LISA Tool Integration and Education Plans

Sankalp Gambhir[1], Simon Guilloud[1], Dragana Milovančević[1], Philipp Rümmer[2], and Viktor Kunčak[1]

[1]EPFL IC LARA, Station 14, CH-1015 Lausanne, Switzerland
[2]University of Regensburg, Faculty of Informatics and Data Science, 93040 Regensburg, Germany
{Sankalp.Gambhir,Simon.Guilloud,Dragana.Milovancevic,Viktor.Kuncak}@epfl.ch,
Philipp.Ruemmer@ur.de

LISA is a theorem prover implemented in Scala based on Tarski-Grothendieck (TG) set theory (foundations similar to those of Mizar [23] and higher-order encoding of TG in Isabelle/HOL [7]). Since our first announcement [11], we have developed a domain-specific proof language and a core library [12,13], including transfinite recursion. LISA proofs are Scala programs whose execution generates sequent calculus proof sequences checked by the LISA kernel [12, 13]. Proof construction can use all features of Scala to prove theorems from previously proven theorems, axioms, and conservative definitions. We present applications and planned developments of LISA. This includes an embedding of HOL to enable use of HOL provers and libraries, reasoning about programs using the Eldarica Horn clause solver and the Stainless Scala verifier, as well as using LISA for checked proofs in functional programming courses.

Embeddings of HOL in set theory, using HOL solvers We plan to develop an embedding of Higher-Order Logic and type theory in LISA's set theory. In set theory, a function with domain and range sets A and B is represented as a subset of their Cartesian product $A \times B$ encoding the graph of the function [18, 20]. Thus, function spaces $A \to B$ are sets, denoted B^A. This yields a natural embedding of classical HOL functions. The type of HOL individuals is interpreted as an infinite set, and Boolean type becomes the ordinal $2 = \{0, 1\}$. Predicates over a set A become the characteristic functions 2^A, and logical connectives functions taking elements of 2 and returning an element of 2. The $\forall$ quantifier reduces to equality, $\forall x \in A.P(x) \;:=\; ((\lambda x \in A.P(x)) =_T (\lambda x \in A.\mathsf{True}))$ with $=_T$ the characteristic function of $\{(x, x) \mid x \in T\}$ over $T \times T$, here used with $T = 2^A$.

We plan to use this encoding in two ways. **First**, we will use automated higher-order theorem provers to prove statements whose subterms have bounded domains. The provers we are considering include LEO-III [32] (also implemented in Scala), Lash [8], Zipperposition [4], and new versions of E [33]. Because the deduction rules of HOL are provable within set theory, it is then possible to recover the proof within LISA. **Second**, we will explore importing proofs and theories from HOL4 [31], HOL Light [15], and Isabelle/HOL [25] into LISA, which will help in bootstrapping our library.

As in HOL, the restriction of set theory statements to functions and predicates whose domains are proper sets imposes a type structure of simply typed lambda calculus, for which there are efficient type checking and inference algorithms. This will provide the basis for a soft-type system over set theory. Further down the line, we will explore the encoding of dependent types using dependent function spaces (products) in set theory. Combined with induction principles (which we have recently derived in LISA using conventional transfinite induction), and universes (whose existence is implied by large cardinals in LISA's Tarski-Grothendieck set theory), we expect [34] to be in a position to enable interoperability with type theoretic libraries such as Lean's standard library [9] and, for example, Coqtail [2] for Coq.

Combining LISA, Stainless and Eldarica to reason about programs Verifying safety and termination properties of (higher-order) programs has been a long-standing problem, and is one of the intended application domains of LISA. There remain significant challenges in making full functional verification of these programs tractable. One approach for solving this problem is to use program verifiers, such as the Stainless tool developed in our group [1, 14]. Stainless allows a programmer to work in Scala and specify functional contracts within the source language. Stainless relies on converting the code and contracts to SMT formulas and passing them to external solvers, providing support for a large subset of functional and imperative Scala code. Stainless is supported by formally verified foundations [14], but has several limitations. First, the implementation relies on external SMT solvers without checking their result, which can be problematic [26] and limits the trustworthiness. Second, Stainless does not automatically infer inductive invariants for safety verification. Third, when an automated proof attempt fails, the abilities to provide proof hints are limited. To address all these limitations, we plan to convert Stainless programs to Constrained Horn Clauses (CHCs) and construct end-to-end formally checked proofs for properties of CHCs. CHCs naturally express program flow, and have seen extensive use in program verification [6, 29]. We will define the semantics of CHCs in LISA and prove generic properties about such systems, which may be used during automated and manual verification. We will use Scala as a user-friendly language for specification of programs and algorithms, and use Stainless to generate CHCs to be verified. We view the correctness of the transformation from a complex language to CHCs as a separate verified compilation problem, similar to that of CakeML [19], [35]. We intend to use the Eldarica CHC solver [16, 30] based on Princess [28] to aid solving of CHCs.

The resulting proofs (both for Horn clause solving of Eldarica and for constraint reasoning of Princess) will be checked by the LISA kernel. In cases where automated verification fails, LISA will enable users to construct semi-manual proofs about CHCs, using tactics and the proof DSL. We believe these developments will greatly increase the automation and the scope of applicability of reasoning about programs.

LISA's potential in education Automated theorem proving is increasingly finding its way in education, both for mathematics and computer science programs [3]. Proof assistants are used not only in graduate courses [17, 24, 27] but also in undergraduate courses [21] and high schools [5, 10]. With LISA, we aim to go one step further, proposing the use of proof assistants for introductory *programming* courses. We have successfully developed methods to automatically and formally verify correctness of student code with respect to a given reference solution [22] using Stainless. In functional programming courses, students typically not only code, but also have to prove some property of a given implementation, for example that a tail-recursive variant of a function is equivalent to its non-tail-recursive variant. This is typically done with the substitution semantics of functional programs, so that such proofs only need instantiation of free parameters and equational reasoning. This makes them feasible for automated grading with guaranteed correctness.

We believe that students can write such proofs in LISA without deep knowledge of proof assistants, because LISA's high-level interface and a DSL provide an intuitive and programmer-friendly environment. To illustrate the key features, consider an example exercise from a midterm exam of a past edition of EPFL's Functional Programming course. The goal of this exercise is to prove that the methods `map` and `mapTr` (a tail-recursive variant of `map`) on singly-linked lists are equivalent. The following example shows how the first intermediate lemma that students have to prove would look like in LISA.

```
val AccOutNil = Theorem( Nil.mapTr(f, (x :: xs)) === (x :: Nil.mapTr(f, xs)) ) {
  have    ( Nil.mapTr(f, (x :: xs)) === (x :: xs) )
                                      by Apply(mapTr.NilCase of (acc → (x :: xs)))
  thenHave( Nil.mapTr(f, (x :: xs)) === (x :: Nil.mapTr(f, xs)) )
                                      by Apply(mapTr.NilCase of (acc → xs)) }
```

Note that the terms in formulas above appear syntactically identical to their Scala program counterparts. Using Stainless and Scala 3 multi-stage programming, these proofs can apply to executable Scala programs. We expect further synergies, such as using Scala's pattern matching on algebraic data types to write proofs and definitions by case analysis.

References

[1] Stainless - a verification framework for a subset of the Scala programming language. https://github.com/epfl-lara/stainless/.

[2] Guillaume Allais, Sylvain Dailler, Hugo Férée, Jean-Marie Madiot, Pierre-Marie Pédrot, and Amaury Pouly. Coq-community/coqtail-math: Coqtail is a library of mathematical theorems and tools proved inside the Coq proof assistant. Results range mostly from arithmetic to real and complex analysis. [maintainer=@jmadiot]. https://github.com/coq-community/coqtail-math.

[3] Evmorfia Bartzia, Antoine Meyer, and Julien Narboux. Proof assistants for undergraduate mathematics and computer science education: elements of a priori analysis. In María Trigueros, editor, *INDRUM 2022: Fourth conference of the International Network for Didactic Research in University Mathematics*, Hanovre, Germany, October 2022. Reinhard Hochmuth.

[4] Alexander Bentkamp, Jasmin Blanchette, Sophie Tourret, and Petar Vukmirovic. Superposition for higher-order logic. *J. Autom. Reason.*, 67(1):10, 2023.

[5] Yves Bertot, Frédérique Guilhot, and Loic Pottier. Visualizing Geometrical Statements with GeoView. *Electr. Notes Theor. Comput. Sci.*, 103:49–65, 11 2004.

[6] Nikolaj S. Bjørner, Arie Gurfinkel, Kenneth L. McMillan, and Andrey Rybalchenko. Horn clause solvers for program verification. In Lev D. Beklemishev, Andreas Blass, Nachum Dershowitz, Bernd Finkbeiner, and Wolfram Schulte, editors, *Fields of Logic and Computation II - Essays Dedicated to Yuri Gurevich on the Occasion of His 75th Birthday*, volume 9300 of *Lecture Notes in Computer Science*, pages 24–51. Springer, 2015.

[7] Chad E. Brown, C. Kaliszyk, and Karol Pak. Higher-Order Tarski Grothendieck as a Foundation for Formal Proof. In *ITP*, 2019.

[8] Chad E. Brown and Cezary Kaliszyk. Lash 1.0 (system description). In Jasmin Blanchette, Laura Kovács, and Dirk Pattinson, editors, *Automated Reasoning - 11th International Joint Conference, IJCAR 2022, Haifa, Israel, August 8-10, 2022, Proceedings*, volume 13385 of *Lecture Notes in Computer Science*, pages 350–358. Springer, 2022.

[9] 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*, Lecture Notes in Computer Science, pages 378–388, Cham, 2015. Springer International Publishing.

[10] Frédérique Guilhot. Formalisation en Coq et visualisation d'un cours de géométrie pour le lycée. *Technique et Science Informatiques*, 24:1113–1138, 11 2005.

[11] Simon Guilloud, Florian Cassayre, and Viktor Kunčak. LISA: Towards a foundational theorem prover. 7th Conference on Artificial Intelligence and Theorem Proving, AITP 2022, September 4-9, 2022, Aussois and online, France, http://aitp-conference.org/2022/abstract/AITP_2022_paper_23.pdf.

[12] Simon Guilloud, Sankalp Gambhir, and Viktor Kuncak. LISA – A Modern Proof System. In *To appear in ITP, Conference on Interactive Theorem Proving*, 2023.

[13] Simon Guilloud, Sankalp Gambhir, and Viktor Kuncak. Lisa – a modern proof system (preprint). http://infoscience.epfl.ch/record/300562, 2023.

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

[15] John Harrison. The HOL Light manual (1.1). page 116.

[16] Hossein Hojjat and Philipp Rümmer. The eldarica horn solver. In *2018 Formal Methods in Computer Aided Design (FMCAD)*, pages 1–7. IEEE, 2018.

[17] Frederik Jacobsen and Jørgen Villadsen. On Exams with the Isabelle Proof Assistant. *Electronic Proceedings in Theoretical Computer Science*, 375:63–76, 03 2023.

[18] Thomas J. Jech. *Set Theory*. Number 79 in Pure and Applied Mathematics, a Series of Monographs and Textbooks. Academic Press, New York, 1978.

[19] Ramana Kumar, Magnus O. Myreen, Michael Norrish, and Scott Owens. Cakeml: a verified implementation

of ML. In Suresh Jagannathan and Peter Sewell, editors, *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14, San Diego, CA, USA, January 20-21, 2014*, pages 179–192. ACM, 2014.

[20] Kenneth Kunen. *Set Theory An Introduction To Independence Proofs*. North Holland, Amsterdam Heidelberg, reprint edition edition, December 1983.

[21] Hendriks Maxim, Cezary Kaliszyk, Femke van Raamsdonk, and Freek Wiedijk. Teaching logic using a state-of-art proof assistant. *Acta Didactica Napocensia*, 3, 06 2010.

[22] Dragana Milovančević and Viktor Kunčak. Proving and Disproving Equivalence of Functional Programming Assignments. In *ACM SIGPLAN Conf. Programming Language Design and Implementation (PLDI)*, 2023.

[23] 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, 2009.

[24] Tobias Nipkow. Teaching Semantics with a Proof Assistant: No More LSD Trip Proofs. In Viktor Kuncak and Andrey Rybalchenko, editors, *Verification, Model Checking, and Abstract Interpretation*, pages 24–38, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[25] Tobias Nipkow and Gerwin Klein. *Concrete Semantics - With Isabelle/HOL*. Springer, 2014.

[26] Jiwon Park, Dominik Winterer, Chengyu Zhang, and Zhendong Su. Generative type-aware mutation for testing smt solvers. *Proc. ACM Program. Lang.*, 5(OOPSLA), oct 2021.

[27] Benjamin Pierce. Lambda, the Ultimate TA Using a Proof Assistant to Teach Programming Language Foundations. pages 121–122, 08 2009.

[28] Philipp Rümmer. A constraint sequent calculus for first-order logic with linear integer arithmetic. In *Proceedings, 15th International Conference on Logic for Programming, Artificial Intelligence and Reasoning*, volume 5330 of *LNCS*, pages 274–289. Springer, 2008.

[29] Philipp Rümmer, Hossein Hojjat, and Viktor Kuncak. Classifying and solving horn clauses for verification. In *Fifth Working Conference on Verified Software: Theories, Tools and Experiments*, 2013.

[30] Philipp Rümmer, Hossein Hojjat, and Viktor Kuncak. Disjunctive interpolants for horn-clause verification. In *Computer Aided Verification (CAV)*, 2013.

[31] Konrad Slind and Michael Norrish. A Brief Overview of HOL4. pages 28–32, August 2008.

[32] Alexander Steen and Christoph Benzmüller. The Higher-Order Prover Leo-III (Extended Version), April 2018.

[33] Petar Vukmirovic, Jasmin Blanchette, and Stephan Schulz. Extending a high-performance prover to higher-order logic. In Sriram Sankaranarayanan and Natasha Sharygina, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 29th International Conference, TACAS 2023, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Paris, France, April 22-27, 2023, Proceedings, Part II*, volume 13994 of *Lecture Notes in Computer Science*, pages 111–129. Springer, 2023.

[34] 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.

[35] Johannes Åman Pohjola, Henrik Rostedt, and Magnus O. Myreen. Characteristic Formulae for Liveness Properties of Non-Terminating CakeML Programs. In *10th International Conference on Interactive Theorem Proving (ITP 2019)*, Leibniz International Proceedings in Informatics (LIPIcs), pages 32:1–32:19, 2019.