LISA: Towards a Foundational Theorem Prover

Simon Guilloud , Florian Cassayre, Viktor Kunčak

EPFL IC LARA, Station 14, CH-1015 Lausanne, Switzerland
{Simon.Guilloud,Viktor.Kuncak}@epfl.ch

We present the foundations and initial implementation of a new interactive theorem prover, named LISA. In a slight contrast to most popular type-theoretic frameworks, and much like Mizar [9], LISA aims to use classical mainstream foundations of mathematics, taking a hint, among others from the talk of John Harrison in this very venue in 2018 [12]. LISA uses (single-sorted) first order logic (with schematic variables) as the syntactic framework and set theory axiom schemas as the semantic framework. On top of these foundations we can construct numbers and other mathematical theories and models of computation without introducing new axioms. As the target use of LISA we envision mathematical statements as well as formal proofs of computer programs and systems, possibly with probabilistic and distributed behavior. For automation in LISA we expect to employ newly developed algorithms for equivalence checking of formulas and proofs [4], existing high-performance superposition-based theorem provers such as Vampire [7], SPASS [13], E [10] and Zipperposition, as well as SMT solvers such as Z3 [8], CVC5 [**?**], and veriT [11], and OpenSMT [2]. An important aim of LISA is interoperability with other proof assistants. We hope that the design of LISA with small, fresh code base, simple foundation and explicit proof objects will encourage building bridges with other tools. We also expect that the system will serve as a good vehicle to explore machine-learning guided superposition proof search, with learned heuristics complementing hand-tuned ones. We also plan to explore high-level translation of proofs from other systems into LISA, inspired by the success of deep neural networks in natural language translation.

Development LISA is separated into a trusted logical core called the kernel that we keep as small as possible, and a front part adding arbitrarily complex or powerful layers of abstractions and features for the user, but which need not be trusted. Our language of choice for LISA is Scala (version 3). On one hand, Scala offers very powerful syntax and features, such as dependent types, string interpolation and implicits, which allows (for example) to use Scala compiler to check that terms are well formed, or to define very nice syntax for proofs or objects we want to manipulate. On the other hand, the kernel is developed only with a restricted subset of the Scala language, paving the door for future verification of the kernel using program verification tools such as Stainless [5].

We strived to keep the core of LISA as close as possible to well-known and uncontroversial mathematical theory, but nonetheless includes some improvements, in particular relating to space and time complexity. Hence, our base language is first order logic, to which we add schematic functions and predicates. Those schematic symbols give some flavour of second-order logic, the possibility to define a single syntactic object representing an axiom schema or theorem schema. Our experience already suggests that schematic variables make many further developpements and proofs simpler, but do not increase the proof strength of FOL.

LISA kernel's proof checker is based on an implementation of Gentzen's sequent calculus, with the addition of some deduced proof steps such as the substitution of equals for equals and the possibility to enclose subproofs to reduce the length of proofs.

Proofs in LISA are represented as linearized directed acyclic graphs (i.e. lists of proof steps), which can verified by the proof checker. Proofs can either exists in isolation, or be part of *theories*. Theories in LISA form the LCF part of the kernel. They use the proof checker to verify theorems, and to then accept such theorems as assumptions of future proofs without needing rechecking. Theories also enable introduction of (set theoretic) axioms and definitional extensions of new symbols.

Equivalence Checker A unique feature of our kernel is the inclusion of a quasilinear-time *sufficient-equivalence checker* for FOL formulas that takes into account associativity, commutativity, and other laws and has a clear algebraic completeness characterization [4]. The aim of implementing such equivalence checker in the kernel is to shorten the proofs (by a constant factor), making the system more efficient to use by humans and tactics alike.

The equivalence checker takes into account properties of FOL such as alpha equivalence and symmetry of equality, but most of its complexity take place at the level of propositional logic. Since equivalence of propositional formulas is coNP complete, we can only aim for an approximation of it. Nonetheless, it is important to note that our algorithm is not an arbitrary heuristic; it is a complete decision procedure for a well-defined subalgebra of boolean algebra called orthocomplemented bisemilattice, which is essentially Boolean algebra without the distributivity law. We expect that this aspect helps make the equivalence checker predictable. Moreover, it runs in quasilinear time, meaning it can replace syntactic equality checking everywhere in the proof checking procedure with only a logarithmic cost. Equivalence checking algorithm is implemented using a combination of techniques for tree isomorphism with memoization and term rewriting. Our anecodotal experience suggests that manual construction of proofs greatly benefited from the implementation of this formula equivalence.

High Level Proofs We also succesfully implemented a semi-interactive interface to the logical kernel allowing forward and backward reasoning, creation of new tactics and combination of existing ones, high syntax insurance through Scala type checking, higher order matching and more. Works is still ongoing, but it demonstrates the feasibility of designing a powerfull interface for the system.

Why set theory? Even though set theory is by far the most studied, accepted and well-known foundation of mathematics, this choice is uncommon among interactive theorem provers and the Computer Science community in general. Indeed, most modern ITPs such as Coq [1], Lean [3], Isabelle [14] or the HOL-family [6] are based either on some form of Type Theory or on Higher Order Logic. While those theory have the advantage to look closer from the start to the mathematical formalism, we believe there are several reasons to prefer set theory. It may need higher upfront work, but we strongly believe that through the use of powerful tactics and a soft type system (where types are represented by set or class membership with meta-information for polymorphism) apposed on top of set theory, arbitrarily familiarity in the writing of proofs can be achieved with set theory, with ultimately greater flexibility. More arguments in favour of set theoretic foundations have been proposed by John Harrison [12].

Verification of Programs One of LISA's goals is to contribute to verificaton of reliability of our computing infrastructure. Medium term, we aim for two different paths for using LISA inside program verifiers, such as Stainless, in which it can be cumbersome to reason about quantified propositions. On the one hand, we hope to develop semantics of transition systems and programming languages and use such semantics to provide high-confidence proofs about behavior of programs and embedded systems.

Future work and conclusion LISA is in early stage of development, which has focused on proof checking for FOL with equality and schematic variables, as well as the support for theories and LCF-style tracking of theoremhood. Most of the development dependent on set theoretic axiom schemas remains in the future. Medium term goals include further development of abstraction layers and proof tactics, development of a core library of results, implementation of the program verification side of LISA and more exploratory work such as the tradeoff between , or the extension of our equivalence checker to orthocomplemented lattices. We also had some exploratory work on the interoperability of proofs, and we plan to continue to work on that aspect.

References

[1] 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.

[2] Roberto Bruttomesso, Edgar Pek, Natasha Sharygina, and Aliaksei Tsitovich. The OpenSMT Solver. In David Hutchison, Takeo Kanade, Josef Kittler, Jon M. Kleinberg, Friedemann Mattern, John C. Mitchell, Moni Naor, Oscar Nierstrasz, C. Pandu Rangan, Bernhard Steffen, Madhu Sudan, Demetri Terzopoulos, Doug Tygar, Moshe Y. Vardi, Gerhard Weikum, Javier Esparza, and Rupak Majumdar, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 6015, pages 150–153. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010.

[3] 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.

[4] Simon Guilloud and Viktor Kuncak, editors. *Equivalence Checking for Orthocomplemented Bisemilattices in Log-Linear Time*. Springer, 2022.

[5] Jad Hamza, Nicolas Voirol, and Viktor Kunčak. System FR: Formalized foundations for the Stainless verifier. *Proc. ACM Program. Lang*, 3, November 2019.

[6] 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.

[7] 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.

[8] Stephan Merz and Hernán Vanzetto. Automatic Verification of TLA + Proof Obligations with SMT Solvers. In Nikolaj Bjørner and Andrei Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning*, Lecture Notes in Computer Science, pages 289–303, Berlin, Heidelberg, 2012. Springer.

[9] Adam Naumowicz and Artur Kornilowicz. A Brief Overview of Mizar. volume 5674, pages 67–72, August 2009.

[10] 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.

[11] Hans-Jörg Schurr, Mathias Fleury, and Martin Desharnais. Reliable Reconstruction of Fine-grained Proofs in a Proof Assistant. In André Platzer and Geoff Sutcliffe, editors, *Automated Deduction – CADE 28*, Lecture Notes in Computer Science, pages 450–467, Cham, 2021. Springer International Publishing.

[12] John HarrisonAmazon Web Services. Let's make set theory great again! *axiomatic set theory*, page 46.

[13] 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.

[14] 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.