Filip Marić

A SURVEY OF INTERACTIVE THEOREM PROVING

Abstract. Fully formally verified mathematics and software are long-standing aims that became practically realizable with modern computer tools. Reasoning can be reduced to several basic logical principles, and performed using specialized software, with significant automation. Although full automation is not possible, three main paradigms are represented in formal reasoning tools: (i) decision procedures for special classes of problems, (ii) complete, but potentially unterminating proof search, (iii) checking of proof-sketches given by a human user while automatically constructing simpler proof steps. In this paper, we present a survey of the third approach, embodied in modern *interactive theorem provers (ITP)*, also called *proof-assistants*. These tools have been successfully developed for more than 40 years, and the current state-of-the-art tools have reached maturity needed to perform real-world large-scale formalizations of mathematics (e.g., Four-Color Theorem, Prime Number Theorem, and Feith-Thompson's Odd Order theorem) and software correctness (e.g., substantial portions of operating systems and compilers have been verified). We discuss history of ITP, its logical foundations, main features of state-of-the-art systems, and give some details about the most prominent results in the field. We also summarize main results of the researchers from Serbia and personal results of the author.

Mathematics Subject Classification (2010): Primary: 03-02, 03B35, 68T15;

Keywords: interactive theorem proving; proof assistants; formal logic

Matematički fakultet, Beograd, Serbia
filip@matf.bg.ac.rs

Contents

1. Introduction

Mathematics is deductive and proofs lie in its very heart. Exposing it as a formal theory where all statements are proved using just a few basic assumptions and inference rules is a long-standing goal. The problem with this approach is that fully *formal proofs* involve many steps and it requires a tremendous level of effort to construct them, write them precisely all the way down to the axiomatic level, and carefully check them. Therefore, the common practice usually considers only *proof-sketches*–approximations of real, formal proofs. A proof-sketch is considered to be rigorous enough when both its author and the reader agree that a formal proof can, at least in principle, be established based on it. This undertake does not need to be easy–it could be tedious and time-consuming, but it must be straightforward and must not require complex reasoning. The peer-review process, central for publishing and disseminating mathematical results, is centered around a human reviewer who checks proof-sketches given by the author. The reviewer tries to convince himself that these do not contain errors and cover all important cases, so that they could be (at least in principle) converted into a formal proof. Unfortunately, this can be error-prone.

There is a long tradition of errors in mathematical literature, and some of them are summarized by Avigad and Harrison [9]. The book *"Erreurs de mathématiciens"* written by Lecat in 1935, gives a survey of errors made by major mathematicians up to 1900. Grcar describes errors in the contemporary mathematical literature and laments the fact that corrections are not published as often as they should be [53]. Considering the abundance of theorems being published today, one fears that a project like the Lecat's one would be practically impossible today. Some false results in mathematics went undetected for long periods of time. Errors can often be easily fixed, but that need not always be the case. For example, it took Wiles's and Taylor, close to a year to find a way to circumvent the error found in Wiles's first proof of Fermat's Last Theorem [150, 137]. Around 1980, Mason and Gorenstein announced that the classification of finite simple groups had been completed, unaware there was a gap that was not filled until 2001 and doing so required a 1,221-page proof by Aschbacher and Smith [3].

These examples show that classic peer-review process in mathematics has many weaknesses and that reviewers must be very careful when examining complex mathematical results, as, in spite of the reputation of the authors, there could be significant imprecisions or even flaws in the manuscripts. Reviewers can have a very hard task. A famous case happened when a panel of 12 referees appointed by the Annals of Mathematics studied the proof of the Kepler Conjecture given by Hales and Ferguson for four full years (from 1998 to 2002), finally returning with the verdict that they were "99% certain" of the correctness [113, 9]. The proof consisted of 300 pages of mathematics and calculations performed by approximately 40,000 lines of computer code, and referees claimed that they had run out of energy to devote to the problem. The proof was finally published in 2005 [60].

Fortunately, computer science gave us a new technology that might change the way that mathematics is done and verified. During the second half of the 20th century, the field of *computer supported theorem proving* arose. Programs that help to prove theorems come in several flavors. *Automated theorem provers (ATP)* are used to establish the truth (or even to construct proofs) of mathematical statements, fully automatically. They either implement specialized decision procedures for some limited fragments of logic and mathematics (e.g., SAT/SMT solvers [16], automated provers for geometry based on Wu's method or Gröbner bases [152, 83]), or are general enough to cover wider ranges of mathematics (e.g., resolution or tableaux-based theorem provers [127]), but then they must be incomplete. On the other hand, *interactive theorem provers (ITP)*, sometimes called *proof-assistants*, have more modest goals. Instead of trying to construct proofs fully automatically, they heavily rely on user guidance. User cooperates with the prover, specifying the overall proof-structure and giving important hints, while the computer checks the proof, filling in steps that it can prove automatically. This approach gives quite promising results, and significant results in both classical mathematics and computer science have been fully formally verified lately using interactive theorem provers.

In the rest of the paper, we will briefly describe history of ITP (Section 2), their logical foundations (Section 3), and their most important, state-of-the-art features (Section 4). Then we will present several prominent results currently achieved in the field of ITP (Section 5). Finally, we will give an overview of ITP in Serbia, and present main results obtained by the author (Section 6).

2. A Brief History of ITP

The history of interactive theorem proving builds upon the history of mathematical deduction, theorem proving and formal logic.

Early history of formal deduction. The most prominent early example of mathematical deduction are the Euclid's "*Elements of Geometry*", written ca. 300 BC. Although they served as an example for rigorous argumentation for more than two millennia, through the ages, the proofs became more rigorous, and the language of mathematics became more precise and more symbolic. In the 17th century, *Leibniz* proposed to develop a universal symbolic language (*characteristica universalis*) that could express all statements (not just in mathematics) and a calculus of reasoning (*calculus ratiocinator*) for deciding the truth of assertions expressed in the characteristica. Every dispute could be reduced by calculations–disputants would just take pencils and say to each other *calculemus*–let us calculate.

Around the turn of 20th century, motivated by Leibniz's aims, *Frege* presented a formal system sufficient to express all mathematical arguments. Within the system, all proofs can be expressed by using just several axioms and inference rules, and their correctness could (at least in principle) be verified mechanically. Although the system was later shown to be inconsistent, it marked the beginning of symbolic logic and formalized mathematics. Very soon, other formal systems aiming to serve as a foundation of mathematics emerged, most famous one being "*Principia Mathematica*" developed by Russel and Whitehead. Russel discovered that Frege's system allowed the construction of paradoxical sets and in Principia they tried to avoid this problem by ruling out the creation of arbitrary sets. This was achieved by replacing the notion of a general set by notion of a hierarchy of sets of different *types*–a set of a certain type only allowed to contain sets of strictly lower types. Another approach to rule out paradoxes in set theory is *ZFC* (*Zermelo–Fraenkel* *set theory with the axiom of choice*) – an axiomatic set theory formulated in first-order logic that is today the most common foundation of mathematics. Principia and ZFC were so comprehensive that there was a common belief that it would be possible to reduce the whole mathematics to a few axioms and rules of inference.

In the mid-1930s, a group of mathematicians writing under the collective pseudonym *Bourbaki* adopted set theory as the foundation for a series of influential papers. Their goal was to provide a self-contained, rigorous, and general presentation of the core branches of mathematics (their work was foundational, although they did not believe that such approach would be possible in everyday mathematics). In his famous program, *Hilbert*, the most prominent advocate of formalism in mathematics, proposed to ground all existing theories to a finite set of axioms, and to prove that these were consistent (no contradiction could be derived), complete (all statements that are true could be proved in the formalism), and decidable (there should be an algorithm for deciding the truth or falsity of any statement). Consistency of more complicated systems, could be proven in terms of simpler systems, and ultimately, the consistency of all of mathematics could be reduced to arithmetic.

However, in 1931 Gödel showed that most Hilbert's program goals were impossible to achieve. First Gödel's *incompleteness theorem* states that arithmetic is incomplete–there cannot be a consistent system of axioms whose theorems can be effectively listed, capable of proving all truths about the natural numbers–there will always be statements that are true, but not provable within the system. The second incompleteness theorem, an extension of the first, shows that such a system cannot demonstrate its own consistency. Soon after, Church and Turing independently proved the *undecidability of first-order logic* (i.e., that the Hilbert's *Entscheidungsproblem* is impossible, assuming their definitions of the notion of algorithm). Although these theoretical limitations buried the formalists' dreams, it seems that they did not and do not affect any everyday mathematical results.

Early results in interactive theorem proving. Computer assisted theorem proving emerged in 1960s. Ideas of having computer programs for checking mathematical proofs can be attributed to McCarthy, who has written that "Checking mathematical proofs is potentially one of the most interesting and useful application of automatic computers" [103].

The first notable interactive theorem provers were *SAM (semi-automated mathematics)* series [54] (SAM V solved an open problem in lattice theory).

In the late 1960s, *de Bruijn* designed the language *Automath* [36, 110] for expressing mathematical theories in such a way that a computer can verify the correctness. Computer assistance in constructing proofs was minimal, and emphasis was on a compact, efficient notation for describing mathematical proofs. A proof assistant satisfies the *de Bruijn's criterion* if it has a proof checker that is small enough to be verified by hand—proof assistants should be able to store proofs, so that they can be independently checked by such simple checkers. Automath proofs allowed a checker coded in only 200 lines of code. The capabilities of Authomath were demonstrated during 1970s by Jutting [80] who formalized E. Landau's textbook "*Grundlagen der Analysis*", in spite of computer power limitations and lack of software support and text editing facilities. Automath introduced many notions that were later adopted or reinvented. It was the first practical system that exploited the Curry-Howard correspondence (although de Brujin was not aware of Howard's work) which serves as a basis of many modern proof-assistants based on type-theory. Therefore, Automath can be seen as their predecessor (although it was not much publicized and never reached wide-spread use). De Bruijn's name is today also connected to *de Bruijn indices* for variable binding—representation of λ-calculus terms by using indices instead of variable names, thus eliminating the need for renaming i.e., α-conversion (e.g., the term $\lambda x.\ \lambda y.\ y\,x$ is represented by $\lambda\,\lambda\,2\,1$). Also, the *de Bruijn factor* measures the ratio between the size of a formal proof and its informal counterpart. Smaller this ratio is, less human effort is needed to do the formalization, and the prover is considered to be better.

In the early 1970s, Boyer and Moore begun work on *Nqthm*—a fully automatic theorem prover that used LISP as the working logic and provided automation of mathematical induction [22]. In mid 1970s, Kaufmann added user guidance to the system. The interactive system was successfully used to formalize many significant results, both in mathematics (e.g., Shankar formalized Gödel's incompleteness theorem [130]) and computer science. The authors were awarded ACM Software System award for 2005. Nqthm evolved into *ACL2* (A Computational Logic for Applicative Common Lisp)–an industrial strength ITP system that is still actively developed and used. Unlike most ITP systems used today that use higher-order logics, Nqthm and ACL2 use first-order logic.

In the late 1970s, Trybulec developed *Mizar* [138]. Mizar proof checker is still actively being used and the body of mathematics formally built up in Mizar, known as the "*Mizar Mathematical Library*" (MML) and published in the *Journal of Formalized Mathematics* still seems unrivaled in any other system. Mizar proofs are presented in a declarative, human-readable form.

LCF systems. In 1972, at Stanford, Milner begun implementing *LCF*—an interactive proof-checker for *"Logic of Computable Functions"*, a logic devised by Scott in 1969, but not published until 1993 [129]. LCF was convenient for reasoning about program semantics, and about computable functions over integers, lists, and similar domains. The system introduced backward proof style—user enters a theorem to be proved in the form of a *goal* and applies *tactics* that transform goals into simpler subgoals. Subgoals are either discharged using a simplifier or split into simpler subgoals until they can be discharged. *Tacticals* are used to combine tactics and support automatic proof search (e.g., repeatedly apply Rule X and then apply either Rule Y or Rule Z). Milner moved to Edinburgh and around 1977 his group developed a programmable proof checker, latter dubbed *Edinburgh LCF.* It demonstrated how to implement advanced proof search procedures in a safe way, so that the soundness of the whole system depends just on a very small kernel that implements only the axioms and basic rules of the logic. To make that possible, the whole system was implemented in a programmable *meta-language ML*—a functional language specially designed for that purpose. *Cambridge LCF* [120] developed by Paulson (in collaboration with Huet) around 1985 brought a dramatically improved implementation and design. It introduced full predicate logic, and was equipped with a comprehensive set of tactics and tacticals, implementing advanced proof tools (e.g., simplification, rewriting). Principles introduced by LCF were soon applied to proof-assistants for more conventional logics (e.g., for higher order logic–HOL). Subsequently, many programmable proof checkers were designed on the LCF principles or, at least, used some LCF features (e.g., HOL, Isabelle, Coq).

Building upon Cambridge LCF, Gordon implemented *HOL* [50, 51] with the purpose of hardware verification. The system has always been very open: many people contributed to its development, and several groups built their versions of the system, essentially starting from scratch. Original implementation is now referred to as *HOL/88*, and the systems it has influenced were *HOL90*, *HOL98*, *HOL4*, *HOL Zero*, *ProofPower*, *Isabelle/HOL*, and *HOL Light*.

HOL Light[1] [68, 74] is a rework of HOL done by Harrison and Slind. It is implemented in OCaml and has a simpler logical core compared to other HOL variants. Its main features are many automated tools and a vast library of mathematical theorems (e.g. arithmetic, basic set theory and real analysis).

Modern type theory. A cornerstone in the development of ITP systems is the discovery of *Curry-Howard correspondence* between typed λ-calculus and natural deduction [76]. The paradigm has tight connections to intuitionism and constructivism in mathematics, as introduced by Brower and Heyting. After its discovery, it served as a basis of *modern type theory*, and led to a new class of formal systems and calculi designed to act both as proof systems and as typed functional programming languages. Along simple types, these calculi support dependent types (types depending on values) making it possible to encode many logical propositions as types so that provability of a formula in the original logic reduces to a type inhabitation problem in the underlying type theory. Proofs are regular objects and properties of proofs can be stated the same way as of any program. Two most significant calculi of this kind are the *Intuitionistic Type Theory* introduced by Martin-Löf in 1970s and refined during 1980s [101, 115] and *Calculus of Constructions (CoC)* introduced by Coquand and Huet in the late 1980s [30]. Homotopy type theory [139] is a new branch of mathematics that combines aspects of several different fields in a surprising way. It is based on a recently discovered connection between homotopy theory and type theory. Modern type theory serves as a logical foundation of many state-of-the-art ITP systems.

The most successful ITP system (awarded by the ACM software system award for 2013) based on modern type theory is Coq[2]. The development of Coq has been initiated by Huet and Coquand. Today, the implementation team of more than 40 researchers has been coordinated by Huet, Paulin and Herbelin. Typical applications include the formalization of programming languages semantics, mathematics and teaching. Several very-large-scale formalizations have been done using Coq [46, 48, 90]. System has many extensions, most famous being the SSReflect [49] developed by Gonthier and used to prove the 4-color and odd-order theorem.

[1] https://code.google.com/p/hol-light/

[2] http://coq.inria.fr

Coq is based on the *Calculus of Inductive Constructions (CiC)* [15]—an extension of the calculus of constructions. The system is implemented mostly in OCaml and is centered around a small, trusted proof-checking kernel.

Nuprl[3] [29] is a proof-assistant founded by Bates and Constable at Cornell in 1979, and it is still actively developed. It provides logic-based tools to support programming and formal computational mathematics. It includes a programming language, but system is designed primarily for implementing mathematics. Nuprl uses a constructive logic called the *Computational Type Theory (CTT).* The system is implemented in ML, follows the LCF approach, and has a trusted kernel.

PVS: A Prototype Verification System[4] [118] is a language and mechanized environment for formal specification and verification actively developed since early 1990s at SRI International by Owre, Shankar, Rushby et al. It is based on a type theory and features a rich dependently typed higher-order logic. It provides a highly expressive specification language and powerful automated deduction. It combines specification, proof checking, and model checking and its typical applications are formalization of mathematics (e.g. analysis, graph theory, and number theory), and verification of hardware, sequential and distributed algorithms. It also serves a back-end verification tool for computer algebra and code verification systems.

Other modern ITP systems based on the type-theory include *Matita*[5] [5] by the Helm team lead by Asperti at the University of Bologna, and programming languages with dependent types such as *Agda*[6] [116] developed Norell and programming logic group at Chalmers and Gothenburg University, and *Epigram*[7] [102] developed by McBride based on joint work with McKinna.

Generic provers and logical frameworks. During 1980s, many specialized logics were developed, and LCF-style provers for each of them needed to be implemented. Implementing such provers from scratch or adapting existing code to different underlying logic was not easy and developers struggled to keep the pace. There was widespread concern that computer scientists could not implement logics as fast as logicians could conceive them [122]. So, people started to develop *generic provers* or *logical frameworks* that offer meta-language and meta-logic for formalizing different object-logics (and their deductive systems). In such frameworks, proof rules of object logics are described declaratively (in classic LCF systems, rules are represented as ML programs, i.e. they are implemented rather than specified) and provided proof procedures are applicable to a variety of object-logics.

In 1986, Paulson developed *Isabelle* [119, 121]–a generic theorem prover that implemented many theories (e.g., intuitionistic natural deduction, Constructive Type Theory, classical first-order logic, ZFC). Its most widespread instance nowadays is *Isabelle/HOL*[8] [114]—a well developed higher order logic theorem prover, developed by the groups led by Paulson at Cambridge, Nipkow at Münich, and Wenzel

[3] `http://www.nuprl.org/`

[4] `http://pvs.csl.sri.com/`

[5] `http://matita.cs.unibo.it/`

[6] `http://wiki.portal.chalmers.se/agda/`

[7] `https://code.google.com/p/epigram/`

[8] `http://isabelle.in.tum.de/`

at Paris. In 1980s, Felty et al. developed λ*Prolog* [39]–a version of Prolog that uses higher-order unification, used to formalize logics and theorem provers. Its logical basis is very close to Isabelle's. Harper, Honsell, and Plotkin formulated the *Edinburgh Logical Framework (LF)* [67]. It is based on type theory and inspired by Martin-Löf's work and the idea of propositions-as-types and dependently-typed $\lambda\Pi$-calculus. It is later implemented as the *Twelf*[9] system.

Further reading. Several survey papers covering ITP history, foundations, state-of-the-art and perspectives have been published. Hales analyzed current developments in formal proofs [65, 62]. In 2008, he edited a special issue of Notices of AMS on formal proofs [63]. Harrison (alone and with co-authors) published several survey papers on ITP [70, 73, 9, 71]. Wiedijk analyzed 17 different proof assistants and demonstrated their main features by showing proofs of irrationality of $\sqrt{2}$ in each of them [147]. He also maintains a list of 100 major theorems and their formal proofs[10]–currently, around 90% of them has been formalized [148]. Barendregt and Geuvers give a detailed description of proof-assistants, and their theoretical foundations [127, 12]. Chlipala discusses features of different state-of-the-art proof-assistants [25]. In 2005, in the Science magazine, Mackenzie introduces ITP to the wider audience [92]. History of LCF and HOL systems is described by Gordon [51]. Paulson presented a survey of earlier versions of the Isabelle system, its central ideas and its meta-logic [122]. Huet and Herbelin analyze 30 years of research and development around Coq [77].

3. Foundations of ITP

In this section we will briefly summarize some classic results of mathematical logic and type theory that serve as the foundation of interactive theorem provers.

3.1. Natural Deduction, λ-calculi, Curry-Howard Correspondence.

Gentzen's natural deduction, Church's λ-calculi and correspondence between them discovered by Curry and Howard serve as logical foundations of most proof-assistants.

3.1.1. Natural deduction. *Natural deduction* is a formal deductive system developed by Gentzen in 1930s [40, 41], aiming to have formal proofs similar to actual reasoning in traditional mathematical texts. Each logical connective is associated with two kinds of inference rules—introduction and elimination. Natural deduction style calculi are formulated for various logics (e.g., propositional/first-order/higher-order, intuitionistic/classical). There are also many notations used to represent natural deduction rules and proofs. Figure 1 shows some natural deduction rules for intuitionistic fragment of propositional logic. The sequent $\Gamma \vdash A$ denotes that A can be proved from the set of assumptions (the context) Γ. The formulation of rules given in Figure 1 uses explicit context and although not suitable for pen-and-paper proofs, it can be convenient for computer-assisted proving.
For example, the rule $\rightarrow_I$ says that to prove $\alpha \rightarrow \beta$ from the set of assumptions Γ, it suffices to prove β from Γ and the additional assumption α.

[9] http://twelf.org/

[10] http://www.cs.ru.nl/~freek/100/

$$\frac{\alpha \in \Gamma}{\Gamma \vdash \alpha}\ \mathrm{Ax} \qquad \frac{\Gamma \vdash \bot}{\Gamma \vdash A}\ \bot_E \qquad \frac{}{\Gamma \vdash \top}\ \top_I$$

$$\frac{\Gamma, \alpha \vdash \beta}{\Gamma \vdash \alpha \to \beta}\ \to_I \qquad \frac{\Gamma \vdash \alpha \to \beta \quad \Gamma \vdash \alpha}{\Gamma \vdash \beta}\ \to_E$$

$$\frac{\Gamma \vdash \alpha \quad \Gamma \vdash \beta}{\Gamma \vdash \alpha \wedge \beta}\ \wedge_I \qquad \frac{\Gamma \vdash \alpha \wedge \beta}{\Gamma \vdash \alpha}\ \wedge_{E1} \qquad \frac{\Gamma \vdash \alpha \wedge \beta}{\Gamma \vdash \beta}\ \wedge_{E2}$$

$$\frac{\Gamma \vdash \alpha}{\Gamma \vdash \alpha \vee \beta}\ \vee_{I1} \qquad \frac{\Gamma \vdash \beta}{\Gamma \vdash \alpha \vee \beta}\ \vee_{I2} \qquad \frac{\Gamma \vdash \alpha \vee \beta \quad \Gamma, \alpha \vdash \gamma \quad \Gamma, \beta \vdash \gamma}{\Gamma \vdash \gamma}\ \vee_E$$

$$\frac{\Gamma \vdash \alpha \quad \Gamma \vdash \neg\alpha}{\Gamma \vdash \bot}\ \neg_E \qquad \frac{\Gamma, A \vdash \bot}{\Gamma \vdash \neg A}\ \neg_I$$

FIGURE 1. Natural deduction rules

3.1.2. λ-calculi. There is a large number of formal systems that share the name *λ-calculi*. Investigation of such systems begun in late 1920s in the works of Church [28]. In the 1930s, the untyped λ-calculus was used for describing computations (algorithms) and to show the undecidability of first-order logic [26]. However, as a logical foundation, that calculus was shown to be inconsistent. Inconsistencies were eliminated if the calculus is extended with types [27], and typed λ-calculi serve as foundations of logics and programming languages. Such caculi are computationally weaker than the untyped λ-calculus, but are logically consistent.

The *simply typed λ-calculus* (sometimes denoted by $\lambda^{\to}$) supports basic types (from some set B) and function types, formed using the $\to$ constructor ($\tau ::= \tau \to \tau \,|\, T$, where $T \in B$). $x : \tau$ denotes that the variable x has the type τ. The syntax of the λ-terms (λ-expressions) is given by $e ::= x \mid \lambda x\colon \tau.\ e \mid e\ e$, so terms are either variables, λ-abstractions[11], or function applications. Each term has an associated type. Terms are typed with respect to a typing context (environment)—a set of typing assumptions of the form $x : \tau$. $\Gamma \vdash e : \tau$ denotes that the e is a term of type τ in context Γ. Types of terms are derived using the rules shown on Figure 2. For example, for any types τ and σ, $\vdash (\lambda x\colon \tau.\ x) : \tau \to \tau$ and $\vdash (\lambda x\colon \tau.\ \lambda y\colon \sigma.\ x) : \tau \to (\sigma \to \tau)$ can be derived. Functions of more than one variable are treated as *curried* functions. For example, instead of a function that has the type $\rho \times \sigma \to \tau$, a corresponding function with the type $\rho \to (\sigma \to \tau)$ is considered. The term $\lambda x\colon \rho.\ \lambda y\colon \sigma.\ f\ x\ y$ corresponds to a function of two variables x and y. Once the first argument is fixed, a function of a single variable is obtained (e.g., $\lambda y\colon \sigma.\ f\ e_x\ y$), and when both arguments are fixed, the final value is obtained (e.g., $f\ e_x\ e_y$).

$$\frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau} \qquad \frac{\Gamma, x : \tau \vdash e : \sigma}{\Gamma \vdash (\lambda x\colon \tau.\ e) : (\tau \to \sigma)} \qquad \frac{\Gamma \vdash e1 : \tau \to \sigma \quad \Gamma \vdash e_2 : \tau}{\Gamma \vdash e_1\ e_2 : \sigma}$$

FIGURE 2. Typing rules for simply typed λ-calculus

[11] Note that the variable in abstraction is explicitly given a type, i.e., abstraction is written as $\lambda x\colon \tau.\ e$. Contrary to this so-called Church-style typing, in the so-called Curry-style typing the type of the abstraction variable is not give explicitly, i.e., abstraction is written as $\lambda x.\ e$.

λ-calculus can be given semantics (in several ways). Basic types are usually interpreted as sets and function types by the set-theoretic function space. Abstractions represent functions—$\lambda x\colon \tau.\ e$ is a function of a variable x returning e (that might depend on x), and applications represent function applications.

λ-calculi involve reduction relations between terms, allowing to define an equational theory. The central role is given to the β-reduction (denoted by $\rightarrow_\beta$), corresponding to function application and defined by the relation $(\lambda x\colon \tau.\ e)e' \rightarrow_\beta e[e'/x]$, in the context Γ such that $\Gamma, x : \tau \vdash e : \sigma$ and $\Gamma \vdash e' : \tau$. Meta expression $e[e'/x]$ denotes the λ-term obtained from e by replacing all free-occurrences of variable x by the λ-term e'. For example, the term $\lambda x\colon \tau.\ x$ denotes the identity function of the type τ and it holds that $(\lambda x\colon \tau.\ x)e \rightarrow_\beta e$. λ-calculi feature other important reductions. For example, α-reduction allows to rename a bound variable (e.g. $\lambda x\colon \tau.\ x \rightarrow_\alpha \lambda y\colon \tau.\ y$). Extensionality (two functions are equal iff they give the same results for all arguments) is expressed through η-reduction defined by $(\lambda x\colon \tau.\ e)\ x \rightarrow_\eta e$ (in the context Γ such that $\Gamma \vdash e : \tau \rightarrow \sigma$ and x is not free in e). $\beta\eta$-equivalence is decidable (as β-reduction is strongly normalizing i.e., every reduction sequence terminates), but the unification for $\beta\eta$-equivalence is undecidable.

3.1.3. Curry–Howard Correspondence. The connection between (intuitionistic) proofs and computations is deep. First, Curry noticed a resemblance between types of combinators in combinatory logic (a model for computation, somewhat connected to the λ-calculus) and axioms of Hilbert-type systems for intuitionistic propositional logic. In 1969, Howard described a similar connection between natural deduction and the typed λ-calculus [76]. The connection is quite obvious if we compare, for example, the natural deduction system for a minimal logic with just the $\rightarrow$ connective, with the simply typed λ-calculus (Figure 3).

$$\frac{\alpha \in \Gamma}{\Gamma \vdash \alpha}\,\mathrm{Ax} \qquad\qquad \frac{x : \sigma \in \Gamma}{\Gamma \vdash x : \sigma}$$

$$\frac{\Gamma, \alpha \vdash \beta}{\Gamma \vdash \alpha \rightarrow \beta}\,{\rightarrow_I} \qquad\qquad \frac{\Gamma, x : \tau \vdash e : \sigma}{\Gamma \vdash (\lambda x\colon \tau.\ e) : (\tau \rightarrow \sigma)}$$

$$\frac{\Gamma \vdash \alpha \rightarrow \beta \quad \Gamma \vdash \alpha}{\Gamma \vdash \beta}\,{\rightarrow_E} \qquad\qquad \frac{\Gamma \vdash e_1 : \tau \rightarrow \sigma \quad \Gamma \vdash e_2 : \tau}{\Gamma \vdash e_1\ e_2 : \sigma}$$

Figure 3. Natural deduction vs simply typed λ-calculus

Propositions in the logic correspond to types in the λ-calculus, so, the correspondence is also called *propositions-as-types* or *formulae-as-types.* There is also a correspondence between (non-empty) types and (intuitionistic) tautologies. Proving a tautology corresponds to checking that its corresponding type is inhabited i.e., finding a term of that type. Such term corresponds to a proof, so the correspondence is also called *proofs-as-terms* (or *proofs-as-programs*). Proof-checking corresponds to type-checking, interactive theorem proving to interactive construction of a term of a given type, and normalization of proofs to evaluation ($\beta\eta$-reduction of λ-terms).

Under propositions-as-types the proving process is highly constructive. For example, to prove the implication $\alpha \rightarrow \beta$, one must construct a function that takes a term of the type that corresponds α (that term corresponds to a proof of α) and transforms it to a term (a proof) of the type that corresponds to β.

The correspondence between the minimal intuitionistic propositional logic (featuring only $\rightarrow$) and the simply typed λ-calculus extends to richer logics, but requires making extensions to λ-calculi. Gilezan and Likavec give a detailed overview of computational interpretations of several deductive systems (including classical and intuitionistic natural deduction and sequent-calculus for propositional logic) [43].

3.1.4. Barendregt's $\boldsymbol{\lambda}$ cube. Simply typed λ-calculus corresponds to the intuitionistic propositional logic. To obtain computational interpretation of richer logics (e.g., first-order or higher-order logics), richer type systems must be introduced. Barendregt created a three-dimensional hierarchy of λ-calculi, classifying them with respect to the relationship between types and terms in the system.

In normal functions, considered so far, *terms depend on terms* (e.g., the application term $e_1\ e_2$ depends on terms e_1 and e_2). For a fixed type τ, the term $\lambda x\colon \tau.\ x$ is of the type $\tau \rightarrow \tau$ and denotes the identity function on the type τ. However, to express the polymorphic identity function we need to introduce a type variable T, and look at the term $\lambda x\colon T.\ x$. This is a *term that depends on a type.* With the type variable T, the type $T \rightarrow T$ is a *type that depends on a type.* Finally, *types can depend on terms* (such types are called *dependent types*). For example, the type of n-tuples of type τ (the type τ^n) depends on a natural number n, which is a term.

Barendregt described all such type and term dependencies within a uniform system [11]. The main idea was to move the formation of types from the meta-level to the formal system itself (the idea comes from de Bruijn and AutoMath). Therefore, types themselves can be represented by terms, so they also must have a type. Let the constant $*$ denote the sort of all types. Then $\tau : *$ expresses that τ is a type (here τ is considered to be a term, $*$ its type, so $\tau : *$ is an ordinary typing judgment). For example, the meta-statement "if τ is a type then so is $\tau \rightarrow \tau$" now becomes a formal type derivation $\tau : * \vdash (\tau \rightarrow \tau) : *$. But then $*$ is also a term, so it must have a type. Putting $* : *$ would lead into inconsistencies, so the sort $\Box$ is introduced and $* : \Box$ holds. Then, we can consider the following examples. The normal function term $\lambda x\colon \tau.\ f\ x$ has some type $\tau \rightarrow \sigma$. The polymorphic function term $\lambda T\colon\ *\ .\ (\lambda x\colon T.\ x)$ has the type $\Pi T\colon\ *\ .\ (T \rightarrow T)$, where Π is a so-called dependent product operator. The type constructor term $\lambda T\colon\ *\ .\ T \rightarrow T$ has the sort $* \rightarrow *$. That is not a regular type, so $(* \rightarrow *) : \Box$. Finally, the term $\lambda n\colon \mathbb{N}.\ A^n$ has the kind $\mathbb{N} \rightarrow *$.

In the formal definition of the λ-cube, expressions (terms) are either variables (e.g., x), applications (of the form $e_1\ e_2$), λ-abstractions (of the form $\lambda x : A.\ e$), or Π-abstractions (of the form $\Pi x : A.\ B$). The rules shown on Figure 4 consider the set $S = \{*, \Box\}$, and variables s, s_1 and s_2 range over S.
Note that $\rightarrow$ is not explicitly present in the system. The more general operator Π subsumes it, and $A \rightarrow B$ is just an abbreviation for $\Pi x\colon A.\ B$ when x is not free in B. The (s_1, s_2) rule is a schema that can be instantiated in 4 different ways ((s_1, s_2) can be $(*, *)$, $(\Box, *)$, $(*, \Box)$, or $(\Box, \Box)$). The instance $(*, *)$ will always be included, while including any of the three remaining instances gives rise to 8 different calculi. Including only the rule $(*, *)$ gives a simply typed λ-calculus $\lambda^{\rightarrow}$.

$$\frac{}{\vdash * : \Box}\ \text{axiom} \qquad \frac{\Gamma \vdash A : s}{\Gamma, x : A \vdash x : A}\ \text{start} \qquad \frac{\Gamma \vdash A : B \quad \Gamma \vdash C : s}{\Gamma, x : C \quad A : B}\ \text{weakening}$$

$$\frac{\Gamma \vdash F : (\Pi x\colon A.\ B) \quad \Gamma \vdash a : A}{\Gamma \vdash Fa : B[x/a]}\ \text{application} \qquad \frac{\Gamma, x : A \vdash b : B \quad \Gamma \vdash (\Pi x\colon A.\ B) : s}{\Gamma \vdash (\lambda x\colon A.\ b) : (\Pi x\colon A.\ B)}\ \text{abstraction}$$

$$\frac{\Gamma \vdash A : s_1 \quad \Gamma, x : A \vdash B : s_2}{\Gamma \vdash (\Pi x\colon A.\ B) : s2}\ (s_1, s_2) \qquad \frac{\Gamma \vdash A : B \quad \Gamma \vdash B' : s \quad B \to_\beta B'}{\Gamma \vdash A : B'}\ \text{conversion}$$

Figure 4. λ-cube rules

The rule instance $(\Box, *)$ gives rise to the second order λ-calculus $\lambda 2$ (studied in Girard's system F [44]) that includes polymorphism and terms that depend on types. For example, since $* : \Box$, in $\lambda 2$ it is possible to derive $\vdash (\Pi\tau : *.\ (\tau \to \tau)) : *$. That type is inhabited, since $\vdash (\lambda\alpha : *.\ \lambda a : \alpha.\ a) : (\Pi\tau : *.\ (\tau \to \tau))$.

The instance $(\Box, \Box)$ gives rise to the calculus $\lambda\underline{\omega}$ and type constructing terms. For example, as $(\tau \to \tau) : *$, it is possible to derive $\vdash (\lambda\tau : *.\ \tau \to \tau) : (* \to *)$.

Finally, $(*, \Box)$ gives rise to calculus $\lambda\Pi$ (sometimes also printed as λP). That calculus is similar to the Edinburgh Logical Framework (LF) [67]. $\lambda\Pi$ contains dependent types. For example, in $\lambda\Pi$ it is possible to derive $A : * \vdash (A \to *) : \Box$, or $A : *, P : (A \to *) \vdash (\Pi a : A.\ (P\,a \to P\,a)) : *$. The last derivation states that if A is a type, and P is a predicate on that type, then $\Pi a : A.\ (P\,a \to P\,a)$ is a type. Under propositions-as-types, that type corresponds to the proposition $\forall a \in A.\ P\,a \to P\,a$ (so $\to$ corresponds to implication while Π corresponds to universal quantification). A proof of that proposition is the term $\lambda a : A.\ \lambda x : P\ a.\ x$, since it can be shown that $A : *, P : (A \to *) \vdash (\lambda a : A.\ \lambda x : P\ a.\ x) : (\Pi a : A.\ (P\,a \to P\,a))$.

Interesting combinations are $\lambda\omega$ that combines $\lambda 2$ and $\lambda\underline{\omega}$, and $\lambda\Pi\omega$ (also known as λC or the Calculus of Constructions, introduced by Coquand and Huet [30]) that is on the top of the hierarchy and includes all the rules. All calculi of the cube share nice features, such as strong-normalization and subject reduction. All systems have decidable type-checking, while simpler type systems like $\lambda^{\to}$ and $\lambda 2$ also have decidable type-inference (using Hindley-Milner type-inference algorithm [106]).

In Curry–Howard isomorphism, $\lambda^{\to}$ corresponds to propositional logic, $\lambda 2$ to the second order propositional logic, $\lambda\omega$ to the higher order proposition logic, $\lambda\Pi$ to the predicate logic, and λC to the higher order predicate logic. For example, in λC an operator $\forall \equiv \lambda A\colon *\ .\ \lambda P\colon A \to *.\ \Pi a\colon A.\ P\,a$ can be defined and it has the role of the universal quantifier (it takes a type and a predicate on that type, and returns a predicate corresponding to the proposition that the predicate holds for every element of that type). It holds that $\forall\ A\ P \to_\beta \Pi a\colon A.\ P\,a$ and $A : *, P : (A \to *) \vdash \forall\ A\ P : *$.

The λ-cube systems only consider implication and universal quantification. It is possible to introduce new type forming operators that would correspond to other intuitionistic logical connectives. For example, in Martin-Löf's system [101] function types $\to$ correspond to the implication, product types $A \times B$ to the conjunction, disjoint union types $A + B$ to the disjunction, Π-types (dependent product types) to the universal, and Σ-types (dependent sum types) to the existential quantification.

3.2. Examples of proof-assistant foundations. In this section we will present logical foundations of HOL Light, Isabelle, and Coq.

3.2.1. HOL Light. One of the most widely used provers of the HOL family [50] is John Harrison's HOL Light [68, 74], with a quite minimalistic foundation. Its logic is a simple type theory with polymorphic type variables. The terms are those of simply typed lambda calculus, with just two primitive types: `bool` (Booleans) and `ind` (individuals). Equality ($=$) with polymorphic type $\alpha \to \alpha \to \mathit{bool}$ is the only predefined logical constant. HOL Light determines the deducibility of one-sided sequents $\Gamma \vdash p$ where p is a Boolean term, and Γ is a set of Boolean terms, with respect to its inference rules (Figure 5, the usual provisos are assumed—e.g., in MK_COMB type must agree, and in ABS x must not be free in any of the assumptions Γ).

$$\frac{}{\{p\} \vdash p}\ \text{ASSUME} \qquad \frac{}{\vdash t = t}\ \text{REFL} \qquad \frac{\Gamma \vdash s = t \qquad \Delta \vdash t = u}{\Gamma \cup \Delta \vdash s = u}\ \text{TRANS}$$

$$\frac{\Gamma \vdash s = t}{\Gamma \vdash (\lambda x.\ s) = (\lambda x. t)}\ \text{ABS} \qquad \frac{}{(\lambda x.\ t)x = t}\ \text{BETA} \qquad \frac{\Gamma \vdash s = t \qquad \Delta \vdash u = v}{\Gamma \cup \Delta \vdash s(u) = t(v)}\ \text{MK\_COMB}$$

$$\frac{\Gamma \vdash p \Leftrightarrow q \qquad \Delta \vdash p}{\Gamma \cup \Delta \vdash q}\ \text{EQ\_MP} \qquad \frac{\Gamma \vdash p \qquad \Delta \vdash q}{(\Gamma \setminus \{q\}) \cup (\Delta \setminus \{p\}) \vdash p \Leftrightarrow q}\ \text{DEDUCT\_ANTISYM}$$

$$\frac{\Gamma[x_1, \dots, x_n] \vdash p[x_1, \dots, x_n]}{\Gamma[t_1, \dots, t_n] \vdash p[t_1, \dots, t_n]}\ \text{INST} \qquad \frac{\Gamma[\alpha_1, \dots, \alpha_n] \vdash p[\alpha_1, \dots, \alpha_n]}{\Gamma[\gamma_1, \dots, \gamma_n] \vdash p[\gamma_1, \dots, \gamma_n]}\ \text{INST\_TYPE}$$

FIGURE 5. HOL Light inferrence rules

The constant definition rule allows to introduce new constant c and an axiom $\vdash c = t$ (subject to some conditions on free variables and polymorphic types in t, and provided that c is a fresh name). This enables defining other usual logical binders (e.g., $\top = ((\lambda p.\ p) = (\lambda p.\ p))$, $\wedge = (\lambda p.\ \lambda q.\ (\lambda f.\ f\ p\ q) = (\lambda f.\ f\ \top\ \top))$, $\forall = (\lambda P.\ P = \lambda x.\ \top))$. $\forall x.P$ is the abbreviation for $\forall(\lambda x.P)$. There are also 3 mathematical axioms present in the system. Extensionality ETA_AX: $\vdash (\lambda x.\ tx) = t$, choice SELECT_AX: $Px \Rightarrow P((\varepsilon)P)$, where ε is Hilbert's choice operator of type $(\alpha \to \mathit{bool}) \to \alpha$, and infinity INFINITY_AX implying that the type `ind` is infinite.

3.2.2. Isabelle. Isabelle [121, 122] is a generic theorem prover providing a meta-logic (originally denoted by $\mathcal{M}$, and later as Isabelle/Pure) for encoding various object logics. We only briefly describe its logical foundations (for its everyday use, the reader can consult tutorials for its most developed instance Isabelle/HOL [114]).

Isabelle's meta-logic is the fragment of simple type theory, including basic types (introduced by object logics) and function types (denoted by $\sigma \Rightarrow \tau$). The terms are those of the typed λ-calculus (constants introduced by object logics, variables, abstractions, applications). The only type defined by the meta-logic is *prop*—the type of propositions. Meta-logic formulas are terms of type *prop*. The meta-logic supports implication ($\Longrightarrow :: \mathit{prop} \Rightarrow \mathit{prop} \Rightarrow \mathit{prop}$), the polymorphic universal quantifier ($\bigwedge :: (\sigma \Rightarrow \mathit{prop}) \Rightarrow \mathit{prop}$), and polymorphic equality ($\equiv :: \sigma \Rightarrow \sigma \Rightarrow \mathit{prop}$). Non-standard symbols leave the standard ones free for object-logics. The inference rules of the meta-logic are given in natural-deduction style (Figure 6, standard provisos apply and $\phi[b/x]$ denotes substitution of the variable x with the term b).

$$\dfrac{\begin{array}{c}[\phi]\\ \psi\end{array}}{\phi \Longrightarrow \psi}\ (\Longrightarrow I) \qquad \dfrac{\phi \Longrightarrow \psi \qquad \phi}{\psi}\ (\Longrightarrow E) \qquad \dfrac{\phi}{\bigwedge x.\phi}\ (\textstyle\bigwedge I) \qquad \dfrac{\bigwedge x.\phi}{\phi[b/x]}\ (\textstyle\bigwedge E)$$

$$a \equiv a\ (\text{refl}) \qquad \dfrac{a \equiv b}{b \equiv a}\ (\text{sym}) \qquad \dfrac{a \equiv b \qquad b \equiv c}{a \equiv c}\ (\text{trans})$$

$$\dfrac{\begin{array}{c}[\phi]\\ \psi\end{array} \qquad \begin{array}{c}[\psi]\\ \phi\end{array}}{\phi \equiv \psi}\ (\equiv I) \qquad \dfrac{\phi \equiv \psi \qquad \phi}{\psi}\ (\equiv E)$$

$$(\lambda x.a) \equiv (\lambda y.a[y/x])\ (\alpha) \qquad ((\lambda x.a)(b)) \equiv a[b/x]\ (\beta) \qquad \dfrac{f(x) \equiv g(x)}{f \equiv g}\ (\eta)$$

$$\dfrac{a \equiv b}{(\lambda x.a) \equiv (\lambda x.b)}\ (\text{abs}) \qquad \dfrac{a \equiv b \qquad f \equiv g}{f(a) \equiv g(b)}\ (\text{app})$$

Figure 6. Isabelle's meta-logic inference rules

Object-logics are represented by introducing types, constants, and axioms. For example, encoding a fragment of first-order logic (FOL) introduces the type *term* for FOL terms, *form* for FOL formulas, and the constants $\longrightarrow :: form \Rightarrow form \Rightarrow form$ for object-level implication, and $\forall : (term \Rightarrow form) \Rightarrow form$, for object-level universal quantification. To connect the object-level with the meta-level truth, a meta-predicate $true :: form \Rightarrow prop$ is introduced ($[[A]]$ is a shorthand for $true(A)$). Then, the natural deduction rules of FOL can be encoded in the meta-logic by the meta-level formulas (rules) shown in Figure 7, which are given as axioms.

$$\begin{gathered}\textstyle\bigwedge AB.([[A]] \Longrightarrow [[B]]) \Longrightarrow [[A \longrightarrow B]]\\ \textstyle\bigwedge AB.[[A \longrightarrow B]] \Longrightarrow [[A]] \Longrightarrow [[B]]\\ \textstyle\bigwedge F.(\bigwedge x.[[F(x)]]) \Longrightarrow [[\forall x.F(x)]]\\ \textstyle\bigwedge Fy.[[\forall x.F(x)]] \Longrightarrow F(y)\end{gathered}$$

Figure 7. An object-logic representation in Isabelle's meta-logic

Goals are represented in the meta-logic the same way as rules, so proven goals become new rules. Proof construction combines rules, unifying the conclusion of one rule with a premise of another. Due to presence of λ-abstractions, higher-order unification must be used. Basic meta-level inferences (implemented by tactics) are resolution, and assumption (Figure 8). For details, see Paulson [122, 121, 119].

$$\dfrac{\begin{array}{l}\text{rule: } \mathbf{A\,a} \Longrightarrow \mathbf{B\,a}\\ \text{goal: } (\textstyle\bigwedge \mathbf{x}.\ \mathbf{H\,x} \Longrightarrow \mathbf{B'\,x}) \Longrightarrow C\\ \text{unifier: } (\lambda \mathbf{x}.\ \mathbf{B\,a\,x})\,\theta = \mathbf{B'}\,\theta\end{array}}{(\textstyle\bigwedge \mathbf{x}.\ \mathbf{H\,x} \Longrightarrow \mathbf{A\,a\,x})\,\theta \Longrightarrow C\,\theta}\ \text{res} \qquad \dfrac{\begin{array}{c}\text{goal: } (\textstyle\bigwedge \mathbf{x}.\ \mathbf{H\,x} \Longrightarrow A\,\mathbf{x}) \Longrightarrow C\\ \text{unifier: } A\,\theta = H_i\,\theta\ (\text{for some } H_i)\end{array}}{C\,\theta}\ \text{ass}$$

Figure 8. Resolution and assumption in Isabelle

3.2.3. Coq—Calculus of Inductive Constructions. Next, we will briefly present the *Calculus of Constructions (CoC)* [30], and its extension, the *Calculus of Inductive Construction (CiC)* that is a basis of the proof assistant Coq. CoC is on the very top of Barendregt's λ-cube—it is a typed λ-calculus, and it does not make a syntactic distinction between types and terms. Each term has a type, so types themselves have a type which is called a sort. Although the λ-cube contains only $*$ as the sort of types, Coq distinguishes two such sorts: *Prop* is the sort of logical propositions (a proposition denotes the class of terms representing its proofs), and *Set* is the sort of small sets, including Booleans, naturals, and products, subsets, and function types of these data types. Therefore, Coq makes distinction between terms that represent proofs, and terms that represent programs. *Type* is a higher order sort (that corresponds to $\square$ in the λ-cube). The sort of *Prop* and *Set* is *Type* (corresponding to $* : \square$) and the sort of *Type* is *Type* (however, since this would introduce inconsistencies, instead of a single sort *Type*, there is an infinite well-founded hierarchy of sorts $Type_i$, such that $Set : Type_1$, $Prop : Type_1$, and $Type_i : Type_{i+1}$, but this is hidden from the user). Basic terms are the sorts (*Set*, *Prop*, and *Type*), constants, and variables. Complex terms are built using abstraction $\lambda x : T$, application $(T\ U)$, and dependent product $\forall x : T, U$. A declaration of a variable x is either an assumption $(x : T)$ or a definition $(x := t : T)$. The type of a term depends on the declarations in the global environment and the local context. $E[\Gamma] \vdash t : T$ denotes that the term t has the type T in the environment E and context Γ. Coq primitive inference rules are minor modifications of the λ-cube rules. For example, (s_1, s_2) rules are given as follows (note that in the second rule s cannot be *Type*—that would give impredicative sets, removed from Coq from v8.0.)

$$\frac{E[\Gamma] \vdash T : s \quad s \in \{Prop, Set, Type\} \quad E[\Gamma, x : T] \vdash U : Prop}{E[\Gamma] \vdash \forall x : T, U : Prop}$$

$$\frac{E[\Gamma] \vdash T : s \quad s \in \{Prop, Set\} \quad E[\Gamma, x : T] \vdash U : Set}{E[\Gamma] \vdash \forall x : T, U : Set}$$

$$\frac{E[\Gamma] \vdash T : Type_i \quad i \leqslant k \quad E[\Gamma, x : T] \vdash U : Type_j \quad j \leqslant k}{E[\Gamma] \vdash \forall x : T, U : Type_k}$$

Similar to the λ-cube, Coq contains the conversion rule that enables computations within the logic (we discuss this feature in more details in Section 4.3).

Along with CoC, *inductive definitions* are the main feature of the Coq's logic. For their general rules we refer the reader to the Coq reference manual[12] and we only show two examples. The inductive definition $Ind()(nat : Set \ := \ O : nat, S : nat \to nat)$ introduces natural numbers. It extends the environment by the term *nat* of the sort *Set*, and two constructor terms: O of the type *nat* that stands for zero, and S of the type $nat \to nat$. The inductive definition $Ind(A : Set)(List : Set \ := \ nil : List, cons : A \to List \to List)$ introduces polymorphic lists (parameterized by the given parameter A of the sort *Set*). The constructor *nil* stands for the empty list, while the constructor *cons* prepends an element to the given list. Coq allows recursive function definitions (using a fixpoint construction). However, all functions must be terminating, as otherwise inconsistencies would be introduced.

[12] https://coq.inria.fr/doc/

4. Features of Proof Assistants

In this section we will describe some important features of modern proof-assistants.

4.1. Reliability of proof checking.

Proof assistants must be reliable. If we are to use them to verify mathematics, hardware, and software, how can we guarantee that they themselves are correct?[13] Although absolute reliability is not possible (the correctness of the underlying hardware, compilers, programming-language run-time environment must be assumed), there are several methods to obtain high reliability.

4.1.1. De Bruijn principle and proof objects.

The *de Brujin principle* (or *de Brujin criterion*) requires that a proof-assistant contains a very simple *proof-checker* that checks the whole formalization (e.g., well-formedness of definitions and correctness of proofs within some given logic). Along the proof checker, proof-assistants can contain many other tools (together called the *proof-development system*). These must produce *proof-objects* (i.e., *proof-terms*) which are then checked by the proof-checker. The soundness of the whole system depends only on the soundness of the proof-checker. Even if some parts of the proof-development system contain bugs, if the proof-object for a given theorem and the proof-checker are correct, the theorem is very-likely correctly proved within the corresponding logical system. Many proof-assistants (e.g., Coq) generate and explicitly store proof-objects.

4.1.2. LCF Principle.

Storing proof-objects required by the de Bruijn principle can consume a lot of memory. As we already said, Edinburgh LCF [107, 52] was one of the most influential ITP systems, and one of the most-important problems that it solved was how to avoid the need to explicitly keep proof-objects, but still retaining high reliability. Milner's idea was to use an abstract data type (often denoted by `thm`) for representing theorems[14], whose only constructors are instances of axioms and inference rules of the underlying logic (e.g., inference rules are implemented as functions from theorems to theorems). Although there are no explicit proof-objects, the system still has a very small kernel—the `thm` datatype and its constructors.

The other Milner's big idea was to use a specialized programming language (Meta Language, ML) to enable extending the prover with custom proof-commands (derived rules, and automated proving procedures). Many proof-commands are readily available in the system, and the user is free to implement his own extensions. ML is strictly typed to support the abstract type mechanism needed to ensure theorem security. Strict typechecking ensures that no theorem can be created by any means, except through the small fixed kernel, so all theorems present in the system (values of type `thm`) are provable in the underlying logic calculus (as they are creating by applying inference rules, starting from the axioms). Exception mechanism of ML is used to signal when a rule (or tactic) is wrongly applied. The soundness of the whole system does not depend on the soundness of proof-procedures implemented in ML—they can have bugs, but the mere fact that they produced a value of the

[13] *"Quis custodiet ipsos custodes?"*—a Latin phrase from Juvenal's Satires literally translated as "Who will guard the guards themselves?".

[14] Other important datatypes in LCF-style provers are `type` for representing types, and `term` for representing terms of the underlying logic.

type thm guaranteed that the particular statement represented by that value is a theorem.

Let us illustrate LCF principle on the example of HOL Light. HOL Light is famous for its simple and very readable code and many things about the implementation of LCF-style theorem provers can be learned from browsing its open-source code-base. In his *Handbook of Practical Logic and Automated Reasoning* [73] Harrison gives a quite detailed tutorial introduction on how to implement an interactive theorem prover (by an example prover for first-order logic). Each inference rule in HOL Light is implemented as a function. For example, TRANS rule (Figure 5) is implemented by a function `TRANS` of the type `thm->thm->thm`:

```
let TRANS (Sequent(asl1,c1)) (Sequent(asl2,c2)) =
match (c1,c2) with Comb(Comb(Const("=",_),l),m1),Comb(Comb(Const("=",_),m2),r)
when aconv m1 m2 -> Sequent(term_union asl1 asl2, mk_eq(l,r))
| _ -> failwith "TRANS"
```

Derived rules are obtained by combining simpler ones (they are also OCaml functions that call simpler functions). Consider the following example.

$$\frac{\Gamma \vdash x = y}{\Gamma \vdash fx = fy}\ \text{AP\_TERM}$$

Using the axioms of HOL Light (Figure 5), this inference rule can be proved by

$$\dfrac{\Gamma \vdash x = y \qquad \dfrac{}{\vdash f = f}\ \text{REFL}}{\Gamma \vdash fx = fy}\ \text{MK\_COMB}$$

This proof is directly reflected in the OCaml code.

```
let AP_TERM tm th=
try MK_COMB(REFL tm, th) with Failure _->failwith "AP_TERM";;
```

Backwards reasoning–tactics and tacticals. Although rules work forward (from premises to conclusions), LCF was famous for also enabling the backward-proof style (from goals to assumptions). Tactics are "inverse" to rules, i.e., they are functions that take goals and reduce them to one or more simpler subgoals. Along with the list of subgoals, each tactic also returns a justification (validation) function that reconstructs the original goal once the subgoals are discharged. When the backwards proof is done, the justification function is applied, and it yields the desired theorem (proven subgoals yield values of the thm type, and the justification function performs a forward reasoning to build the new thm value corresponding to the main goal). Rules can be composed as functions (since they are represented by functions). On the other hand tactics can be composed by tacticals (e.g., one of the basic tactical is THEN that is used to chain the application of two tactics).

4.1.3. Definitional principle. To ensure soundness, whenever possible new objects should be introduced by definitions, rather then axioms. Namely, axiomatic developments can easily introduce inconsistencies, so are axioms should be treated with special care. Although definitions cannot introduce inconsistencies, errors in definitions are also very dangerous since every proof is checked only modulo given definitions. If definitions are wrong, then the theorem proves something different to what was originally intended. Therefore, it is a good practice to separate basic definitions that are necessary to state central theorems of a development. These definitions and theorem statements must be carefully checked manually and it must be confirmed that they agree with the intuition—this step cannot be automated.

4.2. Procedural vs Declarative Proofs. There are two important dimensions of mathematical proofs. First, proofs serve to give a *justification* (a *certificate*) that the statement holds. Second, proofs serve to give an *explanation* why the statement holds (they convey a *message*).[15] Interactive theorem provers can be classified according to the style of their proof-languages (formal languages in which proofs are specified). *Procedural* proof languages are usually designed with justification in mind (they offer strong automation and very short and concise proofs). However, if proofs are meant to convey an explanation, they must be human-readable. *Declarative* proof languages give the possibility to write proof texts in a (controlled) natural language like syntax. A good analogy is with a game of chess [4]. A game can be described either as a sequence of moves, or as a sequence of positions. In the first case (corresponding to the procedural style) positions are implicit and can only be reconstructed if moves are executed from the very beginning. Similarly, a procedural proof is represented only as a sequence of proof rules (or tactics in a backward proof) that transform the current proof-state when applied. Consider the procedural style proof in Isabelle/HOL, shown on the left side of Figure 9.

```
lemma "¬ p ∨ ¬ q ⟶ ¬ (p ∧ q)"     lemma "¬ p ∨ ¬ q ⟶ ¬ (p ∧ q)"
apply (rule impI)                  proof
apply (rule notI)                     assume "¬ p ∨ ¬ q" thus "¬ (p ∧ q)"
apply (erule conjE)                   proof
apply (erule disjE)                      assume "¬ p" show "¬ (p ∧ q)"
apply (erule notE, assumption)           proof (rule notI)
apply (erule notE, assumption)              assume "p ∧ q" hence "p" ..
done                                        with ‘¬ p‘ show False ..
                                         qed
                                      next
                                         assume "¬ q" show "¬ (p ∧ q)"
                                         proof (rule notI)
                                            assume "p ∧ q" hence "q" ..
                                            with ‘¬ q‘ show False ..
                                         qed
                                      qed
                                   qed
```

Figure 9. Procedural and declarative proof in Isabelle/Isar

Each step in this simple proof applies only a single natural deduction rule or the axiom (although the whole proof could be done by `apply auto` which would call the automated prover). The proof state is implicit in the proof text, and the only way to see the intermediate formulae in a proof is to execute proof steps from the very beginning. For example, the proof state, after applying `conjE` is:

```
⟦¬ p ∨ ¬ q; p; q⟧ ⟹ False
```

[15] A wider discussion of this topic is given by Asperti [4].

On the other hand, declarative proofs contain explicit formulations of intermediate steps in the proof. Therefore, proofs are somewhat longer, but are more easy to understand, and can be read without running them inside the proof-assistant. Instead of using proof-scripts (sequences of commands), proof-assistants now use proof-documents (structured text). Tradition of embedding human-readable proof descriptions into proof-assistants is very old. One of the most important examples is the *Mizar* proof language [109]. *Isar* is a declarative proof language for Isabelle/HOL developed by Wenzel [141], inspired by Mizar (a comparison of Mizar and Isar proof languages is given in [145]). Consider the Isar proof of the previous statement shown on the right side of Figure 9. In Isar, each `proof-qed` block automatically applies a single natural deduction rule (determined by the shape of the goal that it proves). In our example, the top-level proof begins with the implication introduction rule (since $\longrightarrow$ is the leading connective), and in the next line explicitly tells that `¬ p ∨ ¬ q` can be assumed, and that from that assumption, `¬ (p ∧ q)` must be proved. Next proof automatically recognizes that a disjunction elimination should be performed, leading to two cases in that subproof. It is also possible to start a proof with an explicit call (as we illustrated it with `proof (notI)`, although that was not necessary). Terminal proofs are usually performed by a call to some automated tactic, or by some rule (we used `..` that automatically finds the appropriate natural-deduction rule to discharge the current goal).

Declarative proof-languages usually combine backward and forward proving, while procedural proof languages usually favor the backward proving style.

Some other proof-assistants also have declarative proof-languages (e.g., Mizar mode for HOL Light [146], or C-zar language for Coq [31]), although they are not used so intensively as Mizar and Isar. Many languages (including Isar) allow mixing declarative and procedural style. The SSReflect extension for Coq [49] is mainly procedural, but still uses a declarative style for the high level structure of the proof. A new approach called `miz3` significantly differs from the ways in which the procedural and declarative proof styles have been combined before [149].

4.3. Computations and Poincaré principle. Some proof assistants follow the *Poincaré principle*, allowing computations as elementary steps in logical proofs (e.g., $2 + 2 = 4$, holds by computation and there should be no need to justify such inferences with long chains of logical inferences). For example, from the standpoint of the Coq logic, the two statements $P(2 + 2)$ and $P(4)$ are not just equivalent, they are identical (Coq proof term does not need to contain complex justifications of this fact— it holds by reflexivity). The Poincaré principle is usually realized through the *conversion rule*. For example, in λ-cube, there is a rule

$$\frac{\Gamma \vdash A : B \quad \Gamma \vdash B' : s \quad B \to_\beta B'}{\Gamma \vdash A : B'}\ \text{conversion}$$

Therefore, if the two terms are computationally equivalent (there is a β conversion from one to the other), then one can be substituted for the other, and this need not be specially justified, except by referring to the conversion rule itself. Exact computation steps (steps performing the β conversion) are not present in the proof term, and it is up to the proof-checker to check their correctness (β-convertibility). Besides applying functions, computations may involve other operations. For example, in Coq, the conversion rule does not only include β-conversion, but also δ-conversion (unfolding the definitions), ι-conversion (applying inductive definitions), η-conversion (extensionality), and ζ-conversion (unfolding the *let* expression).

4.3.1. Program Extraction and Code Generation. Most proof assistants use HOL that can be treated as purely functional programming language (some even support computations inside the logic). Many of them also support *program extraction* (also called the *code generation*) by translating specifications from their internal language, to an external one. For example, the HOL definition of $f\ n = n + 1$ can be translated to an almost identical function definition in Haskell `f n=n+1`. However, the HOL definition $P\ f \longleftrightarrow (\forall k.\ f\ k > 0)$ cannot easily be translated to executable code (due to the presence of the quantifier that is generally not executable). Coq can generate OCaml, Haskell, and Scheme both from (recursive) function definitions and constructive proofs [91], Isabelle/HOL can generate code in these languages from (recursive) function definitions [57, 55], PVS allows translation to Common Lisp, while the language of ACL2 is (almost) a subset of Common Lisp so the translation is (almost) direct. The correctness of the code generation is usually discussed only informally, and the correctness of the generated code relies on the correctness of the generator. However, if the source, target and all intermediate languages have well-defined and similar equational semantics, code generator can be trusted with a high degree of reliability.

4.3.2. Proofs by Reflection. *Computational reflection* is one approach for integrating decision procedures into proof-assistants, alternative to the LCF-style tactics. The main concern with the LCF approach is that all proofs must be composed from basic inferences, which can be inefficient for large proofs. On the other hand, LCF-style decision procedures need not be verified—if they produce an erroneous result, it will be discarded by the type-system. Contrary to that, computational reflection assumes that decision procedures are verified within the logic. In essence, using computational reflection translates theorem proving and explicit proof-steps (basic inferences) into computation in the meta-theory and implicit proof-steps [21]. To be able to write functions by pattern matching over the syntax, the relevant fragment of formulae in the logic are represented using a datatype—a *shadow syntax.* Each formula φ in the shadow syntax is interpreted back into logic as $[[\varphi]]$ (such interpretation function can easily be defined within the logic). A decision procedure $\mathcal{D}$ is then defined over the shadow syntax and proved to be sound ($\forall\varphi.\ \mathcal{D}(\varphi) = \texttt{true} \Longrightarrow [[\varphi]]$). To prove a proposition Φ, it must be *reified*, i.e., a formula φ is found such that $[[\varphi]] \equiv_\beta \Phi$ holds (this is usually done by a reification function implemented in the meta-logic). Then, the decision procedure $\mathcal{D}$ is applied on φ, and if it returns `true`, the proposition Φ is proved. The evaluation of $\mathcal{D}$ is optimized (its code is sometimes exported to ML and executed on the ML level), and the evaluation steps are not shown in proofs, making reflection more efficient than LCF-tacitcs (especially when explicit proof-terms are recorded).

4.4. Modules, Locales, Refinement. Proof-assistants should support a system of modules for organizing larger formalizations. Isabelle supports *locales* [82], and Haskell-like *type classes* [58] (somewhat similar to classes in Coq). A locale consists of a sequence of context elements declaring parameters (constants, and functions) and assumptions. For example, the locale describing partial orders fixes a function $\leqslant$, and assumes that it is reflexive, anti-symmetric, and transitive. The theory of partial orders is developed within that locale (e.g., a function $<$ is defined using $\leqslant$ and $=$, and its properties are proved). Finally, the locale is interpreted in different contexts (e.g., it is shown that divisibility on natural numbers is a partial order) and all properties proved abstractly, within the locale are transferred to the concrete interpretation. Locales can be extended (e.g., a total order extends partial orders), and interpreted in the context of some other locale (e.g., total order is a sublocale of a lattice, as lattice operations can be defined in terms of the total order operation).

Locales can be used to express *refinement*–a stepwise approach to software development that starts from a very abstract specification, and refines it by introducing details concerning data-structures and algorithms. For example, the Isabelle Collections Framework [87] uses refinement based on locales and provides JAVA-like collections (Lists, Sets, Maps) for use in Isabelle/HOL. Besides locales, there are many alternative ways to use refinement within a proof assistant [56, 88].

4.5. Integration with external automated solvers and provers. There are many theorem provers that work fully automatically, but are either incomplete, or work in some very specialized fragments of logic. Interactive theorem proving could benefit from synergy with such systems, and there is currently a lot of research in that direction.

4.5.1. Integration with SAT/SMT Solvers. SAT and related SMT solvers [16] are very efficient decision procedures for propositional logic and some specific theories of first-order logic (e.g., linear arithmetic, equality with uninterpreted functions). ITP can benefit from the power of SAT/SMT solvers, since goals that can be expressed in the fragment supported by a SAT/SMT solver can be passed to it and solved without human guidance. However, SAT/SMT solvers cannot be trusted (they are very complex and can contain bugs), and their integration into ITP systems must be done very carefully. Usually the integration is based on certificates exported by the solvers (for satisfiable instances these are just models, and for unsatisfiable instances these are resolution proofs) that are independently checked by the proof-assistant kernel. Interestingly, the proof-verification (done by the proof-assistant) is often slower than proof-finding (done by the SAT/SMT solver).

There have been several recent successful attempts to integrate SAT and SMT solvers into ITP systems. Weber has integrated zChaff SAT solver into Isabelle/HOL [140]. Weber and Böhme have integrated Microsoft's z3 SMT solver into Isabelle /HOL [20]. Before integration, the solver had to be adjusted to export object-level proofs of unsatisfiability (details are given in Böhme's PhD thesis [18]). Armand et al. have successfully integrated SAT solver zChaff and the SMT solver veriT into Coq [2]. In all cases, reflexive tactics are implemented, making it possible to apply solvers to native formulas of the proof-assistant logic.

4.5.2. Integration with ATP's (Sledgehammer Approach). There are many efficient, fully-automated theorem provers, based on the first-order logic (FOL) and clausal resolution (e.g., Vampire, Spass, e-prover). Although the higher-order logic (HOL) used in ITP is much more expressible than FOL, many encodings of HOL formulae into FOL are devised and they can be used to employ theorem provers based on FOL to discharge proof obligations in interactive proofs. One approach for this is *Sledgehammer*, a component of Isabelle/HOL, initially devised by Meng, Quigley, and Paulson [104], and later refined by Nipkow, Böhme, and Blanchette. Along with first-order ATP's, Sledgehammer now supports SMT solvers [17]. At any point in an interactive proof session, user may invoke the tool. The current goal (suitably encoded into FOL) is passed to underlying ATP and SMT systems. Sledgehammer uses relevance filtering (based on machine learning heuristics) to select facts that are probable to be used in a proof of the current goal. Facts are selected from enormous Isabelle's lemma-libraries and other facts established in the current context, and usually only several hundred lemmas are selected. The current goal and the selected facts are then passed to automated solvers, which are then run in parallel. If a proof is found, the system reports facts that were used in a proof (usually, only a small number of facts is used). These facts are then given back to some of the Isabelle's internal automated tactics (usually `metis`) that finds the proof from scratch (this usually succeeds, since it gets a short list of relevant facts). From the average Isabelle user's perspective, Sledgehammer is a very useful tool [19]. Although larger proof steps are out of reach of automated proofs, Sledgehammer significantly helps discharging smaller and simpler proof obligations and improves the overall Isabelle's user experience.

4.6. Parallel Proof Checking. In recent years we have been witnessing the stagnation of CPU clock frequencies, and the development of now standard multi-core, shared-memory CPU architectures. Unfortunately, that imposes the burden of explicit parallelism to application developers, and proof-assistants have to be adapted to make full use of these new hardware architectures. Currently, there are not many proof-assistants that answer to the multi-core challenge. The only major proof assistants that do, are Isabelle/HOL [144, 143] and ACL2 [125]. Parallel processing is mostly implicit, without the user intervention. However, to obtain this, substantial reforms of the prover architecture, its implementation, and the underlying run-time environment had to be made. The central idea to parallel proof checking in Isabelle is that proofs rely only on the statements of other lemmas, and not on their proofs, so they can be checked "conditionally", even before all lemmas used in a proof are checked (of course, in the end everything must be checked to have a full valid proof).

4.7. User Interfaces. As provers were built on top a functional language that are usually interpreted, user interaction was command-line based and implemented by a traditional READ-EVAL-PRINT loop.

A step ahead was Aspinall's *Proof General* [6]—an Emacs mode that supports several provers, including Isabelle and Coq. The interaction is still based on a sequence of commands, with a single focus point delimiting the part of the proof that has been checked (locked for further editing), and the part that has not yet been examined. User advances the proof (usually step-by-step), prover does the checks and prints feedback to a separate buffer (window). This interaction model was imitated in several other proof-interfaces (e.g., *CoqIde*, *ProofWeb*, *Matita*).

A completely different approach is implemented through $\pi d.e$ *(Prover IDE framework, PIDE)* [142]. It performs continuous parallel proof checking (similar to modern programming IDE's that do syntax checking of source code while the user is typing). The interaction between the IDE and the underlying prover is asynchronous (there is no current cursor point in the proof, no locking, and the prover reacts to changes made by the user by checking only when necessary). The prover annotates the processed text with information that can be displayed to the user on his request (e.g., mouse tooltips can be used to show deduced types). Although it should support multiple proof assistants, its most developed version is for Isabelle.

5. Major Achievements in ITP

In this section we briefly describe state-of-the-art results and major achievements of interactive theorem proving obtained mainly during the last decade. There are many other successful large-scale formalization projects that deserve to be shown, but we do not describe them due to the lack of space (e.g., Gödel's Incompleteness Theorem [130], real analysis and Fundamental Theorem of Calculus [69, 33], Brower Fixed Point Theorem and StoneWeierstrass theorem [75], Fundamental Theorem of Algebra [105, 42], Jordan Curve Theorem [61], Central Limit Theorem [10]).

5.1. Prime Number Theorem. The *Prime Number Theorem* (PNT) describes the asymptotic distribution of the prime numbers. If $\pi(x)$ denotes number of primes less than or equal to x, the theorem states that $\lim_{x\to\infty} \frac{\pi(x)}{x/\ln(x)} = 1$. This was conjectured by Legendre, Gauss and Dirichlet in the late 18th century. Chebyshev made a significant progress, but the decisive step was made by Riemann who connected the distribution of primes to zeros of the zeta function. This made possible to apply powerful methods of complex analysis, and one hundred years after the original conjecture Hadamard and de la Vallée-Poussin independently came up with a proof. Although some famous mathematicians (e.g., Hardy) believed that the proof requires complex analysis, in 1949 Selberg and Erdös independently found "elementary proofs", based on a "symmetry formula" due to Selberg.

Elementary proof. In 2005, using the proof-assistants Isabelle/HOL, Avigad, Donelly, Gray, and Raff formally verified the proof of PNT [8]. The statement in the Isabelle/HOL assistant is quite close to the original formulation, and the authors did not need to invent a lot of material to state the theorem, although the statement requires some advanced notions (real numbers, logarithms, and convergence). The formalization follows the proof given by Selberg and the exposition given by textbooks of Shapiro [131], Nathanson [108], and Cornaros and Dimitracopoulos [32] who reformulated Selberg's proof in some weak fragments of Peano-arithmetic. The authors claimed that "the formalization of the prime number theorem is a landmark, showing that todays proof assistants have achieved a level of usability that makes it possible to formalize substantial theorems of mathematics" [8]. Besides this, the formalization has several very important by-products, some of which were later included in the Isabelle/HOL distribution:

- a theory of the natural numbers and integers, including properties of primes and divisibility, and the fundamental theorem of arithmetic,
- a library for reasoning about finite sets, sums, and products,
- a library for the real numbers, including properties of logarithms.
- a library for asymptotic "big O" calculations,
- a number of basic identities involving sums and logarithms.

Most effort is spent on gathering a basic library of easy facts, proving trivial and "straightforward" lemmas, entering long expressions correctly, and adapting ordinary mathematical notation to the formal one. Calculations with reals, especially with of inequalities, and combinatorial reasoning with sums were problematic. Unlike in informal proofs where types are only implicit, in strictly typed Isabelle/HOL setting, explicit casting between nturals, integers and reals was omnipresent (a substantial theory for the floor and ceiling functions had to be developed). Some theorems needed to be proved in different versions, depending of the number type. A better automation support for all these issues would be very welcome.

The final formalization includes over 30,000 lines of code (excluding libraries on real and complex numbers). The de Bruijn factor varies between around 5 and 15.

Analytic proof. In 2009, using the proof-assistant HOL-Light, Harrison formalized an analytic proof [72]. It was compiled based on several informal sources, but mostly followed the "second proof" from Newman's book [111], pp. 72–74 using the analytic lemma on pp. 68–70. As the theorem has been already mechanically established prior to his work, the author claim that "the formalization was undertaken purely for fun, involving complex analysis and culminating in a proof of the Prime Number Theorem". However, comparing Harrison's proof to the elementary one shows that they are very different, especially when considering the by-products:

- a library of real numbers, based on an encoding of Cauchy sequences,
- formalization of analysis in $\mathbb{R}^N$,
- a theory of complex numbers $\mathbb{C}$, built on top of $\mathbb{R}^2$ (including notions such as complex differentiation, holomorphic functions, analytic functions, valid paths, and Cauchy integral formula for topologically simple regions).
- definition of the ζ function and its basic properties (e.g., analyticity on $Re\ z > 0$ and $z \neq 1$).

Formalizing advanced proofs gives quite different experience than formalizing tedious elementary proofs. Although the analytic proof uses heavier machinery (complex analysis), the elementary proof is "more complicated and intricate, indicating that there is a price to be paid for avoiding analysis". More effort was spent trying to understand Newmans proof informally, then typing its formalization into HOL. Taking the analytic preliminaries (e.g., the Cauchy integral formula) into account makes things more complicated as it seems that more elementary the results are, it is harder to address them formally (as one does not have any deep results on his disposal, but is forced to rely on basic proof techniques). Harrison advocates that when formalizing mathematical results, the main result itself is not as important as all the background knowledge that also needs to be formalized. In the long-run it pays off to avoid shortcuts and to develop the background library systematically, as it is very likely that it can be reused in some different contexts.

The formalization includes about 5000 lines of code. The de Bruijn factor varies from 5 to more than 80 in the chapter about the ζ-function. Such large factors are mainly due to the fact that Newmann's book [111] is graduate level with some statements given without a proof in any explicit sense.

5.2. Four-Color Theorem. The history of the Four-Color Theorem (FCT) is more than 150 years old (detailed overview is given by Wilson [151]). In 1852, Gurhrie tried to color the map of UK with as little colors as possible, so that no two adjacent counties share the same color. He did it using only four colors and conjectured that only four colors would suffice for any other map. de Morgan became interested and introduced the problem to wider mathematical community. That initiated a history of false counterexamples and incomplete or false proofs (e.g., by Kempe in 1879 or Tait in 1880). Some progress was made by Birkhoff, but it became clear that large case-analysis will be necessary to complete the proof. In 1960s and 1970s Heesch and Durre proposed to use computers in some parts of the proof, and in 1976 Appel and Haken came up with the first successful proof of FCT [1]. It raised much controversy, since it was based on a IBM 370 assembly language program that performed a case analysis with more than 10,000 cases (to make the matters worse, some minor errors were detected after the original publication). It was very hard to relate the computer program to the abstract statement of the theorem, so many mathematicians did not accept this kind proof. In 1995, Robertson, Sanders, Seymour, and Thomas published a refined version of the proof [126]. It was based on similar arguments to Appel and Haken, but they used C programs and significantly reduced the number of cases.

Finally, in 2005 Gonthier developed a full formal proof of FCT using the Coq proof-assistant [45, 46, 47]. The project started in 2000 and in the beginning, they only wanted to check the computations described by Robertson et al. [126] within Coq and to evaluate computational capabilities of a modern formal proof system—at first, only to benchmark its speed. Encouraged by the success of the computation part, they decided to tackle the whole problem. To have the entire proof of FCT, it was necessary to show that the specification of computations is correct, and to connect it with the mathematical statement of the theorem.

The first step was to make a precise formulation (the naive one: "Every map can be colored with only four colors" is obviously very imprecise). Several definitions were introduced [45]. "A *planar map* is a set of pairwise disjoint subsets (*regions*) of the plane. A *simple map* is one whose regions are connected open sets. Two regions are *adjacent* if their respective closures have a common point that is not a corner of the map (a point is a *corner* of a map iff it belongs to the closures of at least three regions)". Formalization of these definitions relies on standard notions of real numbers, open sets, connected sets, closures, etc. When formalized, all definitions required to state the FCT precisely take less than 250 lines of Coq code.

Although the statement is based on topological notions, it is essentially combinatorial: it gives properties of finite arrangements of finite objects. An important step in the proof was to establish a combinatorial characterization of the theorem. This is usually done by constructing the dual graph of the map (coloring a map is trivially equivalent to coloring the graph obtained by taking the regions of the map as nodes, and linking every pair of adjacent regions). However, such approach applies the Jordan curve theorem to make use of the planarity assumption, but that is hard to formalize (formalizing the Jordan curve theorem itself is quite challenging as the proof relies either on complex analysis or homology [61]). Instead of dual graphs, Gonthier used *hypermaps*—well-known combinatorial structures that can explicitly represent all local geometrical connections, support a clearly recognizable definition of "planar" (based on Euler formula, instead of the Jordan Curve Theorem), and are easily manipulated in Coq. For each original map, there is a hypermap assigned to it by means of discretization and it corresponds to a *polyhedral map*—a finite planar, bridgeless connected polygonal map.

The combinatorial proof is roughly based on Kempe's ideas from 1879, subsequently refined by Birkhoff, Heesch, Appel, Haken, and Robertson et al. Informal proof is a proof by contradiction, but to remain intuitionistic, the Coq proof is by induction (on the number of regions in the map). The Kempe's proof outline is as follows. First, it is enough to consider only cubic maps, where exactly three edges meet at each node (if each node of the original map is covered by a small polygonal face and the map is colored, deleting the added faces will yield the coloring of the original map). By analysis based on Euler's theorem, in such map there will always be a *central face* with at most 5 sides surrounded by a ring of other faces. Deleting one or two suitably chosen edges of the central face yields a smaller graph that is 4-colorable by induction. When the deleted edges are redrawn, if the central face is not a pentagon, there is a color available for it. If it is, then there would always be a way to recolor the surrounding faces by locally exchanging the colors of adjacent faces, so that the pentagon can be colored. This part of Kempe's proof contained a flaw that took ten years to discover and more than hundred years to fix. Instead of single central faces, following Birkhoff, the correct proof considers larger map fragments—*configurations* consisting of central whole faces (the *kernel*) surrounded by a *ring* of partial faces. The proof then enumerates a list of such configurations so that it satisfies:

unavoidability: : every map must contain a configuration from the list;

reducibility: : every configuration is reducible, i.e., Kempe's argument is sound for it—there is a recoloring scheme of its surrounding faces, such that its faces can be colored to fit into the surrounding. Birkhoff was the first one to find a reducible configuration (the one shown on Figure 10).

Following [126], the final list contains 633 configurations, with rings of up to 14 partial faces. Showing that a configuration is reducible is straightforward, but requires huge case-analyses (the number of cases increases exponentially to about 20 million for rings of size 14), consuming most of the proof-checking time (originally it was several days, and it was later improved to several hours). Showing

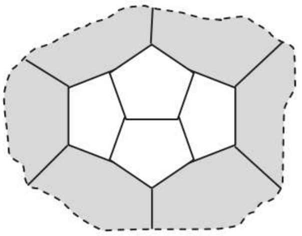

Figure 10. A configuration: kernel faces (white) and a ring (gray)

unavoidability is also based on computations and follows the technique called *discharging* given by Heesch in 1969. It enumerates faces and their neighborhoods satisfying some necessary conditions (determined by an analysis based on Euler's formula and Birkhoff's results), finding reducible configurations in the neighborhoods. The enumeration is complex, but not as computationally intensive as the reducibility checks.

It is obvious that this whole proof heavily relies on the proof-by-computation paradigm. The *computational reflection* of Coq lies in its core. As we have already described, the computation is embedded into the Coq logic. Reflection was applied whenever possible (both "in the large" and "in the small"). For example, showing reducibility worked as follows. First, the predicate `cfreducible` that defines reducibility is defined (it is a standard, non-executable logical definition). Next, an executable function `check_reducible` is defined, that preforms huge case analysis (using efficient data structures and computation algorithms). Then, it is shown that `check_reducible` is valid, i.e., that if it returns `true`, then `cfreducible` must hold. Finally, reducibility of every candidate configuration is shown in just two steps: applying the validity argument for `check_reducible`, and then reflexivity (showing that `true=true`, but only after reducing `check_reducible` call to `true` which is done internally by the proof checker, and is transparent in the proof).

The success was largely due to the fact that the FCT was approached mainly as a programming problem, rather than a formalization problem. Most properties were formalized as computable predicates, so that they can be checked by execution. The success of this approach in the large (checking reducibility proofs that were also done by computation in earlier, non-formal proofs), inspired the authors to follow that style in the small, whenever it was possible, and it turned out to be a success. A byproduct of the FCT formalization was the SSReflect extension for Coq [49]. It started as a tactic "command-shell", but evolved into a powerful dialect of Coq, that facilitates proofs by reflection (hence the name, small-scale reflection). Proofs written using SSReflect are far from declarative and do not read as mathematical text, yet they are very succinct and can be very complex.

The final formalization comprises around than 60,000 lines of Coq code, more than 1000 definitions, and more than 2500 lemmas.

5.3. Odd Order Theorem. The Odd Order Theorem (OOT) is a part of the *classification of finite simple groups*—a grandiose result in mathematics, completely finished in 2008, after more than one hundred years of intensive research. A group is *simple* iff it is not trivial and has no normal subgroups except itself and the trivial group. The reader can notice some resemblance between simple groups and prime numbers (they have no factors except one and themselves). A group that is not simple can be decomposed into its normal subgroup and the corresponding quotient group (called the *factor*). This can be further repeated and each finite group G can be decomposed into a strictly increasing series of normal subgroups $1 = G_0 \lhd G_1 \ldots \lhd G_n = G$ such that all factors G_{k+1}/G_k, $0 \leqslant k < n$ are simple (this is called a *composition series*). By the so-called Jordan-Hölder theorem, this composition is unique, resembling the fundamental theorem of arithmetic.

In 1911, Burnside conjectured that every non-abelian (non-commutative) finite simple group has even order. Conversely, every finite simple group of odd order must be abelian. A group is *solvable* iff it can be decomposed into a series of abelian factors. Since every finite abelian group can be decomposed into cyclic groups of prime order, a finite group is solvable iff its composition series consists only of cyclic factors of prime order. Burnside's conjecture is equivalent to stating that *all finite groups of odd order are solvable.* This statement is known as the *Odd Order Theorem* or *Feit-Thompson Theorem*, since it was was originally proved by Feit and Thompson in 1963 [38]. Maybe the most revolutionary aspect of this proof was that it was more than 250 pages long and consumed the whole volume of the journal that it was published in. This has encouraged other researchers in the group theory to pursue very long and complex proofs (the final proof of the classification of finite groups consumes more than 10,000 pages, while its final part done by Aschbacher and Smith on quasithin groups alone consumes 1221 pages [3]).

In 2012, a team led by Gonthier announced that they have successfully formalized the proof of the OOT in Coq [48]. It was a result of a collaborative effort (the paper was written by 15 authors) of the researchers gathered by the *Mathematical components* project that lasted for more than six years.

A minimal, self-contained formulation of the OOT is, amazingly, less than 100 lines of Coq code (including definitions of natural numbers, finite subsets, and elementary group theory). Although the statement is easily formulated, its proof posed a major challenge for formalization, mainly due to its length, and the range of mathematics involved. It combines finite group theory, linear algebra (representation of groups by matrices, determinants, eigenspace decomposition, extended Gaussian elimination), Galois theory (connecting groups of automorphisms with field extensions), and the theories of real and complex algebraic numbers. Authors mainly followed Bender et al. [14] and Peterfalvi et al. [123] that together give a simplified version of the Feit and Thompson's proof (although the main structure remains the same), with several exceptions where they formalized parts of the original proof [38].

Unlike the proof of FCT which relies on mechanical computation, proving OOT required formalizing common patterns of mathematical reasoning and notation. Classical mathematical texts often hide many details, and rely on the readers ability to infer them from the context. Some of such features are described by Avigad [7]. For example, consider the statement *"If G and H are groups, f is a homomorphism from G to H, and a and b are in G, then $f(ab) = f(a)f(b)$"*. Saying *"G and H are* *groups"*, really means that both G and H are set of elements equipped with a group operation, its inverse and an identity element. Saying *"a and b are in G"* really means that a and b are elements of the underlying set. The notation ab denotes multiplication in G, while $f(a)f(b)$ denotes multiplication in H. To keep a large formalization manageable, the main challenge is to develop techniques that make the formal development similar to such informal presentation as close as possible. In the formalization of the OOT, the authors showed that proof-assistants can preform many inferences such as those shown in this short example, so the formalization does not need to have all details explicitly spelled out. The mechanism heavily relies on the dependent types, and additional features of Coq such as *implicit arguments*, *coercions* and *canonical structures* [7].

Continuing the tradition of the Four-Color Theorem proof, reflection is used wherever possible, and SSReflect extension [49] is used. Although the reflection was mainly small-scale, there was an exception and one two-page combinatorial argument (Section 3 of [123]) was proved using large-scale reflection (this could have been proved using trusted connection with SMT solvers, but that was avoided in order to have everything within Coq). Some of additional features of the SS-Reflect library helped in the formalization. For example, the `wlog` (without-loss-of-generality) tactic helped dealing with symmetric arguments (e.g., if $P(a, b)$ is symmetric argument in numbers a and b, then wlog tactic can introduce the assumption $a \leqslant b$). In many cases, proofs use long chains of non-strict inequalities starting and finishing with the first term. Hence, each inequality must be an equality, and conclusions are drawn from this fact. A special support for this kind of reasoning was used. The whole proof was intuitionistic, although classical properties of finite, countable, and types with decidable first-order theories were used (e.g., Markov's principle enables deriving $\exists n\ P(n)$, from $\neg\neg\exists n\ P(n)$, for decidable P).

The formalization includes more than 150,000 lines of Coq code, around 4,000 definitions and 13,000 lemmas, with a de Bruijn factor of 4-5. The most time-consuming part project was getting the base and intermediate libraries right. These consume almost 80 percent of the work, and what is very important, they are reusable for future formalization efforts. To formalize this amount of mathematics, many techniques had to be combined. The final conclusion is very optimistic: "this success shows that we are now ready for theorem proving in the large" [48].

5.4. Flyspeck Project. In 1611, Kepler conjectured that the densest arrangement of balls of the same radius is the *cannonball packing* (in chemistry known as the *face-centered cubic packing*), shown on Figure 11. This is considered to be one of the oldest problems in discrete geometry. This packing has density $\pi/(3\sqrt{2}) \approx 0.740$, and it is not the only such packing (e.g., the *hexagonal close-packing* has the same density). The *Kepler Conjecture (KC)* turned out to be very hard to prove, and Hilbert listed it as the 18th problem one of his famous list. Thue proved a 2d-variant (circle density can be $\pi/(2\sqrt{3}) \approx 0.91$), using an elementary proof. The 3d-variant turned out to be intractable, even though several great mathematicians tried to solve it (e.g., Gauss, Fejes Tóth). Fejes Tóth was the one who set the general strategy for the proof (reduce KC to an optimization problem over finite number of variables and use computers to solve it).

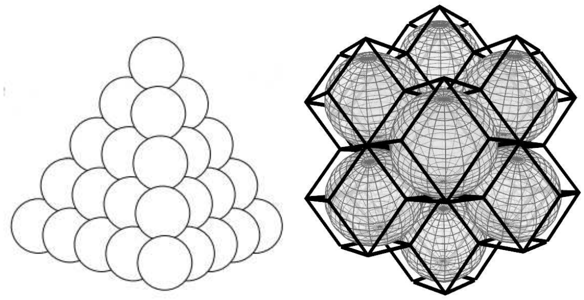

FIGURE 11. *Left*: Cannonball packing. *Right*: Voronoi cells around a face-centered cubic ball packing are rhombic dodecahedra that form a tesselation of space.

Finally, in 1998, Hales announced the first full (although computer-assisted) proof, but it was not published until 2005 [60]. A significant contribution to the proof was given by Ferguson. A high-level, readable description of the proof is given by Hales [59]. The proof follows Fejes Tóth in many aspects. Each packing is determined by the set Λ of ball centers. The packing determines a region around each ball called the *Voronoi cell*–it consists of all points that are closer to the center of that ball than to any other point in Λ. The density of the packing is determined by the ratios of ball volumes and their corresponding Voronoi cell volumes. Voronoi cells of the face-centered cubic packing are rhombic dodecahedra. They form a tessellation (honeycomb) of 3d space (Figure 11, right), and since the volume of each such rhombic dodecahedron is $v_{fcc} = 4\sqrt{2}$, they give the volume ratio of $\frac{(4/3)\cdot\pi}{4\sqrt{2}} = \frac{\pi}{3\sqrt{2}}$. Voronoi cells of the hexagonal close-packing are trapezo-rhombic dodecahedra, which also have the volume $4\sqrt{2}$ and form a tessellation.

The proof of the KC reduces the infinite problem to a finite one. Instead of analyzing all balls in the packing at once, *clusters* of balls are considered. A cluster contains a ball in its center and a several balls surrounding it (these are the balls whose centers are within a given distance from the origin).

One approach to prove the KC is to find the minimal volume of the central Voronoi cell in each cluster. The minimum is achieved only if the cell is a regular dodecahedron that touches the central ball in all its faces—this is the *Dodecahedral Conjecture (DC)* posed by Fejes Tóth and proved by Hales and McLaughlin in 1998, using similar techniques to the KC proof, and published in 2010 [66]. The DC gives an upper bound (around 0.755) on the packing density. However, unlike in the 2d case where the minimal-area Voronoi cells are regular hexagons and they can be tessellated to give the optimal packing of unit circles, a tessellation of the 3d space with regular dodecahedrons is not possible (volumes of Voronoi cells that are locally optimal, cannot be combined to give a globally optimal packing). Therefore, it is not possible to achieve the upper bound derived from the DC (the KC gives the value around 0.740), and the DC is not directly used in the proof of the KC.

The KC is reformulated to finding $\min_p F(p)$, for each cluster, where p ranges over all packings, $F(p) = vol(V(p)) + f(p)$, $vol(V(p))$ is the volume of the central Voronoi cell, and f is a carefully chosen *correction function.* The conjecture follows if the function f is *fcc-compatible*, meaning that the minimum is exactly v_{fcc}, and if f is *transient* meaning that $\sum_{\lambda\in\Lambda_R} f(p_\lambda) = o(R^3)$ when $R \to \infty$, where p_λ is a cluster around the center λ (translated to the origin), and Λ_R is the set of all centers from Λ within the distance R from the origin. Indeed, since $v_{fcc} \leqslant vol(V(p))+f(p)$, summing over Λ_R gives, $|\Lambda_R|v_{fcc} \leqslant (4/3)\cdot R^3\cdot\pi+O(R^3)$ (as volumes of the Voronoi cells consume the volume of a sphere of radius R). Dividing by R^3v_{fcc} gives the density $|\Lambda_R|/R^3 \leqslant \pi/(3\sqrt{2}) + o(1)$, implying the KC in the limit when $R \to \infty$. Indeed, the density is the limit when $R \to \infty$ of the ratio of volumes of the balls in the sphere of the radius R (that is $|\Lambda_R| \cdot (4/3) \cdot \pi$), and the ratio of that sphere (that is $(4/3) \cdot R^3 \cdot \pi$), so it is equal to $\lim_{R\to\infty} |\Lambda_R|/R^3$. The main difficulty in the proof was to construct a function f with the given properties (as Hales admits, on each conference where he presented his work in progress, he minimized a different function [59]). In all cases f was chosen to be transient by its definition, and its fcc-compatibility was shown by solving the (non-linear) optimization problem.

The set of clusters is so complicated, that F could not be minimized directly. The most important geometric features of each cluster can be represented by a planar graph. In most cases, the combinatorial structure of that graph is sufficient to show that $F(p) \geqslant v_{fcc}$. This requires making combinatorial approximations (calculations of a lower bound on $F(p)$ that depends only on the combinatorial structure of the graph associated with p), and computer assistance can be used to compute such bounds. However, there is a finite number of cases (around 5000 plane graphs) for which the combinatorial approximation gives $F(p) < v_{fcc}$. Such graphs are called *tame* and their full list was assembled with the computer assistance. One case turned out to be particularly difficult and it alone was solved in Ferguson's PhD thesis. Other cases were analyzed separately using finer approximations than the crude combinatorial ones. Voronoi cells were divided into simplices, minimizing the sum of their volumes, subject to constraints that pieces fit together. This becomes a massive optimization problem with the overall linear structure. However, some quantities (e.g., volumes of simplices, dihedral angles) involved in these linear expressions are inherently non-linear. The problem was formulated so that all non-linear terms are confined to small number of variables, and so that some computer-verified non-linear inequalities could be used as linear substitutes for the non-linear relations. After such linearizations, there were around 10^5 linear optimization problems over about 200 variables and around 2000 constraints that were all solved using a linear programming package.

Hales' proof was computer assisted (all computations were performed using complex programs written in JAVA) and, as we have already described in Section 1, it raised much controversy, since it was not fully verifiable by a human reader in a reasonable time (referees said that they were 99% sure that his proof contains no error). Dissatisfied, Hales started an open, collaborative *flyspeck* [16] project aiming

[16] FlysPecK, the Formal Proof of Kepler, `https://code.google.com/p/flyspeck/`

to formalize his proof using the ITP technology. The formal proof is based on his blueprint [64]. The project was completed in August 2014. It covers both the mathematical part and the computer computations. The classification of the tame graphs was done in Isabelle/HOL, then imported into HOL Light and together with the nonlinear inequalities and linear programs used to prove the KC. This final theorem in HOL/Light states that for every packing V (the set of centers of balls of radius 1), there exists a constant c controlling the error term, such that for every radius R that is at least 1, the number of ball centers inside a spherical container of radius R is at most $\pi \cdot R^3/(3\sqrt{2})$, plus an error term of smaller order.

Mathematical foundations (basic analytical, geometrical and topological machinery) of formalization are based on the work of Harrison [75]. Flyspeck is a very large, organized base of mathematical knowledge (containing more than 14,000 theorems). Such database is a very convenient corpus for further formalization of mathematics, but also for various experiments (e.g., Kaliszyk and Urban showed that it is possible to use automated reasoning tools on the Flyspeck theorems, successfully proving around 40% of them in a push-button mode [81]).

The flyspeck projects involves three computationally intensive parts:

(1) tame graph enumeration;
(2) verification of nonlinear inequalities used for linearization;
(3) bounds for linear programs.

Nipkow, Bauer, and Schultz presented an enumeration of tame graphs in Isabelle/HOL [113]. Potential counter-examples of the KC give rise to tame graphs, and to prove the KC, all tame graphs should be enumerated (creating an Archive of tame graphs) and refuted. Although Hales originally enumerated 5,128 graphs, Nipkow et al. showed that there are only 2,771 non-isomorphic tame graphs (although the Hales' archive was complete, it contained some isomorphic graphs and some graphs that are not tame). The formal proof relies on a modified enumeration of all plane graphs (based on subdivisions of existing graphs), selecting tame graphs, and cutting the search where it cannot lead to tame graphs anymore. What was around 2,200 lines of JAVA code, became around 600 lines of executable Isabelle/HOL code along with 17,000 lines of proofs. Running the proof took around 2.5 hours.

First attempt to verify flyspeck non-linear inequalities was done by Zumkeller in Coq [153], based on Bernstein polynomials and Taylor approximations (for non-polynomial terms). Solovyev and Hales present an alternative tool for formal verification of multivariate non-linear inequalities (both polynomial and non-polynomial) within HOL-Light [132], based on interval arithmetic with Taylor interval approximations. The mathematical part required formalizing the theory of partial derivatives and multivariate Taylor formula with the second-order error term. The tool is based on an informal procedure (ported from C++to OCaml) that is modified to return a certificate which is then formally verified within HOL Light. Since certificate checking involves calculations with both natural and rational numbers, the tool includes efficient procedures for working with naturals in an arbitrary base, and working with floating-point numbers with fixed precision of mantissa. Still, the certificate checking is very expensive. The implemented procedure that performed the verification was around 3,000 times slower than its C++counterpart.

The reported estimate for the verification of the whole flyspeck code was that it will take over 4 years on a single computer [132]. However, some possible optimizations were described and implemented, and the final verification was much faster than reported in [132] (although it still required a parallelization).

First steps towards verification of flyspeck linear programs were performed by Obua in Isabelle/HOL [117]. In his PhD thesis [133], Solovyev devised his own tool, based on the external linear programming package GLPK. Such packages use floating-point operations, while all calculations in formal proofs must be precise, and therefore, formal arithmetic is usually very slow. Solovyev reports that his tool in HOL-Light is much faster (e.g., 5 seconds for a single large linear program, while Obua's method on the flyspeck problems takes between 8 and 67 minutes).

Since the project was done using different proof assistants, it was necessary to share theorems between them. Tame graph classification was imported into HOL-Light using a translation by hand. Verification of more than 23,000 inequalities was performed on Microsoft's Azure cloud platform and took more than 5,000 processor hours (since it was done in parallel on 32 cores, the real time was about 6.5 days).

The flyspeck success confirms that proof assistants are mature enough to cover very complex mathematical results, especially those that heavily rely on computation. As a result of the formalization, Hales found some bugs in the 1998 code, so it was just a happy coincidence that there were no missed cases in the 1998 proof. This is a good example of the importance of formal proof in computer-assisted proof.

5.5. CompCert – a Verified Compiler. Most software verification efforts prove the correctness of the code only on the source code level, and rely on the correctness of compiler, and the underlying operating system and hardware. However, compilers are complex systems that often perform many highly non-trivial symbolic transformations and optimizations, so, it is not uncommon that they contain bugs. In safety-critical, high-assurance computing that must be eliminated.

CompCert[17] [89, 90] is a realistic compiler from *Clight* (a large subset of C) to PowerPC assembly code, implemented and verified within Coq. Its performance was reported to be as twice as fast as `gcc-O0`, and only 12% slower than `gcc-O2`. There are several parts of the compiler that are not verified (e.g., parser of Clight source code, type checker, printer to PowerPC assembly code), but these were straightforward to program, and it is not likely that they contain critical bugs.

The definition of compiler correctness is subtle and relies on the notion of semantic preservation (denoted by $\approx$). In the strongest sense, the source code S is preserved by the compiled code C iff they have exactly the same behaviors that a user can observe (they perform same input-output operations and make the same system calls), i.e., $S \approx C$ iff $\forall B.\ S \Downarrow B \iff C \Downarrow B$, where $X \Downarrow B$ denotes that the code X has the behavior B. Observable behaviors also include cases when the program diverges or goes wrong (e.g., out of bounds array access). This notion of semantic preservation is too strong to be usable (especially when languages have non-deterministic behaviors, or when compilers include optimizations, since some wrong behaviors can be optimized away). Therefore, in CompCert the definition

[17] `http://compcert.inria.fr/`

is relaxed to $\forall B \notin Wrong.\ S \Downarrow B \Longrightarrow C \Downarrow B$, where $Wrong$ is the set of "going wrong" behaviors. Compiler is a function $Comp$ that returns either $OK(C)$ or $Error$. It's verification requires proving $\forall S\ C.\ Comp(S) = OK(C) \Longrightarrow S \approx C$.

Clight supports a wide subset of C, omitting just several C constructs (`long long` and `long double`, `goto` statement, non-structured forms of `switch`, passing structs and unions by value, and functions with variable number of arguments). This subset has a formally defined, big-step operational semantics, which makes precise a number of behaviors left unspecified in the ISO C standard (e.g., datatype sizes, evaluation order). Other undefined behaviors in ISO C are considered wrong in Clight (e.g., null pointer dereferencing, accessing arrays out of bounds).

Compilation is done in stages, and the compiler is composed of 14 passes that go through 8 intermediate languages. To be able to prove the correctness of each phase, each intermediate language required to have its semantics fully formalized within Coq. Implementation of each pass is done in Coq. A big challenge was to rewrite highly imperative algorithms found in compiler textbooks into pure functional style (balanced trees were used as a central data structure and a monadic programming style enabled handling exceptions and state in a legible manner).

The Coq formalization has around 42,000 lines (14% is the implementation, 10% definitions of semantics, and 76% are the proofs), and approximately 3 person-years of work. Unlike many other verification efforts that serve mainly to evaluate the current theorem proving technology, software verification projects like the CompCert deliver final products—industrial strength software intended to be used in real world applications. CompCert is certainly a milestone project in this sense. It has already been used in production of a flight control software [13].

5.6. L4–a Verified Operating System Microkernel. Microkernel architecture of computer operating systems isolates a near-minimum amount of software (the *microkernel*) upon which a full operating system is built. The microkernel is the only software that runs privileged and that has full access to the hardware. Therefore, the size of software whose misbehavior could corrupt the system is kept minimal (microkernel is usually implemented in up to 10,000 lines of code) and the main benefit of such a small trusted code-base is the increase of the overall system security. Microkernels usually implement only basic inter-process communication (IPC), virtual memory, threads, and scheduling, while device drivers, file servers, application IPC, and similar are implemented as separate services running in unprivileged mode (in user space). On the other side of the spectrum are the operating systems with *monolithic kernels* and they include all these services within the kernel, making the kernel huge (e.g., Linux kernel contains over 5 million lines of code). Main drawback of microkernels (especially the early ones) is that they are less efficient (since there is a higher number of switches between the kernel and user programs).

In 2009, a team or researchers from NICTA, Australia led by Klein, announced that they have developed a fully formally verified operating system microkernel seL4[18] [85, 84]. It comprises 8,700 lines of C code and 600 lines of assembler, and its performance is comparable to other high-performance kernels in the L4 family

[18] `http://sel4.systems/`

that it belongs to. SEL4 was designed and verified for the ARMv6-based hardware, and later ported to x86 (this version is not verified). SEL4 is currently used in several security-crucial systems (e.g., high-assurance drones, and it is planned to transfer the technology to Boing's unmanned Little Bird helicopters).

The verification had to reconcile two contrasting approaches. Developers usually take a bottom-up approach—to achieve high performance, low-level hardware details are in focus from the beginning of the design. On the other hand, formal methods require a top-down approach— system is viewed trough abstract models that are easier to verify, abstracting away low-level details. As a compromise, the SEL4 verification used a refinement approach, implemented trough three layers (Figure 12 shows the SEL4 design and verification process).

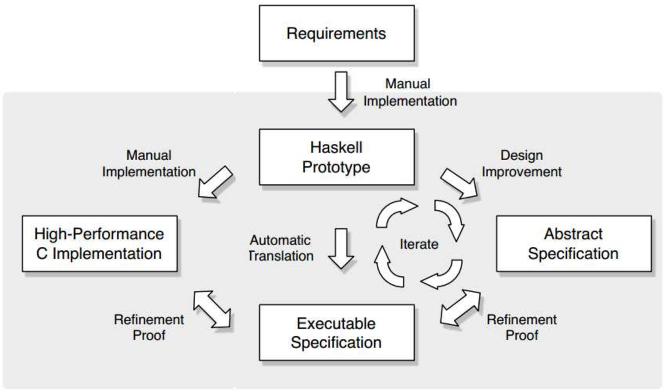

FIGURE 12. The SEL4 design and verification process

On the top there was an abstract, non-executable specification in a theorem prover. It only specified what the system should do, and not how it should be done. Non-determinism was allowed and heavily used (e.g., the scheduler is modeled as a function picking any runnable active or idle thread, and only lower implementation levels introduced specific thread scheduling policies). On the bottom there was the final, hardware aware implementation of the system, written in C and assembly, with high-performance in mind. The compromise between the bottom and the top layer was made through a middle layer—a prototype of the system written in Haskell. The design and implementation of the system went parallel with the verification. Haskell provided a programming language for the OS developers, and an executable model of the system, enabling testing (using a hardware simulator). At the same time it could be automatically translated into a proof-assistant and then reasoned about (and formally connected with the abstract specification). The final C implementation was manually re-implemented based on the Haskell prototype, introducing many hardware specific details and making many micro-optimizations.

Formal verification showed the full functional correctness of the real implementation, with respect to the abstract specification. In the first phase, verification was only up to the C language level, therefore assuming the correctness of the C language compiler, the linker and the underlying hardware. In later phases, the correctness proof included a further refinement step to the level of binary code, either by means of using verified CompCert compiler, or by connecting the semantics of C with the ARM ISA model of semantics of binary ARM code, ported from HOL4 [84].

Connecting the C implementation to abstract specifications required having the semantic of C defined within the prover, and one of the achievements of the project is a faithful formal semantics for a large subset of the C. Imperative features were analyzed using Hoare logic (defined in Isabelle/HOL). Most lemmas show Hoare triples on program statements in each of the specification levels. Around 80% of the properties relate to preserving more than 150 invariants on different specification levels, roughly grouped into 4 categories: low-level memory invariants (e.g., kernel objects are aligned and do not overlap), typing invariants (basically, all reachable references in the kernel point to an object of the expected type), data structure invariants (e.g., all lists are correctly `NULL` terminated), and algorithmic invariants (e.g., the idle thread is always in the state `IDLE`, and there are no other threads in this state). Invariants are usually global, and are preserved throughout the kernel execution. However, they can be temporarily invalidated, making the proofs harder.

OS code is usually not structured in a modular way, suitable for verification (e.g., usually global variables are used, there is a lot of side-effects). Additional problems come from concurrency and non-determinism (only uniprocessor systems were covered, but even on such systems concurrency is present, due to interrupts from I/O devices). As the code evolved in parallel to its verification, the requirements that verification posed forced the designers of to think of the simplest and cleanest way of achieving their goals, leading to better design and less bugs.

The sheer size of the formalization is impressive—the formalization includes more than 450,000 lines of code. Interactive verification was the only viable solution, since state-of-the-art automatic formal methods (e.g., model checking, static analysis, shape analysis) can only show some specific properties and absence of certain errors, but full functional correctness is far out of their reach. Authors' main findings show that "verification focus improved the design and was surprisingly often not in conflict with achieving performance" [85]. Also "formally verified software is actually less expensive than traditionally engineered 'high-assurance' software, yet provides much stronger assurance" [84]. This opens many new perspectives for performing similar verification efforts to other high-assurance software, especially, when the automation in proof-assistants becomes increased.

6. ITP in Serbia: a Personal Perspective

Although ITP is not a mainstream research topic in Serbia, several researchers have successfully used ITP in their work. We list their work and apologize to the authors of results that we are not aware of. Maksimović formalized a large body of work about probability logics in Coq [93], and is currently working on the PiCoq project and formalizations concerning higher-order π-calculi and their properties. Janičić, Narboux and Quaresma have formalized in Coq properties of the Area method for automated theorem proving in geometry [79]. Stojanović, Janičić, and Pavlović have implemented a coherent logic based geometry theorem prover capable of producing both readable and formal proofs (in Isabelle/Isar format) [136]. Stojanović, Janičić, Narboux, and Bezem have created a representation of proofs for coherent logic and a corresponding file format, from which proofs checkable by Isabelle/HOL and Coq can be automatically generated [135]. Todorović, Zeljić, Marinković, Glavan, and Ognjanović were working towards a formal description of the Chord Protocol using Isabelle/HOL.

In the following subsection we describe some of the formalization that the author and his colleagues from the Automate Reasoning GrOup (ARGO) at Faculty of Mathematics, University of Belgrade, have done using Isabelle/HOL.

6.1. Formal Verification of SAT Solvers. Formalization, implementation, and applications of SAT solvers was a subject of the author's PhD thesis [94]. The problem of checking propositional satisfiability (SAT) [16] is one of the central problems in computer science, used in many practical applications (e.g., software and hardware verification, constraint solving, electronic design automation). Most state-of-the-art complete SAT solvers are essentially based on a branch and backtrack procedure called Davis-Putnam-Logemann-Loveland or the DPLL procedure [35, 34] working on formulae in conjunctive normal form (CNF). Modern SAT solvers usually also employ (i) several conceptual, high-level algorithmic additions to the original DPLL procedure, (ii) smart heuristic components, and (iii) better low-level implementation techniques. Thanks to these, spectacular improvements have been made and nowadays SAT solvers can decide satisfiability of CNF formulae with tens of thousands of variables and millions of clauses.

SAT solvers became very complex systems (their implementation usually requires thousands lines of source-code) and it is very hard to establish their correctness. Since SAT solvers are used in applications that are very sensitive, their misbehavior could be both financially expensive and dangerous from the aspect of security. Ensuring trusted SAT solving can be achieved in two ways.

1. In the first approach, SAT solver is modified to produce a certificate for each solved instance. For satisfiable instances certificates are their models (truth assignments of variables), while for unsatisfiable ones, certificates are given by proofs (usually represented as resolution-trees deriving the empty clause). Certificates are independently checked by proof-checkers that are much simpler than the SAT solvers, and are either verified by a human inspection, or are formally verified within a proof-assistant.
2. In the second approach, SAT solvers themselves (their underlying algorithms and data-structures) are formally verified within a proof assistant.

Both approaches have their advantages and drawbacks. The first one is much easier to implement, and is more robust (as it is not sensible to changes in the SAT solver implementation, as long as the solver can emit certificates). The second one removes the overhead of generating and checking certificates (that is not unmanageable, but effects the solving efficiency and storage, since proofs can be very large objects). It also helps in better theoretical understanding of how and why the SAT solvers work. Also, verified SAT solvers or their modules can serve as trusted kernel checkers for verifying results of other untrusted tools.

We have implemented the second approach in Isabelle/HOL [94]. First, we (Marić and Janičić) verified the original formulation of the DPLL algorithm [97]. Then, we covered state-of-the-art SAT solving algorithms and data structures (most notably, the two watch unit propagation). The formal specification and verification of SAT solvers was made in several ways (illustrated in Figure 13, each with an appropriate verification paradigm and its own advantages and disadvantages).

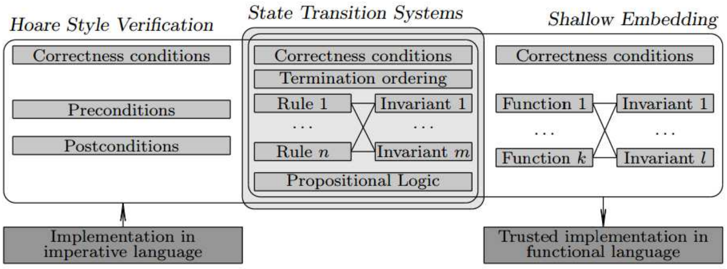

Figure 13. SAT solver verification

Verification of abstract state transition systems. *Abstract state transition systems (ASTS)* specify program behavior as transitions between states, that follow precisely defined transition rules. ASTS are mathematical objects, so it is relatively easy to make their formalization within higher-order logic and to formally reason about them. Also, their verification can be a key building block for other (more detailed) verification approaches. On the down side, ASTS can hide many details present in implementations and that they are not directly executable.

State transition systems describe the top-level architecture of the modern DPLL-based SAT solvers (and related SMT solvers) [86, 112]. We (Marić and Janičić) have formally verified a series of ASTS for SAT, with increasing level of detailedness [98]. As an illustration, we show the definition of a very simple ASTS performing a basic, backtracking search. Let F be the CNF formula currently being solved. The state consist of a list of literals M (the current partial valuation). Literals that arbitrary choices (decisions) made during the search are marked (as l^d). Literals that are not marked (as l) are inferred from the previous decisions and F. The system considers the following three rules.

$$\mathsf{Decide}\ M_1\ M_2 \quad \text{iff}\ \exists l.\ \mathsf{var}\ l \in \mathsf{vars}\ F \wedge l \notin M_1 \wedge \bar{l} \notin M_1 \wedge M_2 = [M_1, l^d]$$

$$\mathsf{UnitPropagate}\ M_1\ M_2 \quad \text{iff} \quad \exists\ c\ l.\ c \in F \wedge \mathsf{unitClause}\ c\ l\ M_1 \wedge M_2 = [M_1, l]$$

$$\mathsf{Backtrack}\ M_1\ M_2\ \text{iff}\ \ \exists\ l\ M\ M'.\ M_1 \vDash \neg F\ \ \wedge\ \ M_1 = [M, l^d, M'] \wedge \\ \mathsf{decisions}\ M' = \{\}\ \ \wedge\ \ M_2 = [M, l]$$

Decide adds a previously undefined literal to M. UnitPropagate performs inference using unit-clauses (clauses that have all the literals false wrt. M, except one literal l that is undefined wrt. M). Backtrack is applied when M and F are mutually inconsistent, and it undoes the last decision and inferred literals after it, inferring the opposite literal of the last decision. Transition relation between states is defined as the union of the Decide, UnitPropagate, and the Backtrack relation. The central theorem states that all sequences of states starting from the empty initial state $M = [\,]$ terminate. They reach a state where no further transition ca be made, and if in that state the initial formula is unsatisfiable iff in that state it holds $M \vDash \neg F$.

Verified implementation within a proof assistant. Program can be shallowly embedded into HOL (i.e., specified within the HOL logic of a proof assistant, regarded as a purely functional programming language). The level of details in the specifications can incrementally be increased (e.g., by using a datatype refinement). Having the specification inside the logic, its correctness can be proved by using the standard mathematical apparatus (induction and equational reasoning). Once the program is defined within the proof assistant, it is possible to verify it without the need for a formal model of the operational or semantics of the programming language. Executable functional programs can be generated from the specification, by means of code extraction, and the extracted code can be trusted with a very high level of confidence. On the other hand, the approach requires building a fresh implementation of a SAT solver within the logic. Since HOL is purely functional, it is unadapted to modeling imperative data-structures and their destructive updates. Special techniques must be used to have mutable data-structures and, consequently, an efficient generated code [24]. The author implemented and proved the total correctness of a SAT solver by shallow embedding into Isabelle/HOL [96].

Verification of the real implementations. The most demanding approach is to directly verify the real SAT solver code. Since SAT solvers are usually implemented in imperative programming languages, verifying the correctness of implementation can be done within the Hoare logic—a formal system for reasoning about imperative programs. The program behavior can then be described in terms of preconditions and postconditions. Since real code is overwhelmingly complex, simpler approximations are often made and given in pseudo-programming languages. This can significantly simplify the implementation, but leaves a gap between the correctness proof and the real implementation.

The author has given a detailed description of a modern SAT solver using pseudocode, and has proved its correctness within the Hoare logic [95]. This work also serves as a tutorial introduction to modern SAT solving technology.

6.2. Formal Verification of SMT Solvers. SMT solvers [16, 112, 86] combine SAT solvers with decision procedures for specific theories in first-order logic. One of the most important SMT theories is linear arithmetic (over reals, rationals or integers). Usually, only the quantifier-free fragment is considered and SMT solvers determine the satisfiability of Boolean combinations of linear equalities and inequalities. Most efficient decision procedures for this fragment are based on an incremental versions of Simplex algorithm [37]. Similar to SAT, trusted SMT solving can be realized either by certificates checking or by verifying SMT solvers themselves.

We (Spasić and Marić) have formalized the version of Simplex used in SMT [134]. The formalization is based on the process of stepwise-refinement, starting from a very abstract solver specification, going through a long sequence of very small refinement steps, and finishing with a fully executable implementation. Refinement is based on Isabelle/HOL's mechanism of locales [82], used to specify functions. For example, the starting specification is given by the following locale.

locale *Solve*=
– Decide if the given list of constraints is satisfiable. Return the satisfiability status and, in the satisfiable case, one satisfying valuation.
fixes $solve :: constraint\ list \Rightarrow bool \times rat\ valuation\ option$
–If the status $True$ is returned, then returned valuation satisfies all constraints.
assumes $let\ (sat,\ v){=}solve\ cs\ in\ sat \longrightarrow the\ v \vDash_{cs} cs$
– If the status False is returned, then constraints are unsatisfiable.
assumes $let\ (sat,\ v){=}solve\ cs\ in\ \neg sat \longrightarrow \neg(\exists v.\ v \vDash_{cs} cs)$

The decision to use a stepwise-refinement approach, with many small steps, enormously simplified reasoning about the procedure. Initially, we did a formalization by formulating the whole algorithm and reasoning about it at once, such monolith approach required proofs that are several times longer and much harder to grasp. Stepwise refinement makes the formalization modular and it is much easier to make changes to the procedure. We have also payed special attention to symmetric cases in the proof, and by introducing suitable generalization, totally avoided to need to handle symmetric cases separately (as it was the case in the pen-and-paper proof).

Although our implementation is purely functional, it was not much slower than imperative C++implementations, since most of the time is spent in computations with arbitrary precision rational numbers (which is very expensive). On the universally quantified fragment, the formalized simplex based procedure significantly outperformed similar formalized procedure based on the quantifier elimination.

6.3. Formalization related to Frankl's Conjecture. *Union-closed set Conjecture* formulated by Péter Frankl in 1979 (therefore also called Frankl's Conjecture), states that for every family of sets closed under unions, there is an element contained in at least half of the sets (or, dually, in every family of sets closed under intersections, there is an element contained in at most half of the sets). Up to the best of our knowledge, the problem is still open, and that is not because of lack of interest—in a recent survey [23] Bruhn and Schaudt list over 50 published research articles on the topic. The conjecture has been confirmed for many finite special cases. It has been proved that the conjecture holds for families such that their union has at most $m = 11$ elements[19], and for families containing $n \leqslant 44$ sets[20].

In our work (Marić, Vučković, and Živković), we addressed some finite cases of the Union-closed Conjecture within Isabelle/HOL, with the final goal to confirm that it holds for $m = 12$ elements (that was previously established by Živković and Vučković using unverified JAVA programs). First, we focused on *Frankl-complete (FC)* families. A family F_c is an FC-family if in every union-closed family $F \supseteq F_c$,

[19] An unpublished article by Živković and Vučković describes the use of computer programs to check the case of $m = 12$ elements.

[20] This becomes $n \leqslant 48$ by the result of Živković and Vučković

one of the elements of F_c is abundant (is contained in at least half the sets). Poonen [124] gives a necessary and sufficient conditions for a family to be FC. We have formally proved it within Isabelle/HOL (in a slightly modified version, that considers only naturals instead of reals, making it more easier to use in computations). For illustration, we print it here (after giving some preliminary definitions).

Definition 1. Let $F \uplus F' = \{A \cup B.\ A \in F \wedge B \in F'\}$. A family F is *union closed* if $F \uplus F \subseteq F$. A family F such that $\bigcup F \subseteq \bigcup F_c$ is a *union closed extension* of a family F_c if it is union-closed and $F \uplus F_c \subseteq F$. A function $w : X \to \mathbb{N}$ is a *weight function on* $A \subseteq X$, iff $\exists a \in A.\ w(a) > 0$. *Share of a set* A *wrt. a weight function* w *and a set* X, denoted by $\bar{w}_X(A)$, is the value $2 \cdot w(A) - w(X)$. *Share of a family* F *wrt.* w *and a set* X, denoted by $\bar{w}_X(F)$, is the value $\sum_{A \in F} \bar{w}_X(A)$.

Theorem 1. *A family* F_c *is an FC-family iff there is a weight function* w *such that shares (wrt.* w *and* $\bigcup F_c$*) of all union closed extension of* F_c *are nonnegative.*

We have formalized these results and formulated effective procedures that prove that a given family is FC (or is not FC), following the proof-by-computation paradigm. To check that a family is an FC-family, Poonen's theorem requires listing all its union closed extensions, which can be very inefficient. Therefore, we had to formalize an algorithm that traverses the space made of all extensions in a systematic way, pruning the significant parts where it can be deduced in advance that they cannot contain a family with a negative share. Such procedure is originally described by Živković and Vučković and implemented in JAVA, but we formalized it in Isabelle/HOL, and proved it correct, connecting it formally with the statement of the Poonen's theorem [100]. We applied those techniques to investigate uniform FC families (where all members have the same number of elements) [100]. We confirmed all previously-known and discovered some new uniform FC-families (e.g., all families containing four 3-element sets whose union is contained in a 7-element set).

Compared to the prior pen-and-paper work, the computer assisted approach significantly reduces the complexity of mathematical arguments behind the proof and employs computing-machinery in doing its best—quickly enumerating and checking a large search space. This enables a general framework for checking various FC-families, without the need of employing human intellectual resources in analyzing specificities of separate families. Compared to the work of Zivković and Vučković, apart from achieving the highest level of trust possible, the significant contribution of the formalization is the clear separation of mathematical background and combinatorial search algorithms, not present in earlier work. Also, separation of abstract properties of algorithms and technical details of their implementation significantly simplifies reasoning about their correctness and brings them much closer to classic mathematical audience, not inclined towards computer science.

In our current work we give a full characterization of all FC-families formed over a six-element domain. In our future work, we to apply developed techniques for FC-families, and to prove the 12-element case formally, within Isabelle/HOL.

6.4. Formalization of the Complex Plane Geometry.

Deep connections between complex numbers and geometry had been carefully studied centuries ago. Fundamental objects that are investigated are the complex plane (extended by a single infinite point), its objects (points, lines and circles), and groups of transformations that act on them (e.g., inversions and Möbius transformations). We (Marić and Petrović) have treated the geometry of complex numbers formally and presented its Isabelle/HOL formalization [99]. Apart from applications in formalizing mathematics and in education, that work serves as a ground for formally investigating various non-Euclidean geometries and their intimate connections.

The crucial step in our formalization was our decision to use the algebraic representation of all relevant objects (e.g., vectors of homogeneous coordinates, matrices for Möbius transformations, Hermitian matrices for circlines). Although this is not a new approach (e.g., Schwerdtfeger's classic book [128] consistently follows this approach), it is not so common in the literature (and in the course material available online). Instead, other, more geometrically oriented approaches prevail. We have tried to follow that kind of geometric reasoning in our early work on this subject, but we have encountered many difficulties and did not have so much success.

Our formalization makes heavy use of quotients (supported by the lifting/transfer package for Isabelle [78]). For example, the extended complex plane is introduced as a projective space of dimension one over $\mathbb{C}$ (using homogeneous coordinates). Let $\mathbb{C}^2 \setminus \{(0,0)\}$ be the set of pairs of complex numbers not both equal to zero. A relation $\approx_{C2}$ introduced on that set, such that

$$(z_1, w_1) \approx_{C2} (z_2, w_2) \longleftrightarrow (\exists\, k \in \mathbb{C}.\ k \neq 0 \wedge z_1 = k \cdot z_2 \wedge w_1 = k \cdot w_2).$$

The extended complex plane $\overline{\mathbb{C}}$ is then defined as the quotient $(\mathbb{C}^2 \setminus \{(0,0)\}) / \approx_{C2}$ (its elements are the equivalence classes). There is a standard embedding of $\mathbb{C}$ into $\overline{\mathbb{C}}$ (mapping each element z to the equivalence class of $(z, 1)$, with element of that class called the homogeneous coordinates of z). The only element of $\overline{\mathbb{C}}$ that is not of this form is the infinite point ∞ (the equivalence class of $(z, 0)$). All arithmetic operations are defined on $\overline{\mathbb{C}}$ (e.g., if (z_1, w_1) and (z_2, w_2) are two representatives then the sum of their equivalence classes is the equivalence class containing the representative $(z_1w_2 + z_2w_1, w_1w_2)$). The circle inversion (a very important geometrical operation) is also defined and its main properties are proved. We have formalized other standard models of the extended complex plane (e.g., the Riemann sphere and the stereographic projection).

Fundamental group of transformations of $\overline{\mathbb{C}}$ are the Möbius transformations. They are defined as equivalence classes of regular 2×2 complex matrices (matrices proportional by a complex non-zero factor are equivalent). Elements of the Möbius group act on the elements of $\overline{\mathbb{C}}$ (the representation matrix is multiplied with the homogeneous coordinates of a point to obtain the homogeneous coordinates of its image). Möbius group has many important subgroups that were formally analyzed (e.g., Euclidean similarities, unit disc automorphisms).

Fundamental objects of $\overline{\mathbb{C}}$ are generalized circles (or circlines, as they are either Euclidean circles or lines). They are defined as equivalence classes of Hermitian non-zero matrices (matrices proportional by a real non-zero factor are equivalent). Each matrix defines a set of points in $\overline{\mathbb{C}}$ (a point with homogeneous coordinates z lies on the circline with a representation matrix H if the quadratic form z^*Hz is zero). Oriented circlines, their interiors (discs) and exteriors were also considered.

Möbius transformations act on (oriented) circlines and one of the basic theorems that was formally shown is that they preserve circlines (a circline is always mapped onto a circline). Möbius transformations are conformal and preserve angles between circlines.

Besides having a well-developed theory of complex-plane geometry, this formalization gave us several important conclusions. For example, the angle preservation property shows how convenient is to have purely algebraic characterizations. If $\left(\begin{smallmatrix} A_1 & B_1 \\ C_1 & D_1 \end{smallmatrix}\right)$ and $\left(\begin{smallmatrix} A_2 & B_2 \\ C_2 & D_2 \end{smallmatrix}\right)$ are matrices representing the circlines, the cosine of the angle between circlines can be defined as

$$-(A_1D_2 - B_1C_2 + A_2D_1 - B_2C_1)/\bigl(2\sqrt{(A_1D_1 - B_1C_1)\cdot(A_2D_2 - B_2C_2)}\,\bigr).$$

This is quite different from the standard definition that considers tangent vectors and the angle between them. Algebraic definition of angle gives an easy and elegant proof of the angle preservation. However, that definition is non-standard, far from intuition, and it does not provide satisfactory explanations. Therefore, for educational purposes, in all such cases we have formalized standard (geometric) definitions and proved that they are equivalent to the non-standard (algebraic) ones.

Another important conclusion is that in formal documents, case analysis should be avoided and extensions that help avoiding it should be pursued whenever possible (e.g., it was much better to use the homogeneous coordinates instead of a single distinguished infinity point, it was much simpler to work with circlines than to distinguish between circles and lines, etc.). Keeping different models of the same concept (e.g., homogeneous coordinates and the Riemann sphere) also helps, as some proofs are easier in one, and some proofs are easier in other models.

Isabelle's automation was quite powerful in equational reasoning about complex numbers. However, tedious reasoning was sometimes required, especially when switching between real and complex numbers. Such type-conversions are not present in informal texts, and better automation of reasoning about them would be welcome. In the presence of inequalities the automation was weak, and many things that would be considered trivial in informal texts, had to be proved manually.

7. Conclusions

We have presented an overview of interactive theorem proving. Proof-assistants have been actively developed since 1960s, and today, state-of-the-art systems are capable to prove very complex mathematical theorems and verify very complex software systems. However, doing formalization is still hard, and there is still a steep learning curve, preventing adopting this style of work in a wider mathematical community. Formal verification still needs: "better libraries of background mathematics, better means of sharing knowledge between the systems, better automated support, better means of incorporating and verifying computation, better means of storing and searching for background facts, and better interfaces, allowing users to describe mathematical objects and their intended uses, and otherwise convey their mathematical expertise" [9]. However, a large community of researchers is addressing all these problems, and there are optimistic estimates that by the middle of the century, mechanically verified mathematics will be commonplace.

Acknowledgement. This work has been partially supported by the Serbian Ministry of Education and Science grant ON174021. The author thanks Predrag Janičić and Silvia Ghilezan for valuable suggestions about the preliminary versions of the manuscript.

References

[1] K. Appel and W. Haken, *Every map is four colourable*, Bull. Am. Math. Soc. **82** (1976), 711–712.

[2] M. Armand, G. Faure, B. Grégoire, C. Keller, L. Thery, and B. Werner, *A Modular Integration of SAT/SMT Solvers to Coq through Proof Witnesses*, in: J. P. Jouannaud and Z. Shao (eds.), *Certified Programs and Proofs*, LNCS 7086, Springer, Berlin–Heidelberg, 2011, 135–150.

[3] M. Aschbacher and S. D. Smith, *The Classification of Quasithin Groups*, Mathematical surveys and monographs, American Mathematical Society, 2004.

[4] A. Asperti, *Proof, message and certificate*, in: J. Jeuring et al. (eds.), *Calculemus 2012, Held as Part of Intelligent Computer Mathematics – CICM*, LNCS 7362, Springer, Berlin-Heidelberg, 2012, 17–31.

[5] A. Asperti, W. Ricciotti, C. Sacerdoti Coen, and E. Tassi, *The Matita interactive theorem prover*, in: N. Bjrner and V. Sofronie-Stokkermans (eds.), *Automated Deduction – CADE-23*, LNCS 6803, Springer, Berlin–Heidelberg, 2011, 64–69.

[6] D. Aspinall, *Proof general: A generic tool for proof development*, in: J. Katoen and P. Stevens (eds.), *Tools and Algorithms for the Construction and Analysis of Systems – TACAS 2000, Held as Part of ETAPS 2000*, Springer, Berlin-Heidelberg, 2000, 38–42.

[7] J. Avigad, *Type inference in mathematics*, Bull. EATCS **106** (2012), 78–98.

[8] J. Avigad, K. Donnelly, D. Gray, and P. Raff, *A formally verified proof of the prime number theorem*, ACM Trans. Comput. Log. **9(1)** (2007).

[9] J. Avigad and J. Harrison, *Formally verified mathematics*, Commun. ACM **57(4)** (2014), 66–75.

[10] J. Avigad, J. Hölzl, and L. Serafin, *A formally verified proof of the central limit theorem*, *CoRR*, abs/1405.7012, 2014.

[11] H. Barendregt, *Introduction to generalized type systems*, J. Funct. Program. **1(2)** (1991), 125–154.

[12] H. Barendregt and H. Geuvers, *Proof-assistants using dependent type systems*, in: A. Robinson and A. Voronkov (eds.), *Handbook of Automated Reasoning*, Elsevier, Amsterdam, 2001, 1149–1238.

[13] R. Bedin França, S. Blazy, D. Favre-Felix, X. Leroy, M. Pantel, and J. Souyris, *Formally verified optimizing compilation in ACG-based flight control software*, in: *ERTS2 2012: Embedded Real Time Software and Systems*, Paris, 2012.

[14] H. Bender, G. Glauberman, and W. Carlip, *Local analysis for the odd order theorem*, London Mathematical Society LNS, Cambridge University Press, 1994.

[15] Y. Bertot and P. Castéran, *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions*, Springer, Berlin–Heidelberg, 2004.

[16] A. Biere, A. Biere, M. Heule, H. van Maaren, and T. Walsh, *Handbook of Satisfiability*, IOS Press, Amsterdam, 2009.

[17] J. C. Blanchette, S. Böhme, and L. C. Paulson, *Extending Sledgehammer with SMT solvers*, in: N. Bjrner and V. Sofronie-Stokkermans (eds.), *Automated Deduction – CADE-23*, LNCS 6803, Springer, Berlin-Heidelberg, 2011, 116–130.

[18] S. Böhme, *Proving Theorems of Higher-Order Logic with SMT Solvers*, PhD thesis, Technische Universität München, 2012.

[19] S. Böhme and T. Nipkow, *Sledgehammer: Judgement day*, in: J. Giesl and R. Hähnle (eds.), *Automated Reasoning – IJCAR 2010*, LNAI 6173, Springer, Berlin–Heidelberg, 2010, 107–121.

[20] S. Böhme and T. Weber, *Fast LCF-style proof reconstruction for Z3*, in: M. Kaufmann and L. Paulson (eds.), *Interactive Theorem Proving*, LNCS 6172, Springer, Berlin–Heidelberg, 2010, 179–194.

[21] S. Boutin, *Using reflection to build efficient and certified decision procedures*, in: M. Abadi and T. Ito (eds.), *Theoretical Aspects of Computer Software*, LNCS 1281, Springer, Berlin–Heidelberg, 1997, 515–529.

[22] R. S. Boyer and J. S. Moore, *Proving theorems about LISP functions*, *J. ACM* **22(1)** (1975), 129–144.

[23] H. Bruhn and O. Schaudt, *The journey of the union-closed sets conjecture*, Graphs Comb. **31(6)**(2015), 2043–2074.

[24] L. Bulwahn, A. Krauss, F. Haftmann, L. Erkök, and J. Matthews, *Imperative functional programming with Isabelle/HOL*, in: O. A. Mohamed, C. Muñoz, and S. Tahar (eds.), *Theorem Proving in Higher Order Logics 2008*, LNCS 5170, 2008, 134–149.

[25] A. Chlipala, *Certified Programming with Dependent Types.* MIT Press, 2011, `http://adam.chlipala.net/cpdt/`.

[26] A. Church, *A Note on the Entscheidungsproblem*, J. Symb. Log. **1(1)** (1936), 40–41.

[27] A. Church, *A formulation of the simple theory of types*, J. Symb. Log. **5(2)** (1940), 56–68.

[28] A. Church, *The Calculi of Lambda-conversion*, Annals of Mathematics Studies, Princeton University Press, 1985.

[29] R. L. Constable, S. F. Allen, H. M. Bromley, W. R. Cleaveland, J. F. Cremer, R. W. Harper, D. J. Howe, T. B. Knoblock, N. P. Mendler, P. Panangaden, J. T. Sasaki, and S. F. Smith, *Implementing Mathematics with the Nuprl Proof Development System*, Prentice-Hall, Upper Saddle River, USA, 1986.

[30] T. Coquand and G. Huet, *The calculus of constructions*, Inf. Comput. **76(2-3)** (1988), 95–120.

[31] P. Corbineau, *A declarative language for the Coq proof assistant*, in: M. Miculan, I. Scagnetto, F. Honsell, *Types for Proofs and Programs*, LNCS 4941, Springer, Berlin–Heidelberg, 2008, 69–84.

[32] C. Cornaros and C. Dimitracopoulos, *The prime number theorem and fragments of PA*, Arch. Math. Logic **33** (1994), 265–281.

[33] L. Cruz-Filipe, *A Constructive Formalization of the Fundamental Theorem of Calculus*, in: H. Geuvers and F. Wiedijk (eds.), *Types for Proofs and Programs*, LNCS 2646, Springer, Berlin–Heidelberg, 2003, 108–126.

[34] M. Davis, G. Logemann, and D. Loveland, *A machine program for theorem-proving*, Commun. ACM. **5(7)** (1962), 394–397.

[35] M. Davis and H. Putnam, *A computing procedure for quantification theory*, J. ACM **7(3)** (1960), 201–215.

[36] F. Dechesne and R. Nederpelt, *N. G. de Bruijn and his road to Automath, the earliest proof checker*, Math. Intell. **34(4)** (2012), 4–11.

[37] B. Dutertre and L. de Moura, *A fast linear-arithmetic solver for DPLL(T)*, in: T. Ball and R.B. Jones (eds.), *Computer Aided Verification*, LNCS 4144, Springer, Berlin–Heidelberg, 2006, 81–94.

[38] W. Feit and J. G. Thompson, *Solvability of groups of odd order*, Pac. J. Math. **13(3)** (1963), 775–1029.

[39] A. P. Felty, E. L. Gunter, J. Hannan, D. Miller, G. Nadathur, and A. Scedrov, *Lambda-prolog: An extended logic programming language*, in: E. Lusk and R. A.. Overbeek (eds.), *Automated Deduction – CADE 1988*, LNCS 310, 1988, 754–755.

[40] G. Gentzen, *Untersuchungen über das logische Schließen. I.* Math. Z. **39(1)** (1935), 176–210.

[41] G. Gentzen, *Untersuchungen über das logische Schließen. II.* Math. Z. **39(1)** (1935), 405–431.

[42] H. Geuvers, F. Wiedijk, and J. Zwanenburg, *A constructive proof of the fundamental theorem of algebra without using rationals*, in: P. Callaghan et al. (eds.), *Types for Proofs and Programs*, LNCS 2277, Springer, Berlin–Heidelberg, 2002, 96–111.

[43] S. Ghilezan and S. Likavec, *Computational interpretations of logics*, Zb. Rad., Beogr. **12(20)** (2009), 159–215.

[44] J. Girard, *Interprétation fonctionnelle et élimination des coupures de l'arithmétique d'ordre supérieur*, PhD thesis, Université Paris 7, 1972.

[45] G. Gonthier, *A computer checked proof of the Four-Colour Theorem*, technical report, Microsoft Research, 2005.

[46] G. Gonthier, *Formal Proof – The Four-Color Theorem*, Notices Am. Math. Soc. **55(11)** (2008), 1382–1393.

[47] G. Gonthier, *The Four Colour Theorem: Engineering of a Formal Proof*, in: D. Kapur (ed.), *Computer Mathematics – ASCM 2007*, LNCS 5081, Springer, Berlin–Heidelberg, 2008, 333–333.

[48] G. Gonthier, A. Asperti, J. Avigad, Y. Bertot, C. Cohen, F. Garillot, S. Le Roux, A. Mahboubi, R. O'Connor, S. Ould Biha, I. Pasca, L. Rideau, A. Solovyev, E. Tassi, and L. Théry, *A machine-checked proof of the odd order theorem.* in: S. Blazy, C. Paulin-Mohring, and D. Pichardie (eds.), *Interactive Theorem Proving*, LNCS 7998, Springer, Berlin–Heidelberg, 2013, 163–179.

[49] G. Gonthier, A. Mahboubi, and E. Tassi, *A Small Scale Reflection Extension for the Coq system.* Research Report RR-6455, INRIA, 2008.

[50] M. Gordon, *Introduction to the HOL system*, in: *International Workshop on the HOL Theorem Proving System and its Applications*, IEEE, 1991, 2–3.

[51] M. J. C Gordon, *From LCF to HOL: A short history*, in: G. Plotkin, C. Stirling, and M. Tofte (eds.), *Proof, Language, and Interaction*, MIT Press, Cambridge, MA, USA, 2000, 169–185.

[52] M. J. C. Gordon, R. Milner, and C. P. Wadsworth, *Edinburgh LCF*, LNCS 78, Springer, Berlin–Heidelberg, 1979.

[53] J. F. Grcar, *Errors and corrections in the mathematical literature*, Notices Am. Math. Soc., **60(4)** (2013), 418–432.

[54] J. R. Guard, F. C. Oglesby, J. H. Bennett, and L. G. Settle, *Semi-automated mathematics*, J. ACM **16(1)** (1969), 49–62.

[55] F. Haftmann, *Code Generation from Specifications in Higher Order Logic*, PhD thesis, Technische Universität München, 2009.

[56] F. Haftmann, A. Krauss, O. Kuncar, and T. Nipkow, *Data Refinement in Isabelle/HOL*, in: S. Blazy, C. Paulin-Mohring, and D. Pichardie (eds.), *Interactive Theorem Proving*, LNCS 7998, Springer, Berlin–Heidelberg, 2013, 100–115.

[57] F. Haftmann and T. Nipkow, *Code generation via higher-order rewrite systems*, in: M. Blume, N. Kobayashi, and G. Vidal (eds.), *Functional and Logic Programming (FLOPS 2010)*, LNCS 6009, Springer, Berlin–Heidelberg, 2010, 103–117.

[58] F. Haftmann and M. Wenzel, *Constructive Type Classes in Isabelle*, in: T. Altenkirch and C. McBride (eds.), *Types for Proofs and Programs*, LNCS 4502, Springer, Berlin–Heidelberg, 2007, 160–174.

[59] T. C. Hales, *Cannonballs and honeycombs*, Notices Am. Math. Soc. **47** (2000), 440–449.

[60] T. C. Hales, *A proof of the Kepler conjecture*, Annals of Math. **162** (2005), 1063–1183.

[61] T. C. Hales, *The Jordan curve theorem, formally and informally*, Am. Math. Mon. **114(10)** (2007), 882–894.

[62] T. C. Hales, *Formal proof*, Notices Am. Math. Soc. **55(11)** (2008), 1370–1380.

[63] T. C. Hales, editor, *Special issue on Formal Proof*, Notices Am. Math. Soc **55(11)** (2008).

[64] T. C. Hales, *Dense Sphere Packings: A Blueprint for Formal Proofs*, Cambridge University Press, New York, 2012.

[65] T. C. Hales, *Developments in formal proofs*, *CoRR*, abs/1408.6474, 2014.

[66] T. C. Hales and S. McLaughlin, *The dodecahedral conjecture.* J. Am. Math. Soc. **23(2)** (2010), 299–344.

[67] R. Harper, F. Honsell, and G. Plotkin, *A framework for defining logics*, J. ACM **40(1)** (1993), 143–184.

[68] J. Harrison, *HOL light: A tutorial introduction*, in: M. Srivas and A. Camilleri (eds.), *Formal Methods in Computer-Aided Design*, Springer, Berlin–Heidelberg, 1996, 265–269.

[69] J. Harrison, *Theorem Proving with the Real Numbers*, Distinguished Dissertations, Springer, Berlin–Heidelberg, 1998.

[70] J. Harrison, *Automated and interactive theorem proving*, in: O. Grumberg, T. Nipkow, and C. Pfaller (eds.), *Formal Logical Methods for System Security and Correctness*, IOS Press, 2008, 111–147.

[71] J. Harrison, *Formal proof — theory and practice*, Notices Am. Math. Soc. **55(11)** (2008), 1395–1406.

[72] J. Harrison, *Formalizing an analytic proof of the Prime Number Theorem*, J. Autom. Reasoning **43(3)** (2009), 243–261.

[73] J. Harrison, *Handbook of Practical Logic and Automated Reasoning*, Cambridge University Press, New York, 2009.

[74] J. Harrison, *HOL Light: An Overview*, in: S. Berghofer et al. (eds.), *Theorem Proving in Higher Order Logics*, LNCS 5674, Springer, Berlin–Heidelberg, 2009, 60–66.

[75] J. Harrison, *The HOL light theory of Euclidean space*, J. Autom. Reasoning **50(2)** (2013), 173–190.

[76] W. A. Howard, *The formulas-as-types notion of construction*, in: *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus, and Formalism*, Academic Press, 1980, 479–490. Reprint of 1969 article.

[77] G. Huet and H. Herbelin, *30 years of research and development around Coq*, in: *Proc. 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages – POPL '14*, ACM, New York, 2014, 249–250.

[78] B. Huffman and O. Kunčar, *Lifting and Transfer: A Modular Design for Quotients in Isabelle/HOL*, in: G. Gonthier and M. Norrish (eds.), *Certified Programs and Proofs*, LNCS 8307, Springer, Berlin–Heidelberg, 2013, 131–146.

[79] P. Janičić, J. Narboux, and P. Quaresma, *The area method – A recapitulation*, J. Autom. Reasoning **48(4)** (2012), 489–532.

[80] L. S. B. Jutting, *Checking Landau's "Grundlagen" in the Automath system*, Mathematical Centre Tracts, Mathematisch Centrum, Amsterdam, 1979.

[81] C. Kaliszyk and J. Urban, Learning-assisted automated reasoning with Flyspeck. J. Autom. Reasoning **53(2)** (2014), 173–213.

[82] F. Kammüller, M. Wenzel, and L. C. Paulson, *Locales – A Sectioning Concept for Isabelle*, in: Y. Bertot et al. (eds.), *Theorem Proving in Higher Order Logics*, LNCS 1690, Springer, Berlin–Heidelberg, 1999, 149–165.

[83] Deepak Kapur, *Using Gröbner bases to reason about geometry problems*, J. Symb. Comput. **2(4)** (1986), 399–408.

[84] G. Klein, J. Andronick, K. Elphinstone, T. C. Murray, T. Sewell, R. Kolanski, and G. Heiser, *Comprehensive formal verification of an OS microkernel*, ACM Trans. Comput. Syst. **32(1)**, 2014, 2:1–2:70.

[85] 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*, ACM, 2009, 207–220.

[86] S. Krstić and A. Goel, *Architecting solvers for SAT modulo theories: Nelson-oppen with DPLL*; in: B. Konev and F. Wolter (eds.), *Frontiers of Combining Systems, FroCoS*, LNCS 4720, Springer, Berlin–Heidelberg, 2007, 1–27.

[87] P. Lammich and A. Lochbihler, *The Isabelle Collections Framework*, in: M. Kaufmann and L. Paulson (eds.), *Interactive Theorem Proving*, LNCS 6172, Springer, Berlin–Heidelberg, 2010, 339–354.

[88] P. Lammich, *Refinement for monadic programs*, Archive of Formal Proofs, 2012. `http://afp.sf.net/entries/Refine_Monadic.shtml`, Formal proof development.

[89] X. Leroy, *Formal verification of a realistic compiler*, Commun. ACM **52(7)** (2009), 107–115.

[90] X. Leroy, *A formally verified compiler back-end*, J. Autom. Reasoning **43(4)** (2009), 363–446.

[91] P. Letouzey, *Extraction in Coq: An Overview*, in: A. Beckmann, C. Dimitracopoulos, and B. Löwe (eds.), *Logic and Theory of Algorithms: Computability in Europe, CiE 2008*, LNCS 5028, Springer, Berlin–Heidelberg, 2008, 359–369.

[92] D. Mackenzie, *What in the name of Euclid is going on here?*, Science **307(5714)** (2005), 1402–1403.

[93] P. Maksimović, *Development and Verification of Probability Logics and Logical Frameworks*, PhD thesis, Université Nice Sophia Antipolis and University of Novi Sad, 2013.

[94] F. Marić, *Formalizacija, implementacija i primene SAT rešavača*, PhD thesis, Matematički fakultet, Univerzitet u Beogradu, 2009.

[95] F. Marić, *Formalization and implementation of modern SAT solvers*, J. Autom. Reasoning **43(1)**, 2009, 81–119.

[96] F. Marić, *Formal verification of a modern SAT solver by shallow embedding into Isabelle/HOL*, Theor. Comp. Sci. **411(50)** (2010), 4333–4356.

[97] F. Marić and P. Janičić, *Formal correctness proof for DPLL procedure*, Informatica, Vilnius **21(1)** (2010), 57–78.

[98] F. Marić and P. Janičić, *Formalization of abstract state transition systems for SAT*, Log. Methods Comput. Sci. **7(3)** (2011).

[99] F. Marić and D. Petrović, *Formalizing complex plane geometry*, Ann. Math. Artif. Intell. **74(3)** (2014), 271–308.

[100] F. Marić, M. Zivković, and B. Vučković, *Formalizing Frankl's conjecture: FC-families*, in: J. Jeuring et al. (eds.), *Calculemus 2012, Held as Part of Intelligent Computer Mathematics – CICM*, LNCS 7362, Springer, Berlin–Heidelberg, 2012, 248–263.

[101] P. Martin-Löf and G. Sambin, *Intuitionistic type theory*, Studies in proof theory. Bibliopolis, Berkeley, 1984.

[102] C. McBride, *Epigram: Practical programming with dependent types*, in: V. Vene and T. Uustalu (eds.), *Advanced Functional Programming*, LNCS 3622, Springer, Berlin–Heidelberg, 2004, 130–170.

[103] J. McCarthy, *Computer programs for checking mathematical proofs*, in: *Recursive Function Theory, Proceedings of a Symposium in Pure Mathematics V*, American Mathematical Society, New York, 1962, 219–227.

[104] J. Meng, C. Quigley, and L. C. Paulson, *Automation for interactive proof: First prototype*, Inf. Comput. **204(10)** (2006), 1575–1596.

[105] R. Milewski, *Fundamental theorem of algebra*, Formaliz. Math. **9(3)** (2001), 461–470.

[106] R. Milner, *A theory of type polymorphism in programming*, J. Comput. Syst. Sci. **17(3)** (1978), 348–375.

[107] R. Milner, *LCF: A way of doing proofs with a machine*, in: *Mathematical Foundations of Computer Science*, LNCS 74, Springer, Berlin–Heidelberg, 1979, 146–159.

[108] M. B. Nathanson, *Elementary methods in number theory*, Springer, Berlin–Heidelberg, 2000.

[109] A. Naumowicz and A. Kornilowicz, *A Brief Overview of Mizar*, in: S. Berghofer et al. (eds.), *Theorem Proving in Higher Order Logics*, LNCS 5674, Springer, Berlin–Heidelberg, 2009, 67–72.

[110] R.P. Nederpelt, J.H. Geuvers, and R.C. de Vrijer, *Selected Papers on Automath*, Studies in Logic and the Foundations of Mathematics, Elsevier Science, Amsterdam, 1994.

[111] D.J. Newman, *Analytic Number Theory*, volume 177 of Graduate Texts in Mathematics, Springer, Berlin–Heidelberg, 1998.

[112] R. Nieuwenhuis, A. Oliveras, and C. Tinelli, *Solving SAT and SAT Modulo Theories: From an abstract Davis–Putnam–Logemann–Loveland procedure to DPLL(*T*)*, J. ACM **53(6)** (2006), 937–977.

[113] T. Nipkow, G. Bauer, and P. Schultz, *Flyspeck I: Tame Graphs*, in: U. Furbach and N. Shankar (eds.), *Automated Reasoning – IJCAR 2006*, LNCS 4130, Springer, Berlin–Heidelberg, 2006, 21–35.

[114] T. Nipkow, L. C. Paulson, and M. Wenzel, *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, LCNS 2283, Springer, Berlin–Heidelberg, 2002.

[115] B. Nordström, K. Petersson, and J. M. Smith, *Martin-Löf's type theory, Handbook of logic in computer science: Volume 5*, Oxford University Press, Oxford, 2001.

[116] U. Norell, *Towards a practical programming language based on dependent type theory*, PhD thesis, Chalmers University of Technology, 2007.

[117] S. Obua and T. Nipkow, *Flyspeck II: The basic linear programs*, Ann. Math. Artif. Intell. **56** (2009), 245–272.

[118] S. Owre, J. M. Rushby, and N. Shankar, *PVS: A prototype verification system*, in: D. Kapur (ed.), *Automated Deduction – CADE-11*, LNCS 607, Springer, Berlin–Heidelberg, 1992, 748–752.

[119] L. C. Paulson, *Natural deduction as higher-order resolution*, J. Log. Program. **3** (1986), 237–258.

[120] L. C. Paulson, *Logic and Computation: Interactive Proof with Cambridge LCF*, Cambridge University Press, New York, 1987.

[121] L. C. Paulson, *The foundation of a generic theorem prover*, J. Autom. Reasoning **5(3)** (1989), 363–397.

[122] L. C. Paulson, *Isabelle: The next 700 theorem provers*, in: *Logic and Computer Science*, Academic Press, 1990, 361–386.

[123] T. Peterfalvi, S. P. G. N. J. Hitchin, and R. Sandling. *Character Theory for the Odd Order Theorem*, London Mathematical Society Lecture Notes. Cambridge University Press, 2013.

[124] Bjorn Poonen, *Union-closed families*, J. Comb. Theory, Ser. A **59(2)** (1992), 253 – 268.

[125] D. L. Rager, W. A. Hunt Jr, and M. Kaufmann, *A parallelized theorem prover for a logic with parallel execution*, in: S. Blazy, C. Paulin-Mohring, and D. Pichardie (eds.), *Interactive Theorem Proving*, LNCS 7998, Springer, Berlin–Heidelberg, 2013, 435–450.

[126] N. Robertson, D. Sanders, P. Seymour, and R. Thomas, *The four-colour theorem*, J. Comb. Theory, Ser. B **70(1)** (1997), 2–44.

[127] A. Robinson and A. Voronkov, eds., *Handbook of Automated Reasoning*, Elsevier, Amsterdam, 2001.

[128] H. Schwerdtfeger, *Geometry of Complex Numbers*, Dover Books on Mathematics, Dover, 1979.

[129] Dana S. Scott, *A Type-theoretical Alternative to ISWIM, CUCH, OWHY*, Theor. Comput. Sci. **121(1-2)** (1993), 411–440.

[130] N. Shankar, *Metamathematics, Machines and Gödel's Proof*, Cambridge Tracts in Theoretical Computer Science, Cambridge University Press, 1997.

[131] H. N. Shapiro, *Introduction to the theory of numbers*, Wiley, New York, 1983.

[132] A. Solovyev and T. C. Hales, *Formal Verification of Nonlinear Inequalities with Taylor Interval Approximations*, in: G. Brat, N. Rungta, and A. Venet (eds.), *NASA Formal Methods*, LNCS 7871, Springer, Berlin–Heidelberg, 2013, 383–397.

[133] Alexey Solovyev, *Formal Computations and Methods*, PhD thesis, University of Pittsburgh, 2012.

[134] M. Spasić and F. Marić, *Formalization of Incremental Simplex Algorithm by Stepwise Refinement*, in: D. Giannakopoulou and D. Méry (eds.), *FM 2012: Formal Methods*, Springer, Berlin–Heidelberg, 2012, 434–449.

[135] S. Stojanović, J. Narboux, M. Bezem, and P. Janičić, *A Vernacular for Coherent Logic*, in: M. Watt et al. (eds.), *Intelligent Computer Mathematics – CICM*, LNCS 8543, Springer, Berlin–Heidelberg, 2014, 388-403.

[136] S. Stojanović, V. Pavlović, and P. Janičić, *A coherent logic based geometry theorem prover capable of producing formal and readable proofs*, in: P. Schrek, J. Narboux, and J. Richter-Gebert (eds.), *Automated Deduction in Geometry*, LNCS 6877, Springer, Berlin–Heidelberg, 2011, 201–220.

[137] R. Taylor and A. Wiles, *Ring-Theoretic Properties of Certain Hecke Algebras*, Annals of Math. **141(3)** (1995), 553–572.

[138] A. Trybulec and H. Blair, *Computer assisted reasoning with MIZAR*, in: *International Joint Conference on Artificial Intelligence – IJCAI'85*, Morgan Kaufmann, San Francisco, 1985, 26–28.

[139] The Univalent Foundations Program, *Homotopy Type Theory: Univalent Foundations of Mathematics*, Institute for Advanced Study, 2013. `http://homotopytypetheory.org/book`,

[140] T. Weber, *Integrating a SAT solver with an LCF-style theorem prover*, Electr. Notes Theor. Comput. Sci. **144(2)** (2006), 67–78.

[141] M. Wenzel, *Isabelle/Isar – a versatile environment for human-readable formal proof documents*, PhD thesis, Technische Universitt München, 2002.

[142] M. Wenzel, *Asynchronous proof processing with Isabelle/Scala and Isabelle/JEdit*, Electron. Notes Theor. Comput. Sci. **285** (2012), 101–114.

[143] M. Wenzel, *Shared-memory multiprocessing for interactive theorem proving*, in: S. Blazy, C. Paulin-Mohring, and D. Pichardie (eds.), *Interactive Theorem Proving*, LNCS 7998, Springer, Berlin–Heidelberg, 2013, 418–434.

[144] M. Wenzel, *Parallel proof checking in Isabelle/Isar*, in: *ACM SIGSAM Workshop on Programming Languages for Mechanized Mathematics Systems (PLMMS 2009)*, ACM Digital Library, 2009.

[145] M. Wenzel and F. Wiedijk, *A comparison of Mizar and Isar*, J. Autom. Reasoning **29(3-4)** (2003), 389–411.

[146] F. Wiedijk, *Mizar Light for HOL Light*, in: R. J. Boulton and P. B. Jackson (eds.), *Theorem Proving in Higher Order Logics*, LNCS 2152, Springer, Berlin–Heidelberg, 2001, 378–393.

[147] F. Wiedijk, *The Seventeen Provers of the World: Foreword by Dana S. Scott (LNCS/LNAI)*, Springer, Berlin–Heidelberg, 2006.

[148] F. Wiedijk, *Formal proof — getting started*, Notices Am. Math. Soc. **55(11)** (2008), 1408–1414.

[149] F. Wiedijk, *A synthesis of the procedural and declarative styles of interactive theorem proving*, Log. Methods Comput. Sci. **8** (2012), 1–26.

[150] A. J. Wiles, *Modular elliptic curves and Fermat's Last Theorem*, Annals of Math. **141(3)** (1995), 443–551.

[151] R. Wilson, *Four Colours Suffice*, Princeton Sciece Library, Princeton University Press, 2002.

[152] W. Wu, *Mechanical Theorem Proving in Geometries: Basic Principles*, Symbolic computation, Springer, Berlin–Heidelberg, 1994.

[153] R. Zumkeller, *Global Optimization in Type Theory*, PhD thesis, Ecolé Polytechnique, Paris, 2008.