposes analogous comparison of two widely used theorem provers, Isabelle and Coq, with respect to expressiveness, limitations and usability.

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 formulas 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.

The formulas consist of *propositional variables*, connected with *logical connectives* (or logical operators) according to rules, defined by a formal language. The formulas, which satisfy such rules, are called *well-formed formulas* (wff). Only wff can form judgements in a formal system. The propositional variable is an atomic formula that can be claimed as 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 $\rightarrow$ 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 operates with two quantifiers, the universal quantifier $\forall$ and the existential quantifier $\exists$. Thereafter, the *second-order logic* extends it by adding quantifiers over first-order quantified sets — 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 (for instance, the expression $\forall f : \mathit{bool} \rightarrow \mathit{bool}, f\ (f\ (f\ x)) = f\ x$ could be considered in higher-order logic), or the *type theory*, which assigns a type for every expression in the formal language (see Section 2.4).

A formal system determines the set of derivable *formulas* (judgements that are provable with respect to the rules of formal system). Let Φ be a set of formulas. Initially, it only consists of *hypotheses*, a priori true formulas, 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 is called *tautology* and denoted as $\vdash \phi$ (meaning $\emptyset \vdash \phi$). The formula is called *contradiction* if $\vdash \neg\phi$. Obviously, all contradictions are equivalent in one formal system, they are denoted as $\perp$.

In current paper, the notation (1), which was borrowed from the Isabelle documentation, will be used for expressing the rules of inference. In this notation, the sign $\Longrightarrow$ means logical implication, which is right-associative, see formula (3). This notation is equivalent to the standard notation (2):

$$[\![A_1; A_2; \ldots A_n]\!] \Longrightarrow B \tag{1}$$

$$\equiv \{A_1, A_2, \ldots A_n\} \vdash B \tag{2}$$

$$\begin{aligned} A_1 \Longrightarrow A_2 \Longrightarrow \cdots \Longrightarrow A_n \Longrightarrow B \\ \equiv A_1 \Longrightarrow (A_2 \Longrightarrow (\cdots \Longrightarrow (A_n \Longrightarrow B))) \end{aligned} \tag{3}$$

The formulas below describe the principal inference rule residing in most logic systems, the *Modus ponens* (MP) rule, and two main axioms of classical logic:

$$[\![A, A \Longrightarrow B]\!] \Longrightarrow B \tag{MP}$$

$$A \Longrightarrow (B \Longrightarrow A). \tag{A1}$$

$$(A \Longrightarrow (B \Longrightarrow C)) \Longrightarrow ((A \Longrightarrow B) \Longrightarrow (A \Longrightarrow C)). \tag{A2}$$

Together with axioms (A1) and (A2), Modus ponens rule forms the Hilbert proof system which can process statements of classical propositional logic. Other classical logic systems often include the axiom of excluded middle (EM), and may derive the double negation introduction (DNi) and double negation elimination (DNe) laws:

$$A \vee \neg A. \tag{EM}$$

$$A \implies \neg\neg A \tag{DNi}$$

$$\neg\neg A \implies A \tag{DNe}$$

Many classical logics may derive the de Morgan's laws (DM1), (DM2), the law of contraposition (CP), the Peirce's law (PL) and many other tautologies:

$$\neg(A \wedge B) \Longleftrightarrow \neg A \vee \neg B \tag{DM1}$$

$$\neg(A \vee B) \Longleftrightarrow \neg A \wedge \neg B \tag{DM2}$$

$$(A \to B) \implies (\neg B \to \neg A) \tag{CP}$$

$$((A \to B) \to A) \implies B \tag{PL}$$

The axiom of excluded middle means that every logical statement is decidable, which might not be true in some applications. Adding this axiom to the formal system leads to the reasoning from *truth* statements, in contrast to *natural deduction systems* that use reasoning from *assumptions*. Although the difference between these two kinds of 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 formulas, 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:
  $\nexists a \in A : \ A \vdash a$.

For instance, the Hilbert system described above is consistent and independent, yet incomplete under the classical semantics. 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.

2.3 Lambda-calculus

The necessity of building the automatic reasoning systems has lead to development of models that abstract the computation process. That time, the concept of effective computability was being evolving rapidly, causing development of multiple formalisations of computation, such as Turing Machine, Normal Markov algorithms, Recursive functions, and other. One of the fist and most effective models was *λ-calculus* invented by Alonzo Church in 1930s. This formalism provides solid theoretical foundation for the family of functional programming languages [8]. 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 [9]. 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* (β*-reduction*): replace formal parameter x with actual argument a: $(\lambda x.t[x])a \rightarrow_\beta t[x := a]$

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 and memory safety) [10].

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 classical set theory [11].

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 $S(S(S(0))))$.

2.4.1 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 [12]. Simple type theory can extend λ-calculus to a higher-order logic through connection between formulas and expressions of type Boolean [13].

2.4.2 Martin-Löf type theory

The Martin-Löf type theory, also known as the *Intuitionistic type theory*[1], is based on the principles of constructive mathematics, 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 [14]).

[1] In this paper, in terms *intuitionistic type theory* and *intuitionistic logic*, the word *intuitionistic* is used as a synonym for *constructive*.

2.4.3 Calculus of Constructions

Another important constructive type theory is the Calculus of Constructions (CoC) developed by Thierry Coquand and Gérard Huet in 1985 [15]. It represents a natural deduction system which incorporates dependent types, polymorphism and type constructors. The typed polymorphic functional language of CoC allow to define inductive definitions, although rather inefficiently [16].

Whenever an inductive type is defined, the task of *type-checking* becomes equivalent to the task of executing corresponding function in a programming language. Although in many programming languages type-checking algorithm is efficient, type-checking in CoC is *undecidable* in general case. This problem is closely related to the *Curry-Howard isomorphism*, a direct relationship between a program and an intuitionistic proof, in which a base type of the program is equivalent to a propositional variable in the proof, an empty type represents *false* and a singletone type represents *truth*, a functional type $T_1 \rightarrow T_2$ corresponds to an implication, a product type $T_1 * T_2$ and a sum type $T_1 + T_2$ correspond to conjunction and disjunction, respectively [17]. Thus the Calculus of Constructions can be considered as an extension of the Curry-Howard isomorphism. An important feature of CoC type system is that it holds the strong normalisation property, which means that every sequence of inference eventually terminates with an irreducible normal form.

Although the language of CoC is rather expressive, its expressiveness is not enough to prove some natural properties of types. In order to overcome this drawback, the *Calculus of Inductive Constructions* (CIC) was developed by Christine Paulin in 1990. CIC is implemented by adding the Martin-Löf's primitive inductive definitions to the CoC in order to perform the efficient computation of functions over inductive data types in higher-order logic [16]. This formalism lies behind the Coq proof assistant.

3 Comparison of two theorem provers

In this work, two automated proof assistants, *Isabelle/HOL*[2] and *Coq* have been chosen for comparison as they both are widely used tools for theorem proving (according to the number of theorems that have already been formalised, see [18]).

This section discusses some common and different features of these two theorem provers, providing illustrative examples of proofs performed in Coq and Isabelle. As a startpoint, the de Morgan's laws (DM1) and (DM2) in propositional and first-order logics have been chosen. Afterwards, the formula for sum of first n natural numbers, defined inductively in both Isabelle and Coq, is being discussed. An example of extraction in Coq the verified code in Haskell and OCaml follows the proof of correctness of this formula.

3.1 The Isabelle/HOL theorem prover

Isabelle was developed as a successor of HOL theorem prover [19] by Larry Paulson at the University of Cambridge and Tobias Nipkow at Technische Universität München. 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., around a relatively small core, which can be extended by numerous basic theories that

[2]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, the Isabelle/HOL has been considered as the startpoint for exploring the power of this proof assistant.

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.

Isabelle exploits classical logic, so even propositional type is declared as a set of two elements `true` and `false` (thus any n-ary logic can be easily formalised). In proofs, Isabelle combines several languages: *HOL* as a functional programming language (which must be always in quotes), and *Isar* as the language for describing procedures in order to manipulate the proof.

3.2 The Coq theorem prover

Coq is another widespread proof assistant system that has been developed at INRIA (Paris, France) since 1984. Coq is based on Calculus of Inductive Constructions, an implementation of intuitionistic logic which uses inductive and dependent types. Nonetheless, Coq's logic may be easily extended to classical logic by assuming the excluded middle axiom (EM). 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 [20]. This facilitates using Coq as a tool for software verification.

Being based on the constructive foundation, Coq has two basic meta-types, `Prop` as a type of propositions, and `Set` as a type of other types. Unlikely Isabelle's type system, the `True` and `False` propositions are defined as of type of `Prop`, so that in order to be valid they need to be either assumed or proven (see Appendix A.1 Fig.5). Nonetheless, Coq's library has the `bool` definition, which is of type of `Set` in the manner of Isabelle's proposition (as simple as enumeration of two elements, tertium non datur; see Appendix A.1 Fig.6).

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, while the proof process itself is being controlled by the commands written in Ltac language.

3.3 Common features

In general, both Isabelle and Coq work in a similar way: given the definition of a statement, they can either verify already written proof, or assist user in developing such a proof in an interactive fashion, so that the invalid proofs are not accepted. During the proof process, the systems save the proof state, a set of *premises* and set of *goals* (the statements to be proved). Therefore, the proof may represent the sequence of *tactics* applied to the proof state. A tactic may be thought as an inference rule, it can use already proved statements, remove hypotheses or introduce variables. Some tactics work on very high level, they can automatically solve complex equations or prove complex statements, so that the proof assistant acquires features of an automated theorem provers described in Section 1.

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 Isabelle and Coq 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 [18]. Moreover, the theorem provers have been successfully used for testing and verifying of software programs, including the general-purpose operating system kernel seL4 (2009) [21], the C standard (2015) [22], and others.

Both Isabelle and Coq 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 backward or forward linearly. Alternatively, both considering theorem provers have numerous of plugins for many popular IDEs, for instance, the Proof General [23] is a plugin for Emacs, which supports numerous proof assistants. During the work on this paper, only native IDEs of each proof assistant have been used in order to minimize the impact of third-party tools to the research.

Both systems accept proofs written in an imperative fashion (*forward proof*), i.e., such proof represents a sequence of tactic calls, that implicitly change the proof state at each step, compounded by the control-flow operators called *tacticals*, that combine tactics together, separate their results, repeat calls, etc. In addition, the syntax of Isar permits writing goals explicitly in the proof (*backward proof*, see Appendix A.6 Fig. 19 and Fig.15).

3.4 Major differences

The key differences between Isabelle and Coq lie in differences between logical theories they based on. While Isabelle/HOL exploits higher order logic along with non-dependent types, Coq is based on intuitionistic logic, which does not include the axiom of excluded middle (EM) essential for classical logics. Consequently, the double negation elimination rule (DNe) does not hold, however the double negation introduction law (DNi) can be easily proven (see Figures 1 and 2). This follows from the fact that, if a proposition is known as truth, then double negation works as in classic logic, but if the proposition truthfulness is to be proven from its double negation, then there is nothing known about the proposition itself so far.

```
Lemma DoubleNegElim_Coq : forall P: Prop,
    ¬¬P -> P.
Proof.
    try tauto.  (* fails *)
Abort.
```

Figure 1: Proof failure of the (DNe) rule in Coq

```
Lemma DoubleNegIntro_Coq : forall P: Prop,
    P -> ¬¬P.
Proof.
    (* automatic 'tauto' works here *)
    unfold not.
    intros P P_holds P_impl_false.
    apply P_impl_false. apply P_holds.
Qed.
```

Figure 2: Proof of the (DNi) rule in Coq

In addition, the double-negated axiom of excluded middle can be proven as well solely in intuitionistic logic, see Appendix A.2 Fig.11. This is a way for embedding the classical propositional logic into intuitionistic logic and known as *Glivenko's double-negation translation* [24], which maps all classical tautologies to intuitionistic ones by double-negating them. Furthermore, there are other schemes of the translation for other classical logics, such as Gödel-Gentzen translation, Kuroda's translation, etc. [25].

Therefore, numerous of theorems, such as the classical logic tautology Peirce's law (PL), can not be proved in intuitionistic logic, while being valid in classical logic, which makes the latter strictly weaker [26] and incomplete (Coq's tactic for automatic reasoning of propositional statements `tauto` fails to prove this automatically).

In classical logic, some proofs remain valid, yet completely inapplicable. For instance, the following non-constructive proof of the statement "*there exist algebraic irrational numbers x and y such that x^y is rational*" may serve as a classic example of it. The proof relies on the axiom of excluded middle [27]. Consider the number $\sqrt{2}^{\sqrt{2}}$; if it is rational, then consider $x = \sqrt{2}$ and $y = \sqrt{2}$, which both are irrational; if $\sqrt{2}^{\sqrt{2}}$ is irrational, then consider $x = \sqrt{2}^{\sqrt{2}}$ and $y = \sqrt{2}$, so that x^y is rational, q.e.d. Although this proof is clear and concise, it reveals no information about whether the number $\sqrt{2}^{\sqrt{2}}$ is rational or irrational. More importantly, it gives no algorithm for finding other such numbers. Therefore, the main purpose of constructive proofs is to define such a solution schema for a problem, in addition to proving the claim. Commonly, the proofs of existence[3] of an element are non-constructive as in order to prove such a statement it is enough to find single valid example.

3.4.1 Proofs in propositional logic

As an example of proof statement in propositional logic, the de Morgan's law (DM2) has been chosen. Although both proof assistants can operate with propositional statements, the proof in Isabelle relies on the classical logic by applying excluded middle (EM) axiom (see "`apply (rule classical)`", Appendix A.3 Fig.12), and the proof in Coq does not use this axiom, working completely within intuitionistic logic with propositional variables of meta-type `Prop` (see Appendix A.3 Fig.13). Note that both system can prove this statement automatically (using tactic `blast` in Isabelle or tactic `tauto` in Coq).

The proof in Coq can be much simpler if the theorem is formulated with usage of *Set*-type `bool` (see definition of `bool` in Appendix A.1 Fig.6, see proof in Appendix A.3 Fig.14). There, it is possible to use the tactic `destruct` to decompose type to different goals and prove them separately (in Coq, the ';' operator between two tactics instructs interpreter to apply next tactic to all subgoals produced by previous tactic call). Note that when the theorem was formulated in terms of variables of meta-type `Set`, the automatic tactic `tauto` fails, as it works only with propositions of meta-type `Prop`.

3.4.2 Proofs in first-order logic

As an example of proof in first-order logic, the first-order quantified de Morgan's laws have been chosen. In both Coq and Isabelle, the proof necessarily relies on the axiom of excluded middle as the *existence* of an element is to be proven[4]. The proof in Isabelle is written as a *backward proof* (see Appendix A.4 Fig.15). The Coq's proof imports the library `Coq.Logic.Classical_Prop`, which contains definitions of classical logic, which are useful to extend intuitionistic logic to classical logic (see Appendix A.4 Fig.16).

3.4.3 Proofs using inductive types

In both Isabelle and Coq, the natural numbers type `nat` is defined inductively on induction on zero as in Peano arithmetic (see Appendix A.1 Fig.7 and Fig.8). As an example of statement with

[3] as well as proofs of non-universally valid statements "$\neg\forall$", which in classical logics are equivalent to existence proofs "$\exists$".

[4] in contract to the previous proofs formulated in propositional logic, where the existence of both propositions was assumed.

the type `nat`, the simple formula $2 \cdot S_n = n \cdot (n+1)$ for sum S_n of first n integer numbers has been chosen (see proof in Isabelle in Appendix A.6 Fig.19, see proof in Coq in Appendix A.6 Fig.20). Note that the proof in Coq uses the library `Coq.omega.Omega`, which contains powerful tactics to simplifying and proving natural numbers formulas.

3.4.4 Code extraction in Coq

Furthermore, after the correctness of defined function `range_sum` has been proven, it is possible to extract from Coq the verified function code in Haskell or Ocaml:

```
range_sum :: Nat -> Nat
range_sum n =
  case n of {
    O -> O;
    S p -> add (range_sum p) (S p)}
```

Figure 3: Extracted function in Haskell

```
(** val range_sum : nat -> nat **)

let rec range_sum = function
    | O -> O
    | S p -> add (range_sum p) (S p)
```

Figure 4: Extracted function in OCaml

3.5 Results of comparison

In this paper, the authors have made an attempt to compare to different theorem provers, Coq and Isabelle/HOL, and both of them have been found highly developed and valuable, although they both require deep understanding of metamathematical concepts of the proof process. The list below summarises the main features of these two tools that the authors have noticed.

- *Expressiveness of underlying logic:*
  - Isabelle/HOL uses classical higher-order logic;
  - Coq uses intuitionistic logic based on Calculus of Inductive Constructions theory, but may be extended to classical logic by assuming the axiom of excluded middle.
- *Necessary background for using the theorem prover:*
  - From the author's personal point of view, Coq requires deeper understanding of underlying logic theory, since usually the intuitionistic logic is being studying as a further development of classical logic that adds large number of additional constraints to it;
  - Nonetheless, the whole proof process may seem unfamiliar for users with traditional mathematical background, so that for these users both systems require large amount of additional learning (at least, understanding and memorising the most common tactics is least necessary requirement for using these systems).
- *The level of the proof automation:*
  - Both systems have automatic tactics for proving (e.g., `auto` in Isabelle; `auto`, `tauto` in Coq) or simplification complex statements (e.g., automatic reasoner `blast` in Isabelle; automatic tactics `simpl`, `omega` in Coq). However, in some cases these tactics offer insufficient level of automation, particularly in proving theorems over natural numbers (see example in Appendix A.6 Fig.20, where numerous steps for rewriting the equation by calling `rewrite` had been performed in order to apply automatic tactic `omega`).
- *Size of proof:*
  - Analogous proofs have approximately equal size in both systems, caeteris paribus.
- *Number of supporting theories:*
  - Both Isabelle and Coq have rather large set of libraries containing formalised theories and data structures, that are being constantly replenished, see [1] and [2].

- *Expressiveness of syntax:*
  - Both systems have the built-in powerful functional languages, which can be used to define complex recursive structures;
  - Both systems accept forward proofs (written in imperative style as a sequence of tactics calls). This method may seem non-natural mathematically, as the search for proof is being performed "blindly", preserving the goal of the implicitly;
  - In contrast to Coq, the backward proof supported by Isabelle firstly states the target goal explicitly for every tactic (with keywords `show`, `have`, `assume`, etc.), so that the proof become much more readable, yet it requires more time to be written.
- *Usability of the syntax:*
  - Although Isabelle recognises common mathematical ASCII symbols in proof which makes it much more readable, it may seem inconvenient to use them within IDE (e.g., character $\forall$ is incoded as `\<forall>`, $\sum$ as `\<Sum>`, etc.);
  - The syntax of Coq is closer to the syntax of a programming language rather than mathematics, apparently it was designed for convenient work with a keyboard.
- *Usability of the native IDE:*
  - The authours are inclined to consider the Isabelle's jEdit Prover IDE more user-friendly as the whole proof is being recompiled every time user changes the syntax tree, which facilitates user to acquire the proof state for any arbitrary step of the proof;
  - In contrast, the CoqIDE can change the proof state backward and forward linearly, which however implies less system overload.
- *Additional comparison information:*
  - Coq has an essential feature that distincts it from most other theorem provers: it can extract the verified code for which compliance with the specification have been proved in a constructive way. This encourages using Coq as a software verification tool.

4 Future work

Although this paper does not pretend to give a fully exhaustive comparative analysis of two such complex systems as Coq and Isabelle, the authors hope it will help users without advanced background in mathematics to be involved into the work with proof assistants more quickly and easily. In future, this paper tends to be a foundation for more advanced survey of automatic tools used in software verification.

References

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

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

[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] D. Griffioen and M. Huisman, "A comparison of pvs and isabelle/hol," *Theorem Proving in Higher Order Logics*, pp. 123–142, 1998.

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

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

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

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

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

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

[14] 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.

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

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

[17] B. C. Pierce, *Types and programming languages*. MIT press, 2002.

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

[19] "HOL, interactive theorem prover." `https://hol-theorem-prover.org/`.

[20] P. Letouzey, "Extraction in Coq: An overview," *Logic and Theory of Algorithms*, pp. 359–369, 2008.

[21] 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.

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

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

[24] V. Glivenko, "Sur quelques points de la logique de m. brouwer," *Bulletins de la classe des sciences*, vol. 15, no. 5, pp. 183–188, 1929.

[25] A. Kolmogorov, "O principe tertium non datur (in russian)," *English trans. in van Heijenoort [1967, 414-437], from Mathematicheskiy sbornik 32: 646–667*, 1925.

[26] P. Rush, *The Metaphysics of Logic*. Cambridge University Press, 2014.

[27] J. Harrison, *Handbook of practical logic and automated reasoning*. Cambridge University Press, 2009.

Appendices

A.1 Basic type definitions

```
(* In Coq, False is an unobservable proposition, which
   is defined as a propositional type without constructor *)
Inductive False : Prop := .

(* On the other hand, True is defined as always true proposition *)
Inductive True : Prop := I : True.
```

Figure 5: Basic `Prop` types definitions in Coq

```
(* boolean type is defined as simple enumeration: *)
Inductive bool : Set :=
    true  : bool | false : bool

(* Similartly to the False, an empty set is a Set without type constructor: *)
Inductive Empty_set : Set := .
```

Figure 6: Basic `Set` types definitions in Coq

```
datatype nat =
    zero ("0") |
    Suc nat
```

Figure 7: Definition of Peano's natural numbers type `nat` in Isabelle

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

Figure 8: Definition of Peano's natural numbers type `nat` in Coq

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

Figure 9: Definition of addition over `nat` in Isabelle

```
Fixpoint add (n m: nat) : nat :=
    match n with
        | O => m
        | S n' => S (n' + m)
    end
where "n + m" := (add n m) : nat_scope.
```

Figure 10: Definition of addition over `nat` in Coq

A.2 Example proof of double-negated classical tautology in Coq

```
Lemma DoubleNegatedExcludedMiddle_Coq: forall P: Prop,
    ¬¬(P \/ ¬P).
Proof.
    (* 'tauto' automatically proves the statement *)
    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.
```

Figure 11: Proof of the double-negated excluded middle in Coq

A.3 Example proofs of de Morgan's laws in propositional logics

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

  (* 'apply blast' automatically solves the equation *)
  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
```

Figure 12: Proof of the de Morgan's law for propositions in Isabelle

```
Theorem DeMorganPropositional_Coq:
    forall P Q : Prop, ¬(P \/ Q) <-> ¬P /\ ¬Q.
Proof.
  (* 'tauto' automatically proves the equation *)
  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 13: Proof of the de Morgan's law for propositions in Coq

```
(* 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.
    try tauto.  (* automatic tactic fails here *)
    intros a b.
    destruct a; simpl; reflexivity.
Qed.
```

Figure 14: Proof of the de Morgan's law for booleans in Coq

A.4 Example proofs of first-order quantified de Morgan's laws

```
lemma DeMorganQuantified_Isabelle⁵:
  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
```

Figure 15: Proof of the de Morgan's law for first-order propositions in Isabelle

```
Require Import Coq.Logic.Classical_Prop.

Lemma DeMorganQuantified_Coq: forall (P : Type -> Prop),
    ¬ (forall x : Type, P x) -> exists x : Type, ¬ P x.
Proof.
    unfold not.
    intros P H_notall.
    apply NNPP.  (* apply classic rule ¬¬P ==> P *)
    unfold not. intro H_not_notexist.
    cut (forall x:Type, 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.
```

Figure 16: Proof of the de Morgan's law for first-order propositions in Coq

[5]This proof was originally taken from the set of examples in Isabelle's documentation, see https://github.com/seL4/isabelle/blob/master/src/HOL/Isar_Examples/Drinker.thy

A.5 Example of higher-order statement definitions

```
lemma lem:
  "∀ (f::bool⇒bool) (b::bool) .
  f (f (f b)) = f b"
```

Figure 17: Higher-order statement definition in Isabelle

```
Lemma lem:
    forall (f : bool -> bool) (b : bool),
    f (f (f b)) = f b.
```

Figure 18: Higher-order statement definition in Coq

A.6 Example proofs of the formula for sum of first n numbers using inductive types

```
fun range_sum :: "nat ⇒ nat"
  where "range_sum n = (∑k::nat=0..n . k)"
value "range_sum 10"  (* check the function *)

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 "... + 2 * (n + 1) = (n + 1) * (n + 2)" by simp
  finally show "2 * (range_sum (Suc n)) = (Suc n) * (Suc n + 1)" by simp
qed
```

Figure 19: Proof of the formula for sum of n first number in Isabelle

```
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.  (* output: '= 6 : nat' *)

Lemma range_sum_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.
    (* goal: '2 * range_sum 0 = 0 * (0 + 1)' *)
    - simpl; reflexivity.
    (* goal: '2 * range_sum (S n) = S n * (S n + 1)' *)
    - rewrite -> Nat.mul_add_distr_l. (* '2*range_sum(S n) = S n * S n + S n * 1' *)
      rewrite -> Nat.mul_1_r.         (* '2*range_sum(S n) = S n * S n + S n' *)
      rewrite -> (Nat.mul_succ_l n).  (* '2*range_sum(S n) = n * S n + S n + S n' *)
      rewrite <- (Nat.add_1_r n).     (* '2*range_sum(n+1) = n*(n+1)+(n+1)+(n+1)' *)
      rewrite -> range_sum_lemma.     (* '2*(range_sum(n)+(n+1)) = n*(n+1)+(n+1)+(n+1)' *)
      omega.                          (* automatically solve arithmetic equation *)
Qed.
```

Figure 20: Proof of the formula for sum of n first number in Coq