

1 LISA – A Modern Proof System

2 Simon Guilloud  

3 EPFL, Laboratory for Automated Reasoning and Analysis, Switzerland

4 Sankalp Gambhir  

5 EPFL, Laboratory for Automated Reasoning and Analysis, Switzerland

6 Viktor Kunčák  

7 EPFL, Laboratory for Automated Reasoning and Analysis, Switzerland

8 — Abstract —

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

21 **2012 ACM Subject Classification** Theory of computation → Logic and verification

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

23 **Digital Object Identifier** 10.4230/LIPIcs...

24 **1** Introduction

25 We present the design and initial implementation of a new proof assistant, named LISA.
26 Much like Mizar [30], LISA aims to use classical mainstream foundations of mathematics with
27 first order logic and set theory. LISA uses (single-sorted) first-order logic (with schematic
28 variables) as the syntactic framework, sequent calculus as the deduction framework and set
29 theory as the semantic framework. On top of this foundation, we can construct mathematical
30 theories without introducing additional axioms. As the target use of LISA we envision a
31 library of theorems, but also correctness proofs of computer systems.

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

<https://github.com/epfl-lara/lisa>

34 **1.1** Design Goals

35 Our design is inspired by the LCF line of proof assistants, including HOL Light, HOL4, and
36 Isabelle. The envisioned path for axiomatic foundations is closer to Mizar. LISA’s logical
37 kernel is a hybrid between LCF-style encoding of theorems as a sealed Theorem type (similar
38 to HOL Light [21]) and explicit requirement of proofs. Namely, proofs are self-contained
39 sequences of proof steps that derive a conclusion from assumptions (they are not explicitly
40 in the form of lambda terms). LISA’s kernel checks the validity of steps, the validity of
41 assumptions, and then creates an instance of a theorem.



© Simon Guilloud, Sankalp Gambhir and Viktor Kunčák;
licensed under Creative Commons License CC-BY 4.0

Leibniz International Proceedings in Informatics

LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

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

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

71 1.2 Contributions

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

- 75 ■ We describe how the logical kernel is constructed and how it can be used or interacted
 76 with by other tools.
- 77 ■ As the most unusual design aspect, we describe ortholattice-based algorithms implemented
 78 in the kernel to make proofs shorter.
- 79 ■ We present a domain-specific language embedded in Scala that makes the writing of
 80 proofs easier and generates and checks kernel proofs to obtain kernel-certified theorems.
- 81 ■ We show that the same domain-specific language can scale from writing proofs of specific
 82 theorems to writing general tactics. As an example of a tactic, we present a (proof
 83 generating) solver for propositional formulas leveraging the ortholattice algorithm.
- 84 ■ 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) [14]. Formally, a sequent in LISA is a pair of sets of formulas Γ and Δ , represented $\Gamma \vdash \Delta$ and its interpretation is $\bigwedge \Gamma \rightarrow \bigvee \Delta$. 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 :

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

This means that any specific term with a distinguished free variable replacing f 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 [32], and in particular Isabelle/FOL [34], 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) [2, Chapter II.1], [6].

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 [17] 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

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': $x \wedge (x \vee y) = x$

■ **Table 1** Laws of ortholattices, an algebraic theory with signature $(S, \wedge, \vee, 0, 1, \neg)$. [17]

signature $(S, \wedge, \vee, 0, 1, \neg)$, we denote $s \leq_{OL} t$ if and only if $OL \models s \leq t$, 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 $OL \models s = t$). Theorem 1 is the main result we rely on.

► **Theorem 1** ([17]). *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 [10], then desugars $\exists.\phi$ into $\neg\forall.\neg\phi$. It then extends the OL algorithm with the rules in Table 2.

	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	$\phi \leq t_1 = t_2$	$s_1 == s_2$
5	$\forall.\phi \leq \forall.\psi$	$\phi \leq \psi$
6	$\mathcal{C}(\phi_1, \dots, \phi_n) \leq \mathcal{C}(\psi_1, \dots, \psi_n)$	$\phi_i \sim_{OL} \psi_i$, for every $1 \leq i \leq n$
7	Anything else	false

■ **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.

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 [17], 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 \mathcal{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

149 \leq is interpreted as logical implication, they are sound. We decided not to include a rule
 150 such as $\forall. \phi \leq \phi(t)$. The reason is that incorporating such a rule systematically runs risk
 151 of introducing higher complexity [26] in the kernel. We instead decided that such steps
 152 should be implemented using tactics, outside the kernel in the future (possibly making use of
 153 type-like hints encoded in first-order logic [13]).

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

159 Moreover, the proof checker contains a special **Restate** proof step, which permits
 160 $F(OL)^2$ -transformations on the entire sequent, leveraging the interpretation of a sequent
 161 as a formula (an implication). We also leverage specifically the partial order computed
 162 by $F(OL)^2$ to expand the usual **Weakening** rule so that the premise sequent only has to
 163 be $\leq_{F(OL)^2}$ stronger than the conclusion, with both interpreted as formulas. **Weakening**
 164 clearly subsumes **Restate**, but the latter ensures that the transformation is actually an
 165 equivalence and hence could be reversed, which can be a useful safeguard in practice. These
 166 rules subsume most propositional rules in Figure 1.

167 2.3 Substitution Rules

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

$$174 \frac{\phi \vdash \psi}{\phi \vdash \psi[\phi := \top]} \text{SubstIff}$$

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

178 The inclusion of $F(OL)^2$ and the substitution and instantiation deduced rules in the
 179 logical kernel is a slight bend to the trust principle, but as the algorithm is only 300 lines
 180 of code, this is largely overshadowed by the increased usability and shorter proofs. In fact,
 181 the whole kernel adds up to a grand total of only 1607 lines of code. This comprises the
 182 implementation of first-order logic, the $F(OL)^2$ algorithm, first and second-order substitution,
 183 the sequent calculus steps, the proof checker, and a manager for definitions and theorems
 184 (detailed in Subsection 2.5). Moreover, LISA’s kernel is efficient: except for the quadratic
 185 $F(OL)^2$ algorithm, every procedure in the Kernel is linear (up to logarithmic coefficients) in
 186 the size of the formulas or proofs being considered.

187 2.4 Proof Objects

188 In LISA, a proof is an explicit list of proof steps, where each step can refer to previous steps
 189 via their respective position in the list and be referred by multiple subsequent steps. In
 190 other words, a proof is represented as a topological linearization of the proof tree, or, more
 191 generally, a directed acyclic graph (permitting reuse of intermediate steps). A proof step

$$\begin{array}{c}
\frac{}{\Gamma, \phi \vdash \phi, \Delta} \text{Hypothesis} \\
\\
\frac{\Gamma \vdash \phi, \Delta \quad \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 \quad \Sigma \vdash \psi, \Pi}{\Gamma, \Sigma \vdash \phi \wedge \psi, \Delta, \Pi} \text{RightAnd} \\
\\
\frac{\Gamma, \phi \vdash \Delta \quad \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 \quad \Sigma, \psi \vdash \Pi}{\Gamma, \Sigma, \phi \rightarrow \psi \vdash \Delta, \Pi} \text{LeftImplies} \qquad \frac{\Gamma, \phi \vdash \psi, \Delta}{\Gamma \vdash \phi \rightarrow \psi, \Delta} \text{RightImplies} \\
\\
\frac{\Gamma, \phi \rightarrow \psi \vdash \Delta}{\Gamma, \phi \leftrightarrow \psi \vdash \Delta} \text{LeftIff} \qquad \frac{\Gamma \vdash \phi \rightarrow \psi, \Delta \quad \Sigma \vdash \psi \rightarrow \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 if } (\bigwedge \Gamma_1 \rightarrow \bigvee \Delta_1) \sim_{F(OL)^2} (\bigwedge \Gamma_2 \rightarrow \bigvee \Delta_2) \\
\\
\frac{\Gamma \vdash \Delta}{\Gamma, \phi \vdash \psi, \Delta} \text{Weakening if } (\bigwedge \Gamma_1 \rightarrow \bigvee \Delta_1) \leq_{F(OL)^2} (\bigwedge \Gamma_2 \rightarrow \bigvee \Delta_2)
\end{array}$$

■ **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.

192 also contains the arguments that allow the proof checker to efficiently verify it. In particular,
 193 LISA’s kernel does not rely on a unification algorithm to check correctness of proof steps
 194 related to quantifiers.

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

203 If the proof relies on external theorems, axioms or definitions, those are stated after the
 204 list of proof steps and referred to with negative positions. We call those *imported* sequents
 205 (*imports*, for short). We adopt an analogous mechanism to support *subproofs*. A subproof
 206 simulates deduced steps by encapsulating an inner proof and appears as a single step in the
 207 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)		$\vdash \phi, (\phi \rightarrow \psi)$
3 LeftImplies(2,0)	$(\phi \rightarrow \psi) \rightarrow \phi$	$\vdash \phi$
4 RightImplies(3)		$\vdash ((\phi \rightarrow \psi) \rightarrow \phi) \rightarrow \phi$

■ **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(           $\phi \vdash \phi,$                  $\phi$ ),
3   Weakening(           $\phi \vdash (\phi, \psi),$            $\emptyset$ ),
4   RightImplies(       $() \vdash (\phi, \phi \implies \psi),$    $1, \phi, \psi$ ),
5   LeftImplies(       $(\phi \implies \psi) \implies \phi \vdash \phi,$    $2, \emptyset, (\phi \implies \psi), \phi$ ),
6   RightImplies(       $() \vdash ((\phi \implies \psi) \implies \phi) \implies \phi,$ 
7                                      $3, (\phi \implies \psi) \implies \phi, \phi$ )
8 ), Seq.empty /* no imports */)

```

■ **Figure 3** The proof from Figure 2 written for LISA’s kernel. \vdash and \implies 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.

208 2.5 Theories

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

214 **Theorems**

215 This part of the kernel, called the **Theory**, is inspired from the LCF style [15]. It allows
 216 checking a proof once, producing a value of a sealed type **Theorem**, which can then be reused
 217 many times. The proof can then be forgotten. The **Theory** will also verify that the given
 218 proof’s imports are properly justified by existing axioms, theorems or definitions, so that the
 219 proven **Theorem** can be considered unconditionally true, unlike its standalone proof. Figure 4
 220 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.

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

229 **Definitions**

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

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

$$234 \quad \exists!y. \phi_{y, x_1, \dots, x_n} \tag{1}$$

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

238 **Remark on unique existence.** One may hope that only the existence (but not uniqueness)
 239 was needed to obtain conservative extensions in first-order logic. Unfortunately, this is not
 240 true in the presence of axiom schemas. In particular, with such a definition principle, it
 241 becomes possible to prove the Axiom of Choice in ZF set theory, while they are well known
 242 to be independent [7]. Indeed, in ZF it is possible to prove

$$243 \quad \forall x. \exists y. (x \neq \emptyset \implies y \in x)$$

244 from which we would obtain a function `pick` with the property

$$245 \quad \forall x. (x \neq \emptyset \implies \text{pick}(x) \in x)$$

246 If then the symbol pick is allowed in axiom schemas, as would be the case in LISA, it is then
247 easy to use pick and the replacement schema to construct a choice function on any set (see
248 also LISA's Reference Manual [16]).

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

263 This issue is addressed in Metamath [28] by assuming a specific construction of the
264 structure to conditionally prove a desired defining property (for example, the axioms of the
265 real field) and then introducing said property independently as an axiom ¹. This mechanism
266 however is not enforced by Metamath itself but only an informal practice. LISA's kernel
267 support for underspecified definitions ensures that the same goal is achieved with guaranteed
268 soundness.

269 **3 DSL for LISA in Scala**

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

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

¹ Example: <https://us.metamath.org/mpeuni/ax-resscn.html>

```

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      $\forall(x, P(x) \implies P(f(x))) \vdash P(x) \implies P(f(f(x)))$ 
8   ) {
9     assume( $\forall(x, P(x) \implies P(f(x)))$ )
10    val step1 = have( $P(x) \implies P(f(x))$ ) by InstantiateForall
11    val step2 = have( $P(f(x)) \implies P(f(f(x)))$ ) 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.

288 3.1 Higher-Level Proofs

289 LISA’s interface defines a proof constructing class. This class uses proof tactics to generate
 290 pieces of the final pure sequent calculus proof, which are encapsulated into kernel subproofs.
 291 The result from the point of view of the user is the ability to define arbitrarily computed
 292 deduced proof steps (here `Tautology` and `InstantiateForall`) from the base steps of
 293 sequent calculus. Thanks to Scala 3’s implicit functions types [31], the proof constructor is
 294 automatically created in the code block following the `Theorem` declaration (line 6 of Figure 5)
 295 without the need for the user to even realize it exists. The existence of an implicit proof
 296 constructor in scope is necessary for the other keywords (`have`, `assume`, ...) to be well-defined,
 297 meaning that using those outside of a theorem environment will fail to compile.

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

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

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

315 When a tactic requires a single premise, and this premise is the most recently proven fact,
 316 `thenHave` passes said premise directly to the tactic without the step having to be named. For
 317 some tactics, such as the `Substitution` step at line 21, the resulting sequent will be inferred
 318 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.

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

```

1  class Proof {
2    class ProofStep {...}
3    class InnerProof extends Proof {
4      val parent:Proof.this.type = Proof.this // The encapsulating proof
5      type Fact = parent.Fact | this.ProofStep
6    }
7  }
8  class BaseProof extends Proof {
9    type Fact = Theorem | Axiom | Definition | this.ProofStep
10 }

```

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

322 **Definitions** Transparent definitions come for free with the Scala host language (see line 2 of
323 Figure 6), these are not visible to the kernel. The DSL offers syntax for the *non-transparent*
324 *definitions*. Predicate symbol and function symbol (of which constant symbols are a special
325 type) definitions can be direct, as illustrated by the two first examples in Figure 8. Function
326 symbols can also be defined by unique existence, as shown in the last example. Note that
327 this is an example of an *underspecified definition*, as mentioned in the previous Section. It
328 defines a constant symbol `nonEmpty` with only the property $\neg(\text{nonEmpty} \equiv \emptyset)$, but the given
329 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( $\emptyset$ , x)  $\wedge$   $\forall$ (y, in(y, x)  $\implies$  in(succ(y), x))
3 val nonEmptySetExists = Lemma( $\exists!$ (x,  $\neg$ (x  $\equiv$   $\emptyset$ )  $\wedge$  (x  $\equiv$  uPair( $\emptyset$ ,  $\emptyset$ )))){...}
4 val nonEmpty = DEF() → The(x,  $\neg$ (x  $\equiv$   $\emptyset$ ))(nonEmptySetExists)

```

■ Figure 8 Definitions in LISA

330 4 Tactics in LISA and Comparison

331 Developing proof tactics in proof assistants where the proof-writing language is different
332 from the host language (and sometimes when both are different from the tactic-writing
333 language), tend to exhibit high entry barriers for newcomers. They require learning multiple
334 new languages and how they interact with each other. This difficulty can be observed for
335 example with the length of the tactic-writing tutorial for Isabelle [39], or in the Coq Reference
336 Manual, where the Ltac tactic language [12] is described as *having unclear semantics, being*
337 *slow, non-uniform, error-prone* and even lacking essential programming features such as *data*
338 *structures*. Ltac2 [23], yet another tactic language, aims to solve *some* of these problems.
339 Newly developed systems, such as Lean [11], have the advantage of being designed from
340 scratch and addressing these problems. We have similar aims with LISA, but rely on an
341 existing programming language which has already solved those issues, has an active user base
342 that draws on more than the development of theorems and has well-developed and actively
343 maintained IDEs and libraries. In particular, for a LISA user, seeing how a proof tactic
344 works is ever only a ctrl-click away from their proof and when a new tactic is written, using
345 it is as simple as writing `import MyTactic`.

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

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

360 4.1 A Proof-Producing SAT Solver Using F(OL)²

361 The Tautology tactic is able to prove any valid sequent that requires only propositional
362 reasoning. It is based on a simple proof-producing DPLL-like [9] procedure complete for
363 propositional logic. The procedure makes decisions on atoms, so the worst case complexity is
364 exponential in the number of unique atoms in the formula. It is a non-clausal solver (like,
365 e.g., [24]) whose unique aspect is that, between each decision, it simplifies the propositional
366 formula using the algorithm presented in Subsection 2.2). In the context of proving validity
367 as in LISA as well as when trying to find a satisfying assignment in a SAT solver, this allows

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

```

1  def solveFormula(f: Formula,
2      decisionsPos:List[Formula],
3      decisionsNeg:List[Formula]): ProofTacticJudgement = {
4  val redF = reduceWithFol2(f)
5  if (redF ==  $\top$ ) {
6      Restate(decisionsPos  $\vdash$  f :: decisionsNeg)
7  } else if (redF ==  $\perp$ ) {
8      InvalidProofTactic("Sequent is not a propositional tautology")
9  } else {
10     val atom = findBestAtom(redF)
11     val substInRedF: Formula => Formula = (f => RedF[atom:=f])
12     TacticSubproof {
13         have(solveFormula(substInRedF( $\top$ ), atom::decisionsPos, decisionsNeg))
14         val step2 = thenHave(atom :: decisionsPos  $\vdash$  redF :: decisionsNeg)
15             by Substitution( $\top$  <=> atom)
16         have(solveFormula(substInRedF( $\perp$ ), decisionsPos, atom::decisionsNeg))
17         val step4 = thenHave(decisionsPos  $\vdash$  redF :: atom :: decisionsNeg)
18             by Substitution( $\perp$  <=> atom)
19         thenHave(decisionsPos  $\vdash$  redF :: decisionsNeg)
20             by Cut(step2, step4)
21         thenHave(decisionsPos  $\vdash$  f :: decisionsNeg)
22             by Restate
23     }
24 }
25 }
```

■ **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.

374
 375 Thanks to properties of ortholattices, the solver is already capable of resolving proposi-
 376 tional problems that are too difficult for some proof assistants. As an example, we found
 377 that Isabelle’s Blast tactic (a general tableau prover, [33]) was in general not able to prove
 378 the equivalence of two reasonably large formulas made only of variables, disjunctions and
 379 conjunctions which only differed in the ordering of their arguments. On the other hand, this
 380 is instantaneous (one step) with our described OL-based approach.

381 4.2 Error Reporting

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

```

386  $\forall x. P(x) \implies P(f(x)) \vdash P(x) \implies P(f(f(x)))$ 
387 0 Hypothesis  $\forall x. P(x) \implies P(f(x)) \vdash \forall x. P(x) \implies P(f(x))$ 
388 1 Hypothesis  $P(x); \forall x. P(x) \implies P(f(x)) \vdash P(x)$ 
389 2 InstantiateForall  $P(x); \forall x. P(x) \implies P(f(x)) \vdash P(x) \implies P(f(x))$ 
390 3 InstantiateForall  $P(x); \forall x. P(x) \implies P(f(x)) \vdash P(f(x)) \implies P(f(f(x)))$ 
391 have(thesis) by Tautology.from(step1)
392
393 Proof tactic Tautology used in (Example.scala:47) did not succeed:
394 The statement is not provable within propositional logic.
395 The proof search needs the truth of the following sequent:
396  $P(f(x)); P(x); \forall x. \neg(P(x) \wedge \neg P(f(x))) \vdash P(f(f(x)))$ 

```

■ **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`.

386 5 Beginning Set Theory Development and Cantor’s theorem

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

389 Inspired by [30] and [5] we make the choice of Tarski-Grothendieck set theory (TG) as
 390 the axiomatic foundation for LISA’s associated mathematical library. As the main reference
 391 for the ZFC aspect of the set theory development, we follow Thomas Jech’s book *Set Theory*
 392 [25]. In the future, we plan to use the axiom on Grothendieck universes (corresponding to
 393 the existence of certain large cardinals) to support the embedding of category theory and
 394 systems such as Coq [43].

395 5.1 Current Theory Development

396 The mathematical library in LISA begins with the ZFC (and TG) axioms, defining the
 397 basic constructs and operations on sets, the subset relation, the empty set, power sets, and
 398 unordered pairs. On top of these axioms, we define structures such as ordered pairs, relations,
 399 and functions. Relations are sets of ordered pairs drawing elements from a set, and functions
 400 are relations which contain the graph of the function. Function symbols have as a domain
 401 the whole set space and must not be mixed with function objects, which are special sets and
 402 considered as constants in the light of first order logic. During exploratory development,
 403 proofs involving case analysis on these basic structures required significant manual effort,
 404 but the `Tautology` and `Substitution` tactics as well as the quick instantiation of axioms
 405 and theorems offered by the `of` keyword tend to automate most tedious manipulations.
 406 Formalization of partial orders, well-ordered sets, ordinals and induction [25, Chapters 2, 3]
 407 is ongoing.

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

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

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

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

```
423
424 val appDefinition = Theorem(  $\forall(f, (\forall x, (\exists!(z,$ 
425     functional(f)  $\wedge$  in(x, dom(f))  $\implies$  in(pair(x, z), f)))) )
426      $\wedge$   $\neg$ functional(f)  $\vee$   $\neg$ in(x, dom(f))  $\implies$   $z \equiv \emptyset$ )
427
```

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

```
430
431 val app = DEF (f, x)  $\rightarrow$  The(z,
432     functional(f)  $\wedge$  in(x, dom(f))  $\implies$  in(pair(x, z), f))
433     (appDefinition)
434
```

435 Cantor's Theorem

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

```
438
439 val cantorTheorem = Theorem(  $\neg$ surjective(f, x, powerSet(x)) )
440
```

441 where f and x are schematic set variables, making the sequent implicitly universally quantified.
 442 The proof of Cantor's theorem is about 25 lines of code². Internally, the proof expands to
 443 130 sequent calculus steps.

444 Cantor's theorem is the first theorem formalized in LISA from the list *Formalizing 100*
 445 *Theorems* [44]. While not a difficult theorem, it requires some ground development and
 446 definitions related to set-theoretic functions and relations. The proof itself requires handling
 447 the quantifiers for a contradiction construction and combining lemmas about surjective
 448 functions. Much of the latter is achieved using `Tautology`. It shows that LISA is capable
 449 of non-trivial mathematical development. We expect future developments to become easier
 450 and faster with gradual development of reasoning tools and proofs.

451 6 Related Work

452 A polynomial algorithm for free ortholattices was presented in [17]. A weaker structure with
 453 log linear complexity was first presented in [18]. In LISA we use the ortholattice normal form
 454 for first-order logic formulas. Our $F(\text{OL})^2$ implementation does not aim to be complete for
 455 structures such as quantum monadic algebras that treat extensions of OL (and orthomodular
 456 lattices) to monadic first-order logic [20].

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

² <https://github.com/epfl-lara/lisa/blob/fc37f2a6e879d5f43679a4476c1d6e4685bb14a2/src/main/scala/lisa/mathematics/SetTheory.scala#L1700>

462 The choice of set theory may be considered unusual by some, as Coq [3], Lean [11],
 463 the HOL-family [21] and Isabelle/HOL [42] are based either on type theory or on higher-
 464 order logic. We consider HOL to be one of the most elegant formulations for formal proof
 465 developments. However, set theory is arguably the most widely recognized foundation of
 466 mathematics in the mathematical community, and, despite type-theory based tools having
 467 the advantage of being easier to express formalisms in from the get-go, we believe that
 468 through the development and use of abstracting tactics, a soft-type system and adequate
 469 tools, more familiarity and flexibility in writing proofs can be achieved with a set-theory
 470 based mathematical library. We also hope to provide a test bed to explore direct first-order
 471 foundations as an alternative to the many current systems based on higher-order logic.
 472 Concrete results in Mizar [30], Isabelle/ZF [19], ZF in Isabelle/HOL [5, 35], and TLA+
 473 [8, 36] suggest substantial relevance of set-theoretic foundations. Arguments in favour of set
 474 theoretic foundations have also been discussed by John Harrison [22].

475 Even one more level of indirection than in Isabelle/ZF is present in Isabelle/HOL/TG [5],
 476 which develops the Tarski-Grothendieck extension of ZF inside Isabelle/HOL. Whereas our
 477 system is less flexible and does not currently connect to such a well-developed ecosystem as
 478 Isabelle/HOL has, our hope is that it is conceptually simpler thanks to fewer layers and a
 479 kernel that does not rely on unification.

480 Another modern approach to theorem proving is Lean [11], a proof assistant based on
 481 dependent type theory and inspired in part by Coq. We believe Lean makes significant
 482 improvements over older proof assistant regarding the *Six Virtues*. In particular, it has a
 483 strong focus on programmability, with the new version of Lean [29] even having a compiler
 484 written in its own proof language. While LISA and Lean’s design objectives share similarities,
 485 their strategies and specific choice (foundations, language, interface) are uniquely different.

486 To automate proofs that do not instantiate schematic formulas we hope to make use of
 487 proof generating theorem provers, such as Vampire [27], SPASS [41], E [37], as well as SMT
 488 solvers [1]. Higher-order provers such as Zipperposition [40], LEO III [38], and Satallax [4]
 489 would further increase automation even in the case of axiom schema instantiation.

490 **7 Conclusion**

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

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

507 — References —

- 508 1 Haniel Barbosa, Clark Barrett, Martin Brain, Gereon Kremer, Hanna Lachnitt, Makai Mann,
509 Abdalrhman Mohamed, Mudathir Mohamed, Aina Niemetz, Andres Nötzli, Alex Ozdemir,
510 Mathias Preiner, Andrew Reynolds, Ying Sheng, Cesare Tinelli, and Yoni Zohar. *cvc5: A*
511 *Versatile and Industrial-Strength SMT Solver*. In Dana Fisman and Grigore Rosu, editors,
512 *Tools and Algorithms for the Construction and Analysis of Systems*, volume 13243, pages
513 415–442. Springer International Publishing, Cham, 2022.
514 doi:10.1007/978-3-030-99524-9_24.
- 515 2 Ladislav Beran. *Orthomodular Lattices (An Algebraic Approach)*. Springer Dordrecht, 1985.
516 doi:10.1007/978-94-009-5215-7.
- 517 3 Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development:*
518 *Coq'Art: The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science.
519 An EATCS Series. Springer-Verlag, Berlin Heidelberg, 2004.
520 doi:10.1007/978-3-662-07964-5.
- 521 4 Chad E. Brown. Satallax: An automatic higher-order prover. In Bernhard Gramlich, Dale
522 Miller, and Uli Sattler, editors, *Automated Reasoning*, pages 111–117, Berlin, Heidelberg,
523 2012. Springer Berlin Heidelberg.
- 524 5 Chad E. Brown, C. Kaliszyk, and Karol Pak. Higher-Order Tarski Grothendieck as a
525 Foundation for Formal Proof. In *ITP, 2019*. doi:10.4230/LIPIcs.ITP.2019.9.
- 526 6 Gunter Bruns and John Harding. Algebraic aspects of orthomodular lattices. In Bob Coecke,
527 David Moore, and Alexander Wilce, editors, *Current Research in Operational Quantum Logic:*
528 *Algebras, Categories, Languages*, pages 37–65. Springer Netherlands, Dordrecht, 2000.
529 doi:10.1007/978-94-017-1201-9_2.
- 530 7 Paul J. Cohen. The independence of the continuum hypothesis. *Proceedings of the National*
531 *Academy of Sciences of the United States of America*, 50(6):1143–1148, 1963. URL:
532 <http://www.jstor.org/stable/71858>.
- 533 8 Denis Cousineau, Damien Doligez, Leslie Lamport, Stephan Merz, Daniel Ricketts, and
534 Hernán Vanzetto. TLA+ proofs. In Dimitra Giannakopoulou and Dominique Méry, editors,
535 *FM 2012: Formal Methods*, pages 147–154, Berlin, Heidelberg, 2012. Springer Berlin
536 Heidelberg.
- 537 9 Martin Davis, George Logemann, and Donald Loveland. A machine program for
538 theorem-proving. *Communications of the ACM*, 5(7):394–397, July 1962.
539 doi:10.1145/368273.368557.
- 540 10 Nicolaas Govert De Bruijn. Lambda calculus notation with nameless dummies, a tool for
541 automatic formula manipulation, with application to the church-rosser theorem. In
542 *Indagationes Mathematicae (Proceedings)*, volume 75, pages 381–392. Elsevier, 1972.
- 543 11 Leonardo de Moura, Soonho Kong, Jeremy Avigad, Floris van Doorn, and Jakob von Raumer.
544 The Lean Theorem Prover (System Description). In Amy P. Felty and Aart Middeldorp,
545 editors, *Automated Deduction - CADE-25*, volume 9195, pages 378–388. Springer
546 International Publishing, Cham, 2015. doi:10.1007/978-3-319-21401-6_26.
- 547 12 David Delahaye. A tactic language for the system coq. In *LPAR*, volume 1955, pages 85–95.
548 Springer, 2000.
- 549 13 Harald Ganzinger, Christoph Meyer, and Christoph Weidenbach. Soft typing for ordered
550 resolution. In William McCune, editor, *Automated Deduction - CADE-14, 14th International*
551 *Conference on Automated Deduction, Townsville, North Queensland, Australia, July 13-17,*
552 *1997, Proceedings*, volume 1249 of *Lecture Notes in Computer Science*, pages 321–335.
553 Springer, 1997. doi:10.1007/3-540-63104-6_32.
- 554 14 G. Gentzen. Untersuchungen über das logische schließen i. *Mathematische Zeitschrift*,
555 39:176–210, 1935. URL: <http://eudml.org/doc/168546>.
- 556 15 Michael J. C. Gordon, Robin Milner, Christopher P. Wadsworth, and P. Ted Christopher.
557 Edinburgh LCF: a mechanized logic of computation. *Lecture Notes in Computer Science*,
558 1978.

- 559 16 Simon Guilloud. LISA Reference Manual. EPFL-LARA, February 2023.
- 560 17 Simon Guilloud, Mario Bucev, Dragana Milovancevic, and Viktor Kunčák. Formula
561 normalizations in verification. Technical Report 297701, EPFL, 2023. URL:
562 <http://infoscience.epfl.ch/record/297701>.
- 563 18 Simon Guilloud and Viktor Kunčák, editors. *Equivalence Checking for Orthocomplemented*
564 *Bisemilattices in Log-Linear Time*. Springer, 2022. doi:10.48550/arXiv.2110.03315.
- 565 19 Emmanuel Gunther, Miguel Pagano, Pedro Sánchez Terraf, and Matias Steinberg. The
566 independence of the continuum hypothesis in isabelle/zf. *Archive of Formal Proofs*, March
567 2022. https://isa-afp.org/entries/Independence_CH.html, Formal proof
568 development.
- 569 20 J Harding. Quantum monadic algebras. *Journal of Physics A: Mathematical and Theoretical*,
570 55(39):394001, sep 2022. doi:10.1088/1751-8121/ac845b.
- 571 21 John Harrison. HOL Light: An Overview. In Stefan Berghofer, Tobias Nipkow, Christian
572 Urban, and Makarius Wenzel, editors, *Theorem Proving in Higher Order Logics*, volume 5674,
573 pages 60–66. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009.
574 doi:10.1007/978-3-642-03359-9_4.
- 575 22 John Harrison. Let’s make set theory great again! In *Axiomatic Set Theory*, page 46, Aussois,
576 2018.
- 577 23 CNRS Inria and contributors. Ltac2 — Coq 8.16.1 documentation. URL:
578 <https://coq.inria.fr/refman/proof-engine/ltac2.html>.
- 579 24 Himanshu Jain, Constantinos Bartzis, and Edmund Clarke. Satisfiability checking of
580 non-clausal formulas using general matings. In Armin Biere and Carla P. Gomes, editors,
581 *Theory and Applications of Satisfiability Testing - SAT 2006*, pages 75–89, Berlin, Heidelberg,
582 2006. Springer Berlin Heidelberg. doi:https://doi.org/10.1007/11814948_10.
- 583 25 Thomas Jech. *Set theory: The third millennium edition, revised and expanded*. Springer, 2003.
- 584 26 Deepak Kapur and Paliath Narendran. Complexity of unification problems with
585 associative-commutative operators. *J. Autom. Reason.*, 9(2):261–288, 1992.
586 doi:10.1007/BF00245463.
- 587 27 Laura Kovács and Andrei Voronkov. First-Order Theorem Proving and Vampire. In Natasha
588 Sharygina and Helmut Veith, editors, *Computer Aided Verification*, Lecture Notes in
589 Computer Science, pages 1–35, Berlin, Heidelberg, 2013. Springer.
590 doi:10.1007/978-3-642-39799-8_1.
- 591 28 Norman Megill. Metamath. *The Seventeen Provers of the World: Foreword by Dana S. Scott*,
592 pages 88–95, 2006.
- 593 29 Leonardo de Moura and Sebastian Ullrich. The lean 4 theorem prover and programming
594 language. In André Platzer and Geoff Sutcliffe, editors, *Automated Deduction – CADE 28*,
595 pages 625–635, Cham, 2021. Springer International Publishing.
- 596 30 Adam Naumowicz and Artur Kornilowicz. A Brief Overview of Mizar. In *Proceedings of the*
597 *22nd International Conference on Theorem Proving in Higher Order Logics*, volume 5674,
598 pages 67–72, August 2009. doi:10.1007/978-3-642-03359-9_5.
- 599 31 Martin Odersky, Aggelos Biboudis, Fengyun Liu, and Olivier Blanvillain. Foundations of
600 implicit function types. Technical report, EPFL, 2017. URL:
601 <http://infoscience.epfl.ch/record/229203>.
- 602 32 Lawrence C. Paulson. Isabelle: The next 700 theorem provers. *CoRR*, cs.LO/9301106, 1993.
603 URL: <https://arxiv.org/abs/cs/9301106>.
- 604 33 Lawrence C. Paulson. A generic tableau prover and its integration with Isabelle. *JUCS -*
605 *Journal of Universal Computer Science*, 5(3):73–87, 1999.
606 arXiv:<https://doi.org/10.3217/jucs-005-03-0073>,
607 doi:10.3217/jucs-005-03-0073.
- 608 34 Lawrence C Paulson. Isabelle’s logics: FOL and ZF, 2013.

- 609 35 Lawrence C. Paulson. Zermelo Fraenkel set theory in higher-order logic. *Archive of Formal*
610 *Proofs*, October 2019. https://isa-afp.org/entries/ZFC_in_HOL.html, Formal
611 proof development.
- 612 36 TLA Proof System Project. TLA+ proof system.
613 <https://tla.msr-inria.inria.fr/tlapps/content/Home.html>.
- 614 37 Stephan Schulz. System Description: E 1.8. In Ken McMillan, Aart Middeldorp, and Andrei
615 Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning*, Lecture
616 Notes in Computer Science, pages 735–743, Berlin, Heidelberg, 2013. Springer.
617 doi:10.1007/978-3-642-45221-5_49.
- 618 38 Alexander Steen. Leo-iii 1.7, July 2022. doi:10.5281/zenodo.7650205.
- 619 39 Christian Urban. The Isabelle Cookbook. URL: [https://web.cs.wpi.edu/~dd/](https://web.cs.wpi.edu/~dd/resources_isabelle/isabelle_programming.urban.pdf)
620 [resources_isabelle/isabelle_programming.urban.pdf](https://web.cs.wpi.edu/~dd/resources_isabelle/isabelle_programming.urban.pdf).
- 621 40 Petar Vukmirović, Alexander Bentkamp, Jasmin Blanchette, Simon Cruanes, Visa Nummelin,
622 and Sophie Tourret. Making Higher-Order Superposition Work. In André Platzer and Geoff
623 Sutcliffe, editors, *Automated Deduction – CADE 28*, Lecture Notes in Computer Science,
624 pages 415–432, Cham, 2021. Springer International Publishing.
625 doi:10.1007/978-3-030-79876-5_24.
- 626 41 Christoph Weidenbach, Dilyana Dimova, Arnaud Fietzke, Rohit Kumar, Martin Suda, and
627 Patrick Wischniewski. SPASS Version 3.5. In Renate A. Schmidt, editor, *Automated*
628 *Deduction – CADE-22*, Lecture Notes in Computer Science, pages 140–145, Berlin,
629 Heidelberg, 2009. Springer. doi:10.1007/978-3-642-02959-2_10.
- 630 42 Makarius Wenzel, Lawrence C. Paulson, and Tobias Nipkow. The Isabelle Framework. In
631 Otmane Ait Mohamed, César Muñoz, and Sofiène Tahar, editors, *Theorem Proving in Higher*
632 *Order Logics*, Lecture Notes in Computer Science, pages 33–38, Berlin, Heidelberg, 2008.
633 Springer. doi:10.1007/978-3-540-71067-7_7.
- 634 43 Benjamin Werner. Sets in types, types in sets. In Gerhard Goos, Juris Hartmanis, Jan van
635 Leeuwen, Martín Abadi, and Takayasu Ito, editors, *Theoretical Aspects of Computer Software*,
636 volume 1281, pages 530–546. Springer Berlin Heidelberg, Berlin, Heidelberg, 1997.
637 doi:10.1007/BFb0014566.
- 638 44 Freek Wiedijk. Formalizing 100 theorems. <https://www.cs.ru.nl/~freek/100/>.