LISA – A Modern Proof System

Simon Guilloud
Laboratory for Automated Reasoning and Analysis, EPFL, Lausanne, Switzerland

Sankalp Gambhir
Laboratory for Automated Reasoning and Analysis, EPFL, Lausanne, Switzerland

Viktor Kunčak
Laboratory for Automated Reasoning and Analysis, EPFL, Lausanne, Switzerland

Abstract

We present LISA, a proof system and proof assistant for constructing proofs in schematic first-order logic and axiomatic set theory. The logical kernel of the system is a proof checker for first-order logic with equality and schematic predicate and function symbols. It implements polynomial-time proof checking and uses the axioms of ortholattices (which implies the irrelevance of the order of conjuncts and disjuncts and additional propositional laws). The kernel supports the notion of theorems (whose proofs are not expanded), as well as definitions of predicate symbols and objects whose unique existence is proven. A domain-specific language enables construction of proofs and development of proof tactics with user-friendly tools and presentation, while remaining within the general-purpose language, Scala. We describe the LISA proof system and illustrate the flavour and the level of abstraction of proofs written in LISA. This includes a proof-generating tactic for propositional tautologies, leveraging the ortholattice properties to reduce the size of proofs. We also present early formalization of set theory in LISA, including Cantor's theorem.

2012 ACM Subject Classification Theory of computation → Logic and verification

Keywords and phrases Proof assistant, First Order Logic, Set Theory

Digital Object Identifier 10.4230/LIPIcs.ITP.2023.17

Supplementary Material *Software (Source Code)*: `https://github.com/epfl-lara/lisa`
archived at `swh:1:dir:614ac272bee1c2bf21308ad532c3ca3dd3ec3832`

1 Introduction

We present the design and initial implementation of a new proof assistant, named LISA. Much like Mizar [31], LISA aims to use classical mainstream foundations of mathematics with first order logic and set theory. LISA uses (single-sorted) first-order logic (with schematic variables) as the syntactic framework, sequent calculus as the deduction framework and set theory as the semantic framework. On top of this foundation, we can construct mathematical theories without introducing additional axioms. As the target use of LISA we envision a library of theorems, but also correctness proofs of computer systems.

LISA's source code and a reference manual, as well as all the examples in the present paper, are available from

`https://github.com/epfl-lara/lisa`

1.1 Design Goals

Our design is inspired by the LCF line of proof assistants, including HOL Light, HOL4, and Isabelle. The envisioned path for axiomatic foundations is closer to Mizar. LISA's logical kernel is a hybrid between LCF-style encoding of theorems as a sealed Theorem type (similar to HOL Light [22]) and explicit requirement of proofs. Namely, proofs are self-contained

sequences of proof steps that derive a conclusion from assumptions (they are not explicitly in the form of lambda terms). LISA's kernel checks the validity of steps and assumptions, and then creates an instance of a theorem.

As the unified *implementation, proof writing and tactic language*, we use Scala instead of the ML family of languages that are common to many proof assistants. Scala is a high-level functional and object-oriented language. We hope to avoid a sharp boundary between user proofs and tactic developments by using a single language with good support for domain-specific constructs. To provide a flavour of LISA, consider several ways of constructing proofs that are available to LISA users. Figure 2 shows a proof of Pierce's law as an explicit sequence of sequent calculus proof steps. Figure 5 and Figure 6 show proofs built using a higher-level domain-specific language (DSL). This DSL detects high-level errors in incorrect proofs, but always generates the underlying lower-level proof and forwards it to the kernel to obtain a kernel-certified theorem. Finally, Figure 9 shows a solver for propositional tautologies that uses the same mechanisms as the proofs to implement a proof tactic. It was not our immediate goal to create an interactive experience, so our interaction model is more HOL4-like than Isabelle/HOL-like. For us, this means using Scala IDEs, rerunning projects, relying on incremental compilation. As the sizes of theories grow, we plan to develop serializations for proofs and theories to reduce re-execution. We discuss both the kernel and the DSL in the rest of the paper.

The design philosophy of LISA focuses on what one might call the *Six Virtues of Modern Proof Systems*. *Efficiency* say a proof system components should have polynomial complexity (as close as possible to linear). *Trust* means high confidence in the system, through a combination of well understood mathematical foundations, explicit proofs and a concise logical kernel. *Usability* is making it simple, both for human users and automated methods, to formalize mathematics and to develop tools. *Predictability* is the property of systems whose behaviour and output have clear characterizations. *Interoperability*, whose importance has become clear over the years, consists in making it as easy as possible for the system to be used by other systems and to export and import proofs to and from other systems. Finally, *Programmability* implies that as a computer system, a proof assistant should provide all the expressiveness allowed by a programming language. When designing and developing the LISA proof system, we aim to respect the six virtues as much as possible, and, when they oppose each other, to strike for the best balance between them.

1.2 Contributions

The contribution of this paper is to present the design of LISA, a new proof construction system embedded in Scala, based on schematic first-order logic with set theory axioms. We focus on the following aspects.

- We describe how the logical kernel is constructed and how it can be used or interacted with by other tools.
- As the most unusual design aspect, we describe ortholattice-based algorithms implemented in the kernel to make proofs shorter.
- We present a domain-specific language embedded in Scala that makes the writing of proofs easier and generates and checks kernel proofs to obtain kernel-certified theorems.
- We show that the same domain-specific language can scale from writing proofs of specific theorems to writing general tactics. As an example of a tactic, we present a (proof generating) solver for propositional formulas leveraging the ortholattice algorithm.
- We report on the initial steps of developing elementary axiomatic set theory in the system.

2 Logical Kernel

LISA's deductive system is a variant of Gentzen's Sequent Calculus for first-order logic (FOL) [15]. Formally, a sequent in LISA is a pair of sets of formulas Γ and Δ, represented $\Gamma \vdash \Delta$ and its interpretation is $\bigwedge \Gamma \to \bigvee \Delta$. LISA extends the prototypical Sequent Calculus with schematic symbols, substitution rules, and a normalization of formulas.

2.1 Schematic Symbols

Formulas in LISA's kernel are built with the usual variables, constant function and predicate symbols, logical connectors and binders, but also admit the use of schematic function, predicate and connector symbols. These symbols behave like uninterpreted constant symbols which can be substituted by any well-typed term or formula across a whole sequent, or like variables which cannot be bound. We refer to them as second-order schematic symbols, as opposed to regular variables, which are first-order schematic symbols. This gives the system a flavour of second order logic and allows writing axiom and theorem schemas, as the following example illustrates:

▶ **Example 1.** The following sequent (whose proof we show in Figure 5) is provable without additional assumptions on the function symbol f and the predicate symbol P:

$$\forall x.P(x) \to P(f(x)) \vdash \forall x.P(x) \to P(f(f(x)))$$

This means that this sequent with f replaced by a specific term (with a distinguished free variable) remains provable by an analogous proof, and similarly for P replaced by a formula.

In traditional first-order logic, this concept is formalized as a meta-theorem stating that for every f and P, a corresponding proof can be built. However, in a formal setting, this requires duplicating the whole proof for every specific f and P. Instantiation of schematic symbols avoids this issue of proof duplication. This is similar to the schematic variables found in Isabelle [33], and in particular Isabelle/FOL [35], where variables from the meta-logic can be used to represent arbitrary functions, predicates, and connectors in FOL. Crucially, it does not increase the expressive power of the system, because it can, in principle, be simulated.

2.2 Ortholattice Algorithm Applied to First-Order Logic

We find that using a proof system that is sensitive to the order of conjuncts and similar semantically irrelevant syntactic differences can be frustrating and increases proof size unnecessarily. To address this issue, LISA's kernel strengthens sequent calculus with a built-in algorithm to compute normal form and equivalence of formulas with respect to a subset of equational rules of propositional logic. These rules, shown in Table 1, characterize the algebraic theory of ortholattices (abbreviated OL) [3, Chapter II.1], [7].

Ortholattices are a generalization of Boolean algebra where instead of the law of distributivity, the weaker absorption law (L9, Table 1) holds. In particular, every identity in the theory of ortholattices is also a theorem of propositional logic.

This algebraic structure has been shown to possess a quadratic-time normalization algorithm [18] and has been suggested as the basis for normalization of formulas in the context of verification and mechanized proofs. Notably, it subsumes negation normal form.

As a special kind of lattices, ortholattices can be viewed as partially ordered sets, with the ordering relation on two elements a and b of an ortholattice defined as $a \leq b \iff a \wedge b = a$, which, by absorption (L9), is also equivalent to $a \vee b = b$. If s and t are terms over the

Table 1 Laws of ortholattices, an algebraic theory with signature $(S, \wedge, \vee, 0, 1, \neg)$. [18]

L1:	$x \vee y = y \vee x$	L1':	$x \wedge y = y \wedge x$
L2:	$x \vee (y \vee z) = (x \vee y) \vee z$	L2':	$x \wedge (y \wedge z) = (x \wedge y) \wedge z$
L3:	$x \vee x = x$	L3':	$x \wedge x = x$
L4:	$x \vee 1 = 1$	L4':	$x \wedge 0 = 0$
L5:	$x \vee 0 = x$	L5':	$x \wedge 1 = x$
L6:	$\neg\neg x = x$	L6':	same as L6
L7:	$x \vee \neg x = 1$	L7':	$x \wedge \neg x = 0$
L8:	$\neg(x \vee y) = \neg x \wedge \neg y$	L8':	$\neg(x \wedge y) = \neg x \vee \neg y$
L9:	$x \vee (x \wedge y) = x$	L9':	$x \wedge (x \vee y) = x$

signature $(S, \wedge, \vee, 0, 1, \neg)$, we denote $s \leq_{OL} t$ if and only if $OL \vDash s \leq t$, i.e., it holds in all ortholattices. We write $s \sim_{OL} t$ if both $s \leq_{OL} t$ and $s \geq_{OL} t$ hold (or equivalently, if $OL \vDash s = t$). Theorem 1 is the main result we rely on.

▶ **Theorem 1** ([18]). *There exists an algorithm running in worst case quadratic time producing, for any terms s over the signature $(\wedge, \vee, \neg)$, a normal form $NF_{OL}(s)$ such that for any t, $s \sim_{OL} t$ if and only if $NF_{OL}(s) = NF_{OL}(t)$. The algorithm is also capable of deciding if $s \leq_{OL} t$ holds in quadratic time.*

Moreover, the algorithm works with structure sharing with the same complexity, which is very relevant for example when $x \leftrightarrow y$ is expanded to $(x \wedge y) \vee (\neg x \wedge \neg y)$. It can produce a normal form in this case as well.

These properties, along with completeness characterization, make the OL algorithm a good candidate to include in a proof system. LISA's kernel further extends OL inequality algorithm to first order logic formulas as follows. It first expresses the formula using de Bruijn indices [11], then desugars $\exists.\phi$ into $\neg\forall.\neg\phi$. It then extends the OL algorithm with the rules in Table 2.

Table 2 Extension of OL algorithm to first-order logic. We call it the F(OL)^2 algorithm. $=$ denotes the equality predicate in FOL, while $==$ denotes syntactic equality of terms.

	To decide...	Reduce to...
1	$\{\wedge, \vee, \rightarrow, \leftrightarrow, \neg\}(\vec{\phi}) \leq \psi$	Base algorithm
2	$\phi \leq \{\wedge, \vee, \rightarrow, \leftrightarrow, \neg\}(\vec{\psi})$	Base algorithm
3	$s_1 = s_2 \leq t_1 = t_2$	$\{s_1, s_2\} == \{t_1, t_2\}$
4	$\phi \leq t_1 = t_2$	$t_1 == t_2$
5	$\forall.\phi \leq \forall.\psi$	$\phi \leq \psi$
6	$'C(\phi_1, ..., \phi_n) \leq 'C(\psi_1, ..., \psi_n)$	$\phi_i \sim_{OL} \psi_i$, for every $1 \leq i \leq n$
7	Anything else	`false`

When either of the two formulas being compared have a top-level propositional operator (cases 1 and 2), the recursion is done according to the algorithm described in [18], considering any non-propositional expressions (predicates, quantified formulas, and schematic connectors) as propositional variables. The third and fourth rules take into account reflexivity and symmetry of equality. The fifth relies on monotonicity of $\forall$, and the sixth rule applies when $'C$ is a schematic connector, i.e., a logical connector about which we know nothing. These rules extend to the normal-form-producing algorithm, and it is easy to see that if $\leq$ is interpreted as logical implication, they are sound. We decided not to include a rule

such as $\forall.\phi \leq \phi(t)$. The reason is that incorporating such a rule systematically runs risk of introducing higher complexity [27] in the kernel. We instead decided that such steps should be implemented using tactics, outside the kernel in the future (possibly making use of type-like hints encoded in first-order logic [14]).

Using the First Order Logic OrthoLattices algorithm, noted $\mathrm{F(OL)}^2$, the proof checker in LISA's kernel performs every correctness check up to $\mathrm{F(OL)}^2$ equivalence. This does not prevent sequents and formulas from having arbitrary constructions and being inspected in a stable, predictable way by tactics, as formulas are not normalized in-place. The set of LISA deduction rules is shown in Figure 1.

Moreover, the proof checker contains a special `Restate` proof step, which permits $\mathrm{F(OL)}^2$-transformations on the entire sequent, leveraging the interpretation of a sequent as a formula (an implication). We also leverage specifically the partial order computed by $\mathrm{F(OL)}^2$ to expand the usual `Weakening` rule so that the premise sequent only has to be $\leq_{F(OL)^2}$ stronger than the conclusion, with both interpreted as formulas. `Weakening` clearly subsumes `Restate`, but the latter ensures that the transformation is actually an equivalence and hence could be reversed, which can be a useful safeguard in practice. These rules subsume most propositional rules in Figure 1.

2.3 Substitution Rules

The substitution rules substitute equal terms or equivalent formulas inside a formula. They are deduced steps whose simulation from simpler steps can take a number of steps linear in the size of the sequent, yet are very frequent both in human-written proofs and automated reasoning (as done by SAT solvers or in systems with rewrite rules, for example), justifying their inclusion as base steps. A special case of substitution that is particularly important is the following:

$$\frac{\phi \vdash \psi}{\phi \vdash \psi[\phi := \top]}\ \text{SubstIff}$$

This holds in a single step because $\phi \leftrightarrow \top \sim_{F(OL)^2} \phi$. In fact, `Restate` and `SubstIff` form a complete basis for propositional logic that we will leverage in Subsection 4.1 to write a complete proof-producing tactic for propositional logic.

The inclusion of $\mathrm{F(OL)}^2$ and the substitution and instantiation deduced rules in the logical kernel is a slight bend to the trust principle, but as the algorithm is only 300 lines of code, this is largely overshadowed by the increased usability and shorter proofs. In fact, the whole kernel adds up to a grand total of only 1607 lines of code. This comprises the implementation of first-order logic, the $\mathrm{F(OL)}^2$ algorithm, first and second-order substitution, the sequent calculus steps, the proof checker, and a manager for definitions and theorems (detailed in Subsection 2.5). Moreover, LISA's kernel is efficient: except for the quadratic $\mathrm{F(OL)}^2$ algorithm, every procedure in the kernel is linear (up to logarithmic coefficients) in the size of the formulas or proofs being considered.

2.4 Proof Objects

In LISA, a proof is an explicit list of proof steps, where each step can refer to previous steps via their respective position in the list and be referred by multiple subsequent steps. In other words, a proof is represented as a topological linearization of the proof tree, or, more generally, a directed acyclic graph (permitting reuse of intermediate steps). A proof step also contains the arguments that allow the proof checker to efficiently verify it. In particular, LISA's kernel does not rely on a unification algorithm to check correctness of proof steps related to quantifiers.

$$\frac{}{\Gamma, \phi \vdash \phi, \Delta}\ \text{Hypothesis}$$

$$\frac{\Gamma \vdash \phi, \Delta \qquad \Sigma, \phi \vdash \Pi}{\Gamma, \Sigma \vdash \Delta, \Pi}\ \text{Cut}$$

$$\frac{\Gamma, \phi, \psi \vdash \Delta}{\Gamma, \phi \wedge \psi \vdash \Delta}\ \text{LeftAnd} \qquad \frac{\Gamma \vdash \phi, \Delta \qquad \Sigma \vdash \psi, \Pi}{\Gamma, \Sigma \vdash \phi \wedge \psi, \Delta, \Pi}\ \text{RightAnd}$$

$$\frac{\Gamma, \phi \vdash \Delta \qquad \Sigma, \psi \vdash \Pi}{\Gamma, \Sigma, \phi \vee \psi \vdash \Delta, \Pi}\ \text{LeftOr} \qquad \frac{\Gamma \vdash \phi, \psi\Delta}{\Gamma \vdash \phi \vee \psi, \Delta}\ \text{RightOr}$$

$$\frac{\Gamma \vdash \phi, \Delta \qquad \Sigma, \psi \vdash \Pi}{\Gamma, \Sigma, \phi \to \psi \vdash \Delta, \Pi}\ \text{LeftImplies} \qquad \frac{\Gamma, \phi \vdash \psi, \Delta}{\Gamma \vdash \phi \to \psi, \Delta}\ \text{RightImplies}$$

$$\frac{\Gamma, \phi \to \psi \vdash \Delta}{\Gamma, \phi \leftrightarrow \psi \vdash \Delta}\ \text{LeftIff} \qquad \frac{\Gamma \vdash \phi \to \psi, \Delta \qquad \Sigma \vdash \psi \to \phi, \Pi}{\Gamma, \Sigma \vdash \phi \leftrightarrow \psi, \Delta, \Pi}\ \text{RightIff}$$

$$\frac{\Gamma \vdash \phi, \Delta}{\Gamma, \neg\phi \vdash \Delta}\ \text{LeftNot} \qquad \frac{\Gamma, \phi \vdash \Delta}{\Gamma \vdash \neg\phi, \Delta}\ \text{RightNot}$$

$$\frac{\Gamma, \phi[t := 'x] \vdash \Delta}{\Gamma, \forall 'x.\phi \vdash \Delta}\ \text{LeftForall} \qquad \frac{\Gamma \vdash \phi, \Delta}{\Gamma \vdash \forall 'x.\phi, \Delta}\ \text{RightForall}$$

$$\frac{\Gamma, \phi \vdash \Delta}{\Gamma, \exists 'x.\phi \vdash \Delta}\ \text{LeftExists} \qquad \frac{\Gamma \vdash \phi[t := 'x], \Delta}{\Gamma \vdash \exists 'x.\phi, \Delta}\ \text{RightExists}$$

$$\frac{\Gamma \vdash \Delta}{\Gamma[\psi(\vec{v}) := 'p(\vec{v})] \vdash \Delta[\psi(\vec{v}) := 'p(\vec{v})]}\ \text{InstSchema}$$

$$\frac{\Gamma, \phi[s := 'f] \vdash \Delta}{\Gamma, s = t, \phi[t := 'f] \vdash \Delta}\ \text{LeftSubstEq} \qquad \frac{\Gamma \vdash \phi[s := 'f], \Delta}{\Gamma, s = t \vdash \phi[t := 'f], \Delta}\ \text{RightSubstEq}$$

$$\frac{\Gamma, \phi[a := 'p] \vdash \Delta}{\Gamma, a \leftrightarrow b, \phi[b := 'p] \vdash \Delta}\ \text{LeftSubstIff} \qquad \frac{\Gamma \vdash \phi[a := 'p], \Delta}{\Gamma, a \leftrightarrow b \vdash \phi[b := 'p], \Delta}\ \text{RightSubstIff}$$

$$\frac{\Gamma, t = t \vdash \Delta}{\Gamma \vdash \Delta}\ \text{LeftRefl} \qquad \frac{}{\vdash t = t}\ \text{RightRefl}$$

$$\frac{\Gamma_1 \vdash \Delta_1}{\Gamma_2 \vdash \Delta_2}\ \text{Restate}\ \ \text{if}\ \left(\bigwedge \Gamma_1 \to \bigvee \Delta_1\right) \sim_{F(OL)^2} \left(\bigwedge \Gamma_2 \to \bigvee \Delta_2\right)$$

$$\frac{\Gamma_1 \vdash \Delta_1}{\Gamma_2 \vdash \Delta_2}\ \text{Weakening}\ \ \text{if}\ \left(\bigwedge \Gamma_1 \to \bigvee \Delta_1\right) \leq_{F(OL)^2} \left(\bigwedge \Gamma_2 \to \bigvee \Delta_2\right)$$

Figure 1 Deduction rules allowed by LISA's kernel. Different occurrences of the same symbols need not represent equal elements, but only elements with the same $F(OL)^2$ normal form.

Moreover, proofs are standalone objects checkable and exportable without the need for any kind of context. Figure 2 shows an example of sequent calculus proof as a sequence of steps. Each step lists a sequent with a rule from Figure 1 and a list of (the position of) previous steps from which the sequent follows. Figure 3 shows executable Scala code that denotes the same proof, which can be given directly to the LISA kernel. The kernel can efficiently check its correctness and create a theorem whose statement corresponds to the last sequent, corresponding to the root of the proof tree. Note that, in this particular case, the same conclusion could be reached in a single step using the `Restate` rule.

If the proof relies on external theorems, axioms or definitions, those are stated after the list of proof steps and referred to with negative positions. We call those *imported* sequents (*imports*, for short). We adopt an analogous mechanism to support *subproofs*. A subproof simulates deduced steps by encapsulating an inner proof and appears as a single step in the outer proof. In that case, the premises of the subproof become imports of the inner proof.

0	`Hypothesis`	$\phi \vdash \phi$
1	`Weakening`(0)	$\phi \vdash \phi, \psi$
2	`RightImplies`(1)	$\vdash \phi, (\phi \to \psi)$
3	`LeftImplies`$(2, 0)$	$(\phi \to \psi) \to \phi \vdash \phi$
4	`RightImplies`(3)	$\vdash ((\phi \to \psi) \to \phi) \to \phi$

Figure 2 The proof of Pierce's Law as a sequence of steps using classical Sequent Calculus rules.

```
1 val PierceLawProof = SCProof(IndexedSeq(
2     Hypothesis(               φ ⊢ φ,                                  φ),
3     Weakening(                φ ⊢ (φ, ψ),                             0),
4     RightImplies(            () ⊢ (φ, φ ⟹ ψ),                  1, φ, ψ),
5     LeftImplies( (φ ⟹ ψ) ⟹ φ ⊢ φ,                  2, 0, (φ ⟹ ψ), φ),
6     RightImplies(            () ⊢ ((φ ⟹ ψ) ⟹ φ) ⟹ φ,
7                                                 3, (φ ⟹ ψ) ⟹ φ, φ)
8 ), Seq.empty /* no imports */ )
```

Figure 3 The proof from Figure 2 written for LISA's kernel. $\vdash$ and $\Longrightarrow$ are alternative, nicer constructors for sequents and formulas and are not part of the kernel. The second argument (here empty) is the sequence of proof imports.

2.5 Theories

LISA's proof checker can be used as a tool to produce and check proofs, independent of any context, but is not a sufficient tool to develop mathematical theories, as it lacks in particular the ability to make definitions. For this task, the kernel also offers a minimal utility to allow development of mathematical theories with the ability to introduce axioms, theorems, and definitions with guaranteed soundness.

Theorems

This part of the kernel, called the `Theory`, is inspired from the LCF style [16]. It allows checking a proof once, producing a value of a sealed type `Theorem`, which can then be reused many times. The proof can then be forgotten. The `Theory` will also verify that the given

proof's imports are properly justified by existing axioms, theorems or definitions, so that the proven `Theorem` can be considered unconditionally true, unlike its standalone proof. Figure 4 shows how to use the `Theory` to obtain a theorem.

```
1 val theory = new Theory
2 val pierceThm: theory.Theorem = theory.makeTheorem(
3    "Pierce's Law",
4    () ⊢ ((ϕ ⟹ ψ) ⟹ ϕ) ⟹ ϕ,
5    PierceLawProof,
6    Seq.empty
7 )
```

Figure 4 The proof from Figure 3 can be transformed into a `Theorem` by a `Theory`. The arguments are, in order, the name of the theorem, its statement, a proof of the statement and the list of previous theorems, axioms or definitions used to justify the proof's imports, if any.

The `Theory` naturally corresponds to the concept of a "mathematical theory" in first-order logic, containing the language and axioms of said theory. To allow coexistence of multiple different theories with different valid theorems, LISA makes the `Theory` a class that can be instantiated multiple times. The `Theorem` type is dependent on a specific instance of `Theory`, so that two different theories will reject the theorem of the other. In a language without dependent types, this could be replaced by a simple runtime check. Note that in proof development, it is expected that the user will never need to use more than one theory at once, so this aspect is abstracted by the DSL.

Definitions

The theory also allows introducing new definitions for predicate and function symbols.

A *predicate symbol* P definition is of the form $P(x_1, \ldots, x_n) := \phi_{x_1,\ldots,x_n}$, where the $x_1, \ldots, x_n$ are the free variables of a given formula ϕ. To define a *function symbol* f, the definition requires a proof of unique existence of the form:

$$\exists! y.\ \phi_{y,x_1,\ldots,x_n} \tag{1}$$

and introduces a definitional axiom $\phi_{f(x_1,\ldots,x_n),x_1,\ldots,x_n}$, where again $x_1, \ldots, x_n$ are the free variables of the formula ϕ. To make such a definition, the `Theory` checks that the symbol has not already been defined and requires a proof of (1), i.e., of existence and uniqueness.

Remark on unique existence. One may hope that only the existence (but not uniqueness) was needed to obtain conservative extensions in first-order logic. Unfortunately, this is not true in the presence of axiom schemas. In particular, with such a definition principle, it becomes possible to prove the Axiom of Choice in ZF set theory, while they are well known to be independent [8]. Indeed, in ZF it is possible to prove

$$\forall x. \exists y.\ (x \neq \emptyset \implies y \in x)$$

from which we would obtain a function pick with the property

$$\forall x.\ (x \neq \emptyset \implies \text{pick}(x) \in x)$$

If then the symbol pick is allowed in axiom schemas, as would be the case in LISA, it is then easy to use pick and the replacement schema to construct a choice function on any set (see also LISA's Reference Manual [17]).

Abstraction via underspecified definitions. We have seen that we need uniqueness to ensure conservative extensions. On the other hand, such requirement often forces the defining formula to be overly specific and representation-dependant. For example, one may want to define the set of real numbers, $\mathbb{R}$, as a structure that satisfies the axioms of real closed fields. Since there are many isomorphic structures satisfying these, a uniqueness proof cannot be obtained. It is then necessary to use a specific construction, such as Cauchy sequences, as the definition of the set of real numbers. This, however, means that it becomes possible to prove properties of real numbers which are specific to the chosen representation, which is undesirable and especially so when transferring proofs to other proof systems, which may have different representations of reals. Our solution is to allow *underspecified* definitions. An underspecified definition still requires existence and uniqueness (ensuring a conservative extension), but the theorem that the kernel provides is only the desired, weaker, one. This mechanism makes use of the $\leq$ relation of $\mathrm{F}(\mathrm{OL})^2$. Section 3 shows an example of the use of underspecified definitions in set theory.

This issue is addressed in Metamath [29] by assuming a specific construction of the structure to conditionally prove a desired defining property (for example, the axioms of the real field) and then introducing said property independently as an axiom. This mechanism however is not enforced by Metamath itself but only an informal practice. LISA's kernel support for underspecified definitions ensures that the same goal is achieved with guaranteed soundness. Underspecification was also discussed in the context of HOL by Rob Arthan [1]. Our approach is similar but the challenges are different. The use of Hilbert's description operator leads to undesired properties being provable, similarly as definition via unique existence, but for the reason explained above, Lisa can't relax the condition to existence only due to the presence of axiom schemas. Forgetting a part of the definition after it was made was tried in HOL Light, but this made reasoning about the system harder, as it has to take into account not only the state of the system but also the sequence of operation leading to this state. LISA avoid this issue by making underspecified definitions an integral part of the foundation.

3 DSL for LISA in Scala

While the minimality of the kernel makes it tedious to use directly, the tools offered by Scala (and especially Scala 3) allow us to design a more intuitive DSL, similar to other proof assistants, directly within the host language. Moreover, essentially all the verification related to the syntactic construction and writing of the proof are checked at compilation time, leaving only the wrong use of proof steps and tactics (such as when trying to prove an invalid statement) as possible failure at runtime. LISA's interface encapsulates the kernel and provides convenient tools and syntax to make mathematical development easier to write and read. Figure 5 shows a minimal example of how to use the DSL to write a proof. This approach makes LISA programmable. It offers the user the full range of tools of the host language when writing proofs, allowing them to express proofs in novel ways or adapted to different areas of mathematics, similar to writing on paper.

LISA's environment is activated simply by creating an object extending `lisa.Main`. This will make available all the essential features to develop mathematics in LISA. Declarations in lines 2, 3 and 4 define a variable, a schematic predicate of arity one, and a schematic function of arity one, such that their symbols are the same as their Scala name, i.e., respectively `"x"`, `"P"` and `"f"`. This is made possible by implicit arguments and reflection. Line 6 starts the declaration of a theorem. (Note that the kernel itself does not rely on such specific features; we expect the kernel to be straightforward to implement in most languages.)

```
1  object Exercise extends lisa.Main {
2     val x = variable
3     val P = predicate(1)
4     val f = function(1)
5
6     val fixedPointDoubleApplication = Theorem(
7           ∀(x, P(x) ⟹ P(f(x))) ⊢ P(x) ⟹ P(f(f(x)))
8        ) {
9        assume(∀(x, P(x) ⟹ P(f(x))))
10       val step1 = have(P(x) ⟹ P(f(x))) by InstantiateForall
11       val step2 = have(P(f(x)) ⟹ P(f(f(x)))) by InstantiateForall
12       have(thesis) by Tautology.from(step1, step2)
13    }
14 }
```

Figure 5 A small proof written with LISA's DSL. Unicode characters are obtained in practice through ligatures or Scala's direct support for unicode.

3.1 Higher-Level Proofs

LISA's interface defines a proof constructing class. This class uses proof tactics to generate pieces of the final pure sequent calculus proof, which are encapsulated into kernel subproofs. The result from the point of view of the user is the ability to define arbitrarily computed deduced proof steps (here `Tautology` and `InstantiateForall`) from the base steps of sequent calculus. Thanks to Scala 3's implicit functions types [32], the proof constructor is automatically created in the code block following the `Theorem` declaration (line 6 of Figure 5) without the need for the user to even realize it exists. The existence of an implicit proof constructor in scope is necessary for the other keywords (`have`, `assume`, ...) to be well-defined, meaning that using those outside of a theorem environment will fail to compile.

The `assume` keyword (line 9) allows stating a formula that will be assumed true for the rest of the proof. Technically, it will be considered as part of the left-hand-side of any further written sequent in the proof. `have` states a proposition that can be reached using a proof tactic (or a subproof, see next example). If a step requires some premises, they can be given as parameters to the tactic, as in line 12. `have` produces a `Fact` that can be used by later steps.

The example in Figure 6 illustrates a more advanced proof structure, using axioms from set theory. As the proof independently proves both directions of the double implication, it makes use of the `subproof` construction. Similarly to the `Theorem` keyword, this construction implicitly creates a new proof constructor environment, internal to the outer proof and with its own goal. In a proof, a `Fact` is a type that contains external theorems, axioms and definitions, as well as previously proven steps from the current or outer proof, but not from any proof that is not a direct ancestor of the current proof. This is made possible by using recursively defined path-dependent types (see Figure 7) and can be checked at compile-time.

Moreover, a fact can also be one of the above, accompanied by information about a specific instantiation of schematic symbols. The actual instantiation step is then carried automatically. This is done in practice with the `of` keyword, as in line 11.

When a tactic requires a single premise, and this premise is the most recently proven fact, `thenHave` passes said premise directly to the tactic without the step having to be named. For some tactics, such as the `Substitution` step at line 21, the resulting sequent will be inferred by the tactic and isn't required to be given by the user. In this case, `have` and `thenHave`

```
1  val unionOfSingleton = Theorem( (union(singleton(x)) ≡ x) ) {
2     val X = singleton(x)
3     val forward = have( (in(z, x) ⟹ in(z, union(X))) ) subproof {
4        ...
5     }
6     val backward = have( in(z, union(X)) ⟹ in(z, x) ) subproof {
7        have(in(z, y) ⊢ in(z, y)) by Restate
8        val step2 = thenHave((y≡x, in(z, y)) ⊢ in(z, x))
9           by Substitution
10       have(in(z, y) ∧ in(y, X) ⊢ in(z, x))
11          by Tautology.from(pairAxiom of (y→x, z→y), step2)
12       val step4 = thenHave(∃(y, in(z, y) ∧ in(y, X)) ⊢ in(z, x))
13          by LeftExists
14       have( in(z, union(X)) ⟹ in(z, x))
15          by Tautology.from(unionAxiom of (x → X), step4)
16    }
17    have( in(z, union(X)) ⟺ in(z, x))
18       by RightIff(forward, backward)
19    thenHave( forall(z, in(z, union(X)) ⟺ in(z, x)))
20       by RightForall
21    andThen(Substitution(extensionalityAxiom of (x → union(X), y → x)))
22 }
```

Figure 6 A LISA proof with more advanced construction.

takes the tactic as argument. The `Tautology` step proves statements using propositional laws and the `Substitution` makes substitution of equals for equals, either everywhere or using unification to find the specific occurrences to replace.

```
class Proof {
   class ProofStep {...}
   class InnerProof extends Proof {
      val parent:Proof.this.type = Proof.this // The encapsulating proof
      type Fact = parent.Fact | this.ProofStep
   }
}
class BaseProof extends Proof {
   type Fact = Theorem | Axiom | Definition | this.ProofStep
}
```

Figure 7 Simplified outline of the type structure for proof constructors and their facts.

Definitions. Transparent definitions come for free with the Scala host language (see line 2 of Figure 6), these are not visible to the kernel. The DSL offers syntax for the *non-transparent definitions*. Predicate symbol and function symbol (of which constant symbols are a special type) definitions can be direct, as illustrated by the two first examples in Figure 8. Function symbols can also be defined by unique existence, as shown in the last example. Note that this is an example of an *underspecified definition*, as mentioned in the previous Section. It defines a constant symbol `nonEmpty` with only the property `¬(nonEmpty≡∅)`, but the given proof shows the existence of a specific non-empty set.

```
1 val succ = DEF(x) → union(uPair(x, singleton(x)))
2 val inductive = DEF(x) → in(∅, x) ∧ ∀(y, in(y, x) ⟹ in(succ(y), x))
3 val nonEmptySetExists = Lemma( ∃!(x, ¬(x ≡ ∅) ∧ (x ≡ uPair(∅, ∅))){...}
4 val nonEmpty = DEF() → The(x, ¬(x ≡ ∅))(nonEmptySetExists)
```

Figure 8 Definitions in LISA.

4 Tactics in LISA and Comparison

Developing proof tactics in proof assistants where the proof-writing language is different from the host language (and sometimes when both are different from the tactic-writing language) tends to exhibit high entry barriers for newcomers. They require learning multiple new languages and how they interact with each other. This difficulty can be observed for example with the length of the tactic-writing tutorial for Isabelle [40], or in the Coq Reference Manual, where the Ltac tactic language [13] is described as *having unclear semantics, being slow, non-uniform, error-prone* and even lacking essential programming features such as *data structures*. Ltac2 [24], yet another tactic language, aims to solve *some* of these problems. Newly developed systems, such as Lean [12], have the advantage of being designed from scratch and addressing these problems. We have similar aims with LISA, but rely on an existing programming language which has already solved those issues, has an active user base that draws on more than the development of theorems and has well-developed and actively maintained IDEs and libraries. In particular, for a LISA user, seeing how a proof tactic works is ever only a ctrl-click away from their proof and when a new tactic is written, using it is as simple as writing `import MyTactic`.

Not unlike in HOL Light, where a proof tactic is essentially any function returning a value of type Theorem, a tactic in LISA is simply a function returning a proof or an error message. The tactic can take arbitrary arguments, such as a target sequent and known facts (which will be imports of the resulting proof) and can access the current state of the proof constructor (if needed). To write a low level or highly optimized proof tactic, the user can directly construct a sequent calculus proof and give it to the kernel, but they can also use LISA's DSL directly inside the body of the function and use pre-existing proof tactics. Writing a tactic then consists in writing a generic LISA proof computationally.

LISA defines tactics that correspond to each basic proof step within the kernel, but with all the parameters automatically inferred. These tactics are intended for didactic purpose. Compared to directly using kernel proof steps, these simple tactics are more convenient to write, but also slightly less efficient to check because the system needs to compute the parameters of the proof step. Moreover, most of these simple tactics are subsumed by more general tactics.

4.1 A Proof-Producing SAT Solver Using F(OL)^2

The Tautology tactic is able to prove any valid sequent that requires only propositional reasoning. It is based on a simple proof-producing DPLL-like [10] procedure complete for propositional logic. The procedure makes decisions on atoms, so the worst case complexity is exponential in the number of unique atoms in the formula. It is a non-clausal solver (like, e.g., [25]) whose unique aspect is that, between each decision, it simplifies the propositional formula using the algorithm presented in Subsection 2.2). In the context of proving validity as in LISA as well as when trying to find a satisfying assignment in a SAT solver, this allows

to close branches early in the exploration of the decision tree, or simply to eliminate atoms before they even need to get decided. Moreover, this procedure does not need to compute Tseytin's normal form, avoiding creating more atoms, and conveniently allows producing a proof of the statement. Figure 9 sketches the proof search procedure as it is implemented in LISA. Our current implementation uses a simple decision heuristics that picks the atom that occurs most frequently. Further work may also include extension of the algorithm with quantifier reasoning, to obtain a complete procedure for FOL.

```
def solveFormula(f: Formula,
            decisionsPos:List[Formula],
            decisionsNeg:List[Formula]): ProofTacticJudgement = {
  val redF = reduceWithFol2(f)
  if (redF == ⊤) {
    Restate(decisionsPos ⊢ f :: decisionsNeg)
  } else if (redF == ⊥) {
    InvalidProofTactic("Sequent is not a propositional tautology")
  } else {
    val atom = findBestAtom(redF)
    val substInRedF: Formula => Formula = (f => RedF[atom:=f])
    TacticSubproof {
      have(solveFormula(substInRedF(⊤), atom::decisionsPos, decisionsNeg))
      val step2 = thenHave(atom :: decisionsPos ⊢ redF :: decisionsNeg)
            by Substitution(⊤ <=> atom)
      have(solveFormula(substInRedF(⊥), decisionsPos, atom::decisionsNeg))
      val step4 = thenHave(decisionsPos ⊢ redF :: atom :: decisionsNeg)
            by Substitution(⊥ <=> atom)
      thenHave(decisionsPos ⊢ redF :: decisionsNeg)
            by Cut(step2, step4)
      thenHave(decisionsPos ⊢ f :: decisionsNeg)
            by Restate
    }
  }
}
```

Figure 9 Outline of the F(OL)^2-based solver. Note that the actual implementation produces directly kernel proofs for optimization. Each recursive call to `solveFormula` adds at most 4 kernel steps to the final proof.

Thanks to properties of ortholattices, the solver is already capable of resolving propositional problems that are too difficult for some proof assistants. As an example, we found that Isabelle's Blast tactic (a general tableau prover, [34]) was in general not able to prove the equivalence of two reasonably large formulas made only of variables, disjunctions and conjunctions which only differed in the ordering of their arguments. On the other hand, this is instantaneous (one step) with our described OL-based approach.

4.2 Error Reporting

LISA's DSL also contains a printer for proof (both kernel and high level) and defines specialized error reporting. Tactics are allowed to fail if they are used incorrectly and return an error. Figure 10 shows LISA's output for an incorrect proof, with the current state of the proof, the faulty step, its line number and the error message from the tactic.

```
∀x.  P(x) ⟹ P(f(x)) ⊢ P(x) ⟹ P(f(f(x)))
  0 Hypothesis ∀x. P(x) ⟹ P(f(x)) ⊢ ∀x. P(x) ⟹ P(f(x))
  1 Hypothesis P(x); ∀x. P(x) ⟹ P(f(x)) ⊢ P(x)
  2 InstantiateForall P(x); ∀x. P(x) ⟹ P(f(x)) ⊢ P(x) ⟹ P(f(x))
  3 InstantiateForall P(x); ∀x. P(x) ⟹ P(f(x)) ⊢ P(f(x)) ⟹ P(f(f(x)))
  have(thesis) by Tautology.from(step1)

Proof tactic Tautology used in (Example.scala:47) did not succeed:
   The statement is not provable within propositional logic.
   The proof search needs the truth of the following sequent:
   P(f(x)); P(x); ∀x. ¬(P(x) ∧ ¬P(f(x))) ⊢ P(f(f(x)))
```

Figure 10 LISA's output when the step in line 12 of proof in Figure 5 is incorrectly modified to not use `step2`. The indicated sequent in fact corresponds to `step2`.

5 Beginning Set Theory Development and Cantor's theorem

In this section, we present a brief overview of the current mathematical development in LISA and outline an example of a short proof in set theory.

Inspired by Mizar[31] and Isabelle/HOTG[6] we make the choice of Tarski-Grothendieck set theory (TG) as the axiomatic foundation for LISA's associated mathematical library. As the main reference for the ZFC aspect of the set theory development, we follow Thomas Jech's book *Set Theory* [26]. In the future, we plan to use the axiom on Grothendieck universes (corresponding to the existence of certain large cardinals) to support the embedding of category theory and of systems such as Coq [44].

5.1 Current Theory Development

The mathematical library in LISA begins with the ZFC (and TG) axioms, defining the basic constructs and operations on sets, the subset relation, the empty set, power sets, and unordered pairs. On top of thes axioms, we define structures such as ordered pairs, relations, and functions. Relations are sets of ordered pairs drawing elements from a set, and functions are relations which contain the graph of the function. Function symbols have as a domain the whole set space and must not be mixed with function objects, which are special sets and considered as constants in the light of first order logic. During exploratory development, proofs involving case analysis on these basic structures required significant manual effort, but the `Tautology` and `Substitution` tactics as well as the quick instantiation of axioms and theorems offered by the `of` keyword tend to automate most tedious manipulations. Formalization of partial orders, well-ordered sets, ordinals and induction [26, Chapters 2, 3] is ongoing.

Technically, we define for sets A and B the set of relations from A to B as the power set of their Cartesian product $P(A \times B)$, and its restriction to functional relations, $A \to B$, the set of all functions with domain equal to A and their codomain included in B.

A function symbol can always apply to any term, meaning we cannot rely on well-definedness of terms to define symbols with partial function semantics. Considering the unique existence requirement for definitions, the standard approach consists in extending the limited domain of the partial function by assigning a default value, for example the empty set, to all inputs where it should be undefined, constructing a unique object. This specific construction and default value can then be forgotten using an underspecified definition. In

particular, interpretations of the function with all combinations of values outside the fixed domain will be valid models for the symbol, and no non-trivial property can be proved about those values.

For example, consider the definition for function application, $\text{app}(f, x)$. When f is not a functional relation, or x is not in its domain, we fix $\emptyset$ as the default value in order to obtain a proof of existence and uniqueness.

```
val appDefinition = Theorem( ∀(f, (∀ x, (∃!(z,
      functional(f) ∧ in(x, dom(f)) ⟹ in(pair(x, z), f)))) )
   ∧ ¬functional(f) ∨ ¬in(x, dom(f)) ⟹ z ≡ ∅)
```

We can then obtain the function symbol `app` with only the desired property using an underspecified definition:

```
val app = DEF (f, x) → The(z,
         functional(f) ∧ in(x, dom(f)) ⟹ in(pair(x, z), f))
         (appDefinition)
```

Cantor's Theorem

Finally, several of these definitions and lemmas build up to the formalization of Cantor's theorem, stating that there is no surjection from any set to its power set:

```
val cantorTheorem = Theorem( ¬surjective(f, x, powerSet(x)) )
```

where f and x are schematic set variables, making the sequent implicitly universally quantified. The proof of Cantor's theorem is about 25 lines of code [1]. Internally, the proof expands to 130 sequent calculus steps.

Cantor's theorem is the first theorem formalized in LISA from the list *Formalizing 100 Theorems* [45]. While not a difficult theorem, it requires some ground development and definitions related to set-theoretic functions and relations. The proof itself requires handling the quantifiers for a contradiction construction and combining lemmas about surjective functions. Much of the latter is achieved using `Tautology`. It shows that LISA is capable of non-trivial mathematical development. We expect future developments to become easier and faster with gradual development of reasoning tools and proofs.

6 Related Work

A polynomial algorithm for free ortholattices was presented in [18]. A weaker structure with log linear complexity was first presented in [19]. In LISA we use the ortholattice normal form for first-order logic formulas. Our F(OL)[2] implementation does not aim to be complete for structures such as quantum monadic algebras that treat extensions of OL (and orthomodular lattices) to monadic first-order logic [21].

Much of what we described is concerned with the schematic first-order logic kernel. We chose to include schematic variables to be able to state explicitly the axiom schemas of Zermelo-Fraenkel set theory and its extensions, as well as theorem schemas. Another way to generalize schematic second-order variables would be to use higher-order logic. This is the approach pursued by Isabelle as a framework, and instantiated in Isabelle/ZF.

[1] https://github.com/epfl-lara/lisa/blob/fc37f2a6e879d5f43679a4476c1d6e4685bb14a2/src/main/scala/lisa/mathematics/SetTheory.scala#L1700

The choice of set theory may be considered unusual by some, as Coq [4], Lean [12], the HOL-family [22] and Isabelle/HOL [43] are based either on type theory or on higher-order logic. We consider HOL to be one of the most elegant formulations for formal proof developments. However, set theory is arguably the most widely recognized foundation of mathematics in the mathematical community, and, despite type-theory based tools having the advantage of being easier to express formalisms in from the get-go, we believe that through the development and use of abstracting tactics, a soft-type system and adequate tools, more familiarity and flexibility in writing proofs can be achieved with a set-theory based mathematical library. We also hope to provide a test bed to explore direct first-order foundations as an alternative to the many current systems based on higher-order logic. Concrete results in Mizar [31], Isabelle/ZF [20], ZF in Isabelle/HOL [6, 36], and TLA$^+$ [9, 37] suggest substantial relevance of set-theoretic foundations. Arguments in favour of set theoretic foundations have also been discussed by John Harrison [23] and Bohua Zhan [46].

Even one more level of indirection than in Isabelle/ZF is present in Isabelle/HOL/TG [6], which develops the Tarski-Grothendieck extension of ZF inside Isabelle/HOL. Whereas our system is less flexible and does not currently connect to such a well-developed ecosystem as Isabelle/HOL has, our hope is that it is conceptually simpler thanks to fewer layers and a kernel that does not rely on unification.

Another modern approach to theorem proving is Lean [12], a proof assistant based on dependent type theory and inspired in part by Coq. We believe Lean makes significant improvements over older proof assistant regarding the *Six Virtues*. In particular, it has a strong focus on programmability, with the new version of Lean [30] even having a compiler written in its own proof language. While LISA and Lean's design objectives share similarities, their strategies and specific choice (foundations, language, interface) are different.

To automate proofs that do not instantiate schematic formulas we hope to make use of proof generating theorem provers, such as Vampire [28], SPASS [42], E [38], as well as SMT solvers [2]. Higher-order provers such as Zipperposition [41], Leo-III [39], and Satallax [5] would further increase automation even in the case of axiom schema instantiation.

7 Conclusion

LISA is both a proof system for automated tools and a proof assistant based on first order logic and set theory. It uses Scala as both a host language and a proof writing language, relying on the advanced features it offers to make the system as programmable as the user desires. LISA is strongly committed to interoperability. In particular, it has a small logical kernel which has guaranteed complexity and completeness characterizations, simple foundations and explicit proofs checkable without context. Moreover, it can be compiled into a Scala and Java library. All these properties should favour transfer of proofs from and to other proof systems and uses of LISA as a tool for program verification. To improve usability and reduce the size of proofs, LISA makes use of an efficient normal form algorithm for propositional logic extended to first order logic. This algorithm is also the basis for a complete propositional proof-producing procedure implemented in LISA as a tactic.

LISA is still under active development, but already proposes an advanced proof writing DSL not entirely dissimilar to already existing interpreted languages in other assistants. LISA also allows defining arbitrary tactics in a simple way and has specialized error reporting. The current embryo of set-theoretic development encompasses properties of relations and functions, and in particular Cantor's theorem has been successfully proven.

References

1 Rob Arthan. HOL Constant Definition Done Right. In Gerwin Klein and Ruben Gamboa, editors, *Interactive Theorem Proving*, Lecture Notes in Computer Science, pages 531–536, Cham, 2014. Springer International Publishing. doi:10.1007/978-3-319-08970-6_34.

2 Haniel Barbosa, Clark Barrett, Martin Brain, Gereon Kremer, Hanna Lachnitt, Makai Mann, Abdalrhman Mohamed, Mudathir Mohamed, Aina Niemetz, Andres Nötzli, Alex Ozdemir, Mathias Preiner, Andrew Reynolds, Ying Sheng, Cesare Tinelli, and Yoni Zohar. cvc5: A Versatile and Industrial-Strength SMT Solver. In Dana Fisman and Grigore Rosu, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 13243, pages 415–442. Springer International Publishing, Cham, 2022. doi:10.1007/978-3-030-99524-9_24.

3 Ladislav Beran. *Orthomodular Lattices (An Algebraic Approach)*. Springer Dordrecht, 1985. doi:10.1007/978-94-009-5215-7.

4 Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development: Coq'Art: The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. An EATCS Series. Springer-Verlag, Berlin Heidelberg, 2004. doi:10.1007/978-3-662-07964-5.

5 Chad E. Brown. Satallax: An automatic higher-order prover. In Bernhard Gramlich, Dale Miller, and Uli Sattler, editors, *Automated Reasoning*, pages 111–117, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

6 Chad E. Brown, C. Kaliszyk, and Karol Pak. Higher-Order Tarski Grothendieck as a Foundation for Formal Proof. In *ITP*, 2019. doi:10.4230/LIPIcs.ITP.2019.9.

7 Gunter Bruns and John Harding. Algebraic aspects of orthomodular lattices. In Bob Coecke, David Moore, and Alexander Wilce, editors, *Current Research in Operational Quantum Logic: Algebras, Categories, Languages*, pages 37–65. Springer Netherlands, Dordrecht, 2000. doi:10.1007/978-94-017-1201-9_2.

8 Paul J. Cohen. The independence of the continuum hypothesis. *Proceedings of the National Academy of Sciences of the United States of America*, 50(6):1143–1148, 1963. URL: http://www.jstor.org/stable/71858.

9 Denis Cousineau, Damien Doligez, Leslie Lamport, Stephan Merz, Daniel Ricketts, and Hernán Vanzetto. TLA+ proofs. In Dimitra Giannakopoulou and Dominique Méry, editors, *FM 2012: Formal Methods*, pages 147–154, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

10 Martin Davis, George Logemann, and Donald Loveland. A machine program for theorem-proving. *Communications of the ACM*, 5(7):394–397, July 1962. doi:10.1145/368273.368557.

11 Nicolaas Govert De Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the church-rosser theorem. In *Indagationes Mathematicae (Proceedings)*, volume 75, pages 381–392. Elsevier, 1972.

12 Leonardo de Moura, Soonho Kong, Jeremy Avigad, Floris van Doorn, and Jakob von Raumer. The Lean Theorem Prover (System Description). In Amy P. Felty and Aart Middeldorp, editors, *Automated Deduction - CADE-25*, volume 9195, pages 378–388. Springer International Publishing, Cham, 2015. doi:10.1007/978-3-319-21401-6_26.

13 David Delahaye. A tactic language for the system coq. In *LPAR*, volume 1955, pages 85–95. Springer, 2000.

14 Harald Ganzinger, Christoph Meyer, and Christoph Weidenbach. Soft typing for ordered resolution. In William McCune, editor, *Automated Deduction - CADE-14, 14th International Conference on Automated Deduction, Townsville, North Queensland, Australia, July 13-17, 1997, Proceedings*, volume 1249 of *Lecture Notes in Computer Science*, pages 321–335. Springer, 1997. doi:10.1007/3-540-63104-6_32.

15 G. Gentzen. Untersuchungen über das logische schließen i. *Mathematische Zeitschrift*, 39:176–210, 1935. URL: http://eudml.org/doc/168546.

16 Michael J. C. Gordon, Robin Milner, Christopher P. Wadsworth, and P. Ted Christopher. Edinburgh LCF: a mechanized logic of computation. *Lecture Notes in Computer Science*, 1978.

17 Simon Guilloud. LISA Reference Manual. EPFL-LARA, February 2023.

18 Simon Guilloud, Mario Bucev, Dragana Milovancevic, and Viktor Kunčak. Formula normalizations in verification. Technical Report 297701, EPFL, 2023. URL: http://infoscience.epfl.ch/record/297701.

19 Simon Guilloud and Viktor Kunčak, editors. *Equivalence Checking for Orthocomplemented Bisemilattices in Log-Linear Time*. Springer, 2022. doi:10.48550/arXiv.2110.03315.

20 Emmanuel Gunther, Miguel Pagano, Pedro Sánchez Terraf, and Matías Steinberg. The independence of the continuum hypothesis in isabelle/zf. *Archive of Formal Proofs*, March 2022. , Formal proof development. URL: https://isa-afp.org/entries/Independence_CH.html.

21 J Harding. Quantum monadic algebras. *Journal of Physics A: Mathematical and Theoretical*, 55(39):394001, September 2022. doi:10.1088/1751-8121/ac845b.

22 John Harrison. HOL Light: An Overview. In Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel, editors, *Theorem Proving in Higher Order Logics*, volume 5674, pages 60–66. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009. doi:10.1007/978-3-642-03359-9_4.

23 John Harrison. Let's make set theory great again! In *Axiomatic Set Theory*, page 46, Aussois, 2018.

24 CNRS Inria and contributors. Ltac2 — Coq 8.16.1 documentation. URL: https://coq.inria.fr/refman/proof-engine/ltac2.html.

25 Himanshu Jain, Constantinos Bartzis, and Edmund Clarke. Satisfiability checking of non-clausal formulas using general matings. In Armin Biere and Carla P. Gomes, editors, *Theory and Applications of Satisfiability Testing - SAT 2006*, pages 75–89, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg. doi:10.1007/11814948_10.

26 Thomas Jech. *Set theory: The third millennium edition, revised and expanded*. Springer, 2003.

27 Deepak Kapur and Paliath Narendran. Complexity of unification problems with associative-commutative operators. *J. Autom. Reason.*, 9(2):261–288, 1992. doi:10.1007/BF00245463.

28 Laura Kovács and Andrei Voronkov. First-Order Theorem Proving and Vampire. In Natasha Sharygina and Helmut Veith, editors, *Computer Aided Verification*, Lecture Notes in Computer Science, pages 1–35, Berlin, Heidelberg, 2013. Springer. doi:10.1007/978-3-642-39799-8_1.

29 Norman Megill. Metamath. *The Seventeen Provers of the World: Foreword by Dana S. Scott*, pages 88–95, 2006.

30 Leonardo de Moura and Sebastian Ullrich. The lean 4 theorem prover and programming language. In André Platzer and Geoff Sutcliffe, editors, *Automated Deduction – CADE 28*, pages 625–635, Cham, 2021. Springer International Publishing.

31 Adam Naumowicz and Artur Kornilowicz. A Brief Overview of Mizar. In *Proceedings of the 22nd International Conference on Theorem Proving in Higher Order Logics*, volume 5674, pages 67–72, August 2009. doi:10.1007/978-3-642-03359-9_5.

32 Martin Odersky, Aggelos Biboudis, Fengyun Liu, and Olivier Blanvillain. Foundations of implicit function types. Technical report, EPFL, 2017. URL: http://infoscience.epfl.ch/record/229203.

33 Lawrence C. Paulson. Isabelle: The next 700 theorem provers. *CoRR*, cs.LO/9301106, 1993. URL: https://arxiv.org/abs/cs/9301106.

34 Lawrence C. Paulson. A generic tableau prover and its integration with Isabelle. *JUCS - Journal of Universal Computer Science*, 5(3):73–87, 1999. doi:10.3217/jucs-005-03-0073.

35 Lawrence C Paulson. Isabelle's logics: FOL and ZF, 2013.

36 Lawrence C. Paulson. Zermelo Fraenkel set theory in higher-order logic. *Archive of Formal Proofs*, October 2019. , Formal proof development. URL: https://isa-afp.org/entries/ZFC_in_HOL.html.

37 TLA Proof System Project. TLA+ proof system. URL: https://tla.msr-inria.inria.fr/tlaps/content/Home.html.

38 Stephan Schulz. System Description: E 1.8. In Ken McMillan, Aart Middeldorp, and Andrei Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning*, Lecture Notes in Computer Science, pages 735–743, Berlin, Heidelberg, 2013. Springer. doi:10.1007/978-3-642-45221-5_49.

39 Alexander Steen. Leo-iii 1.7, July 2022. doi:10.5281/zenodo.7650205.

40 Christian Urban. The Isabelle Cookbook. URL: https://web.cs.wpi.edu/~dd/resources_isabelle/isabelle_programming.urban.pdf.

41 Petar Vukmirović, Alexander Bentkamp, Jasmin Blanchette, Simon Cruanes, Visa Nummelin, and Sophie Tourret. Making Higher-Order Superposition Work. In André Platzer and Geoff Sutcliffe, editors, *Automated Deduction – CADE 28*, Lecture Notes in Computer Science, pages 415–432, Cham, 2021. Springer International Publishing. doi:10.1007/978-3-030-79876-5_24.

42 Christoph Weidenbach, Dilyana Dimova, Arnaud Fietzke, Rohit Kumar, Martin Suda, and Patrick Wischnewski. SPASS Version 3.5. In Renate A. Schmidt, editor, *Automated Deduction – CADE-22*, Lecture Notes in Computer Science, pages 140–145, Berlin, Heidelberg, 2009. Springer. doi:10.1007/978-3-642-02959-2_10.

43 Makarius Wenzel, Lawrence C. Paulson, and Tobias Nipkow. The Isabelle Framework. In Otmane Ait Mohamed, César Muñoz, and Sofiène Tahar, editors, *Theorem Proving in Higher Order Logics*, Lecture Notes in Computer Science, pages 33–38, Berlin, Heidelberg, 2008. Springer. doi:10.1007/978-3-540-71067-7_7.

44 Benjamin Werner. Sets in types, types in sets. In Gerhard Goos, Juris Hartmanis, Jan van Leeuwen, Martín Abadi, and Takayasu Ito, editors, *Theoretical Aspects of Computer Software*, volume 1281, pages 530–546. Springer Berlin Heidelberg, Berlin, Heidelberg, 1997. doi:10.1007/BFb0014566.

45 Freek Wiedijk. Formalizing 100 theorems. https://www.cs.ru.nl/~freek/100/.

46 Bohua Zhan. Formalization of the Fundamental Group in Untyped Set Theory Using Auto2. In Mauricio Ayala-Rincón and César A. Muñoz, editors, *Interactive Theorem Proving*, Lecture Notes in Computer Science, pages 514–530, Cham, 2017. Springer International Publishing. doi:10.1007/978-3-319-66107-0_32.