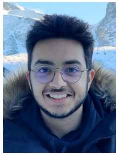
LISA

First-Order Interactive Proof Assistant


Sankalp Gambhir[1]

Simon Guilloud[1]

Viktor Kunčak[1]

Dragana Milovančević[1]

Philipp Rümmer[2]

[1] Laboratory for Automated Reasoning and Analysis, EPFL, Switzerland

[2] Faculty of Informatics and Data Science, University of Regensburg, Germany

LISA: A Proof Framework in Scala

AI for theorem proving needs libraries and frameworks to integrate and manipulate formal knowledge.

We hope LISA framework can be useful because of its

- **foundations** on (TG) set theory — can semantically embed other foundations
- **design** with simple proof kernel (schematic FOL)
- **implementation** in Scala (well-supported ecosystem, DSLs, libraries for distributed computing)

LISA of the Present

Introduction

LISA is a proof assistant in continuous development.

- Based on FOL

Introduction

LISA is a proof assistant in continuous development.

- Based on FOL
- Small Kernel, hybrid LCF-style

Introduction

LISA is a proof assistant in continuous development.

- Based on FOL
- Small Kernel, hybrid LCF-style
- High programmability and integrability focus

Introduction

LISA is a proof assistant in continuous development.

- Based on FOL
- Small Kernel, hybrid LCF-style
- High programmability and integrability focus
- Written in Scala as an extensible library

The Kernel

LISA uses First Order Logic as its foundational language, and extends it with schematic function and predicate symbols.

$$'P(0) \land \forall x. ('P(x) \implies 'P(x+1)) \vdash \forall x. 'P(x)$$

The Kernel

LISA uses First Order Logic as its foundational language, and extends it with schematic function and predicate symbols.

$$'P(0) \land \forall x. ('P(x) \implies 'P(x+1)) \vdash \forall x. 'P(x)$$

- Theory-agnostic kernel
- Uses Set Theory for mathematical library

The Sequent Calculus LK

LISA uses a variation of Sequent Calculus LK.

- Sequents $\Gamma \vdash \Delta$, with Γ and Δ sets of formulas

The Sequent Calculus LK

LISA uses a variation of Sequent Calculus LK.

- Sequents $\Gamma \vdash \Delta$, with Γ and Δ sets of formulas
- Introduction rule for each logical symbol on each side + Cut, Weakening

The Sequent Calculus LK

LISA uses a variation of Sequent Calculus LK.

- Sequents $\Gamma \vdash \Delta$, with Γ and Δ sets of formulas
- Introduction rule for each logical symbol on each side + Cut, Weakening

The Sequent Calculus LK

LISA uses a variation of Sequent Calculus LK.

- Sequents $\Gamma \vdash \Delta$, with Γ and Δ sets of formulas
- Introduction rule for each logical symbol on each side + Cut, Weakening

$$\frac{\Gamma \vdash \phi[s/'x], \Delta}{\Gamma, s = t, \vdash \phi[t/'x], \Delta} \quad \texttt{SubstEq}$$

$$\frac{\Gamma \vdash \Delta}{\Gamma[\psi(\vec{v})/'P] \vdash \Delta[\psi(\vec{v})/'P]} \quad \texttt{InstPredSchema}$$

But strictly formal proofs can be excessively tedious for humans to write

$$\dfrac{\vdash a \wedge (b \vee c) \qquad a \wedge (c \vee b) \vdash d}{\vdash d}\ \text{Cut}$$

But strictly formal proofs can be excessively tedious for humans to write

$$\dfrac{\vdash a \wedge (b \vee c) \qquad a \wedge (c \vee b) \vdash d}{\vdash d}\ \text{Cut}$$

Doesn’t work, but to swap b and c...

$$\dfrac{\vdash a \wedge (b \vee c) \qquad \dfrac{\dfrac{\dfrac{}{a \vdash a}\ \text{Hypothesis}}{a \wedge (b \vee c) \vdash a}\ \text{LeftAnd} \qquad \dfrac{\dfrac{\dfrac{\dfrac{}{b \vdash b}\ \text{Hypothesis}}{b \vdash c \vee b}\ \text{RightOr} \qquad \dfrac{\dfrac{}{c \vdash c}\ \text{Hypothesis}}{c \vdash c \vee b}\ \text{RightOr}}{b \vee c \vdash c \vee b}\ \text{LeftOr}}{a \wedge (b \vee c) \vdash c \vee b}\ \text{LeftAnd}}{a \wedge (b \vee c) \vdash a \wedge (c \vee b)}\ \text{RightAnd}}{\dfrac{\vdash a \wedge (c \vee b) \qquad a \wedge (c \vee b) \vdash d}{\vdash d}\ \text{Cut}}\ \text{Cut}$$

Equivalence checking: Ortholattices

$a \wedge (b \vee c)$ $(c \vee b) \wedge a$

Equivalence checking: Ortholattices

$$a \land (b \lor c) \qquad \equiv \qquad (c \lor b) \land a$$

Equivalence checking: Ortholattices

$$a \land (b \lor c) \quad \sim_{OL} \quad (c \lor b) \land a$$

Equivalence checking: Ortholattices

$$a \wedge (b \vee c) \quad \sim_{OL} \quad (c \vee b) \wedge a$$

Ortholattices:

Distributivity: $a \wedge (b \vee c) = (a \wedge b) \vee (a \wedge c)$

Equivalence checking: Ortholattices

$$a \wedge (b \vee c) \quad \sim_{OL} \quad (c \vee b) \wedge a$$

Ortholattices:

Distributivity: $a \wedge (b \vee c) = (a \wedge b) \vee (a \wedge c)$ ✗

Absorption: $a \wedge (a \vee c) = a$ ✓

Equivalence checking: Ortholattices

$$a \wedge (b \vee c) \quad \sim_{OL} \quad (c \vee b) \wedge a$$

Ortholattices:

Distributivity: $a \wedge (b \vee c) = (a \wedge b) \vee (a \wedge c)$ ✗

Absorption: $a \wedge (a \vee c) = a$ ✓

- Satisfies all other Boolean laws

Equivalence checking: Ortholattices

$$a \land (b \lor c) \quad \sim_{OL} \quad (c \lor b) \land a$$

Ortholattices:

Distributivity: $a \land (b \lor c) = (a \land b) \lor (a \land c)$ ✗

Absorption: $a \land (a \lor c) = a$ ✓

- Satisfies all other Boolean laws
- $\implies$ sound approximation of Boolean equivalence

Equivalence checking: Ortholattices

$$a \land (b \lor c) \quad \sim_{OL} \quad (c \lor b) \land a$$

Ortholattices:

Distributivity: $a \land (b \lor c) = (a \land b) \lor (a \land c)$ ✗

Absorption: $a \land (a \lor c) = a$ ✓

- Satisfies all other Boolean laws
- $\implies$ sound approximation of Boolean equivalence
- Algorithm for quadratic-time equivalence and implication checking

Equivalence checking: Ortholattices

$$a \land (b \lor c) \quad \sim_{OL} \quad (c \lor b) \land a$$

Ortholattices:

Distributivity: $a \land (b \lor c) = (a \land b) \lor (a \land c)$ ✗
Absorption: $a \land (a \lor c) = a$ ✓

- Satisfies all other Boolean laws
- $\implies$ sound approximation of Boolean equivalence
- Algorithm for quadratic-time equivalence and implication checking
- Computes a normal form

Equivalence checking: Ortholattices

$$a \land (b \lor c) \quad \sim_{OL} \quad (c \lor b) \land a$$

Ortholattices:

Distributivity: $a \land (b \lor c) = (a \land b) \lor (a \land c)$ ✗
Absorption: $a \land (a \lor c) = a$ ✓

- Satisfies all other Boolean laws
- $\implies$ sound approximation of Boolean equivalence
- Algorithm for quadratic-time equivalence and implication checking
- Computes a normal form
- Also considers alpha-equivalence, reflexivity and symmetry of equality and more

Equivalence checking: Ortholattices

$$a \wedge (b \vee c) \quad \sim_{OL} \quad (c \vee b) \wedge a$$

Ortholattices:

Distributivity: $a \wedge (b \vee c) = (a \wedge b) \vee (a \wedge c)$ ✗
Absorption: $a \wedge (a \vee c) = a$ ✓

- Satisfies all other Boolean laws
- $\implies$ sound approximation of Boolean equivalence
- Algorithm for quadratic-time equivalence and implication checking
- Computes a normal form
- Also considers alpha-equivalence, reflexivity and symmetry of equality and more

Equivalence checking: Ortholattices

$$a \wedge (b \vee c) \quad \sim_{OL} \quad (c \vee b) \wedge a$$

Ortholattices:

Distributivity: $a \wedge (b \vee c) = (a \wedge b) \vee (a \wedge c)$ ✗

Absorption: $a \wedge (a \vee c) = a$ ✓

- Satisfies all other Boolean laws
- $\implies$ sound approximation of Boolean equivalence
- Algorithm for quadratic-time equivalence and implication checking
- Computes a normal form
- Also considers alpha-equivalence, reflexivity and symmetry of equality and more

Simon Guilloud, Mario Bucev, Dragana Milovančević, and Viktor Kunčak. "Formula normalizations in verification." In: *International Conference on Computer Aided Verification*. Springer. 2023, pp. 398–422

The Kernel

- Small, around 1200 LOC.
- Written in a restricted, simple subset of Scala
- Possibly feasible for formal verification

Proofs

```
val x = variable
val P = predicate(1)
val f = function(1)

val fixedPointDoubleApplication = Theorem(
        ∀(x, P(x) ⟹ P(f(x))) ⊢ P(x) ⟹ P(f(f(x)))
    ) {
    assume(∀(x, P(x) ⟹ P(f(x))))

    val step1 = have(P(x) ⟹ P(f(x))) by InstantiateForall
    val step2 = have(P(f(x)) ⟹ P(f(f(x)))) by InstantiateForall

    have(thesis) by Tautology.from(step1, step2)
}
```

Tactics and Writing Them

- Tactics are simply functions computing proofs

Tactics and Writing Them

- Tactics are simply functions computing proofs
- Freely mix Scala code with LISA proofs and DSL

Tactics and Writing Them

- Tactics are simply functions computing proofs
- Freely mix Scala code with LISA proofs and DSL
- Given a proof state, play with it as you want...

Tactics and Writing Them

- Tactics are simply functions computing proofs
- Freely mix Scala code with LISA proofs and DSL
- Given a proof state, play with it as you want…
- …return a proof at the end

Tactics and Writing Them — Propositional Solver

To prove a formula (“OL-DPLL”):

Tactics and Writing Them — Propositional Solver

To prove a formula (“OL-DPLL”):

- normalize the formula

Tactics and Writing Them — Propositional Solver

To prove a formula (“OL-DPLL”):

- normalize the formula
- if it is true, done

Tactics and Writing Them — Propositional Solver

To prove a formula (“OL-DPLL”):

- normalize the formula
- if it is true, done
- if it is false, throw an error

Tactics and Writing Them — Propositional Solver

To prove a formula (“OL-DPLL”):

- normalize the formula
- if it is true, done
- if it is false, throw an error
- in any other case, choose your favourite atom, say A

Tactics and Writing Them — Propositional Solver

To prove a formula (“OL-DPLL”):

- normalize the formula
- if it is true, done
- if it is false, throw an error
- in any other case, choose your favourite atom, say A
- prove the formula with $A \mapsto \top$

Tactics and Writing Them — Propositional Solver

To prove a formula ("OL-DPLL"):

- normalize the formula
- if it is true, done
- if it is false, throw an error
- in any other case, choose your favourite atom, say A
- prove the formula with $A \mapsto \top$
- prove the formula with $A \mapsto \bot$

Tactics and Writing Them — Propositional Solver

To prove a formula (“OL-DPLL”):

- normalize the formula
- if it is true, done
- if it is false, throw an error
- in any other case, choose your favourite atom, say A
- prove the formula with $A \mapsto \top$
- prove the formula with $A \mapsto \bot$
- combine

```
object Tautology extends ProofTactic {
    def solveFormula(f: Formula,
        decisionsPos: List[Formula],
        decisionsNeg: List[Formula]): proof.ProofTacticJudgement = {
        // proves decisionsPos ⊢ f :: decisionsNeg

        val normF = OLnormalForm(f)

        if (normF == ⊤) Restate(decisionsPos ⊢ f :: decisionsNeg)
        else if (normF == ⊥) InvalidProofTactic("Not a propositional tautology")

        else TacticSubproof {
            val atom = findBestAtom(normF)

            have(solveFormula(normF(atom → ⊤), atom :: decisionsPos, decisionsNeg)) // recursive
            val step2 = thenHave(atom :: decisionsPos ⊢ normF :: decisionsNeg)
                by Substitution(⊤ ⟺ atom)

            have(solveFormula(normF(atom → ⊥), decisionsPos, atom :: decisionsNeg)) // recursive
            val step4 = thenHave(decisionsPos ⊢ normF :: atom :: decisionsNeg)
                by Substitution(⊥ ⟺ atom)

            have(decisionsPos ⊢ normF :: decisionsNeg) by Cut(step4, step2)
            thenHave(decisionsPos ⊢ f :: decisionsNeg) by Restate
        }
    }
}
```

Mathematical Library

- Based on Tarski-Grothendieck (TG) Set Theory

Mathematical Library

- Based on Tarski-Grothendieck (TG) Set Theory
- TG $=$ ZFC with universes

Mathematical Library

- Based on Tarski-Grothendieck (TG) Set Theory
- TG = ZFC with universes
- Set theory — generally accepted foundation among mathematicians

Mathematical Library

- Based on Tarski-Grothendieck (TG) Set Theory
- TG $=$ ZFC with universes
- Set theory — generally accepted foundation among mathematicians
- Can formalize most modern mathematics

Mathematical Library

Currently, formalization includes:

- Functions and relations
- Partial and well orders
- Ordinals
- Transfinite induction and recursion

Mathematical Library

Currently, formalization includes:

- Functions and relations
- Partial and well orders
- Ordinals
- Transfinite induction and recursion

```
val transfiniteInduction = Theorem(
    ∀(x, ordinal(x) ⟹ (∀(y, y ∈ x ⟹ Q(y)) ⟹ Q(x)))
        ⊢ ∀(x, ordinal(x) ⟹ Q(x))
) {
    ...
}
val transfiniteRecursion = Theorem(
    ordinal(a) ⊢ ∃!(g, functionalOver(g, a) ∧
            ∀(b, b ∈ a ⟹ (app(g, b) ≡ F(restrictedFunction(g, b)))))
) {
    ...
}
```

Experience with an undergrad student

- Formalization of Group Theory
- Inside Set Theory

Experience with an undergrad student

- Formalization of Group Theory
- Inside Set Theory
- Homomorphisms, subgroups, etc.
- And some tactics!

LISA of the Future

LISA/HOL

We plan to develop

- an embedding of Higher-Order Logic (HOL) into Set Theory.

LISA/HOL

We plan to develop

- an embedding of Higher-Order Logic (HOL) into Set Theory.
- Embed types as sets, including function types

LISA/HOL

We plan to develop

- an embedding of Higher-Order Logic (HOL) into Set Theory.
- Embed types as sets, including function types
- Corresponds to a “soft” type system: in practice one usually doesn’t write $\emptyset \subset \pi$

LISA/HOL

We plan to develop

- an embedding of Higher-Order Logic (HOL) into Set Theory.
- Embed types as sets, including function types
- Corresponds to a “soft” type system: in practice one usually doesn’t write $\emptyset \subset \pi$
- Soft types carry information both for humans and for automation

LISA/HOL

We plan to develop

- an embedding of Higher-Order Logic (HOL) into Set Theory.
- Embed types as sets, including function types
- Corresponds to a “soft” type system: in practice one usually doesn’t write $\emptyset \subset \pi$
- Soft types carry information both for humans and for automation

LISA/HOL

We plan to develop

- an embedding of Higher-Order Logic (HOL) into Set Theory.
- Embed types as sets, including function types
- Corresponds to a "soft" type system: in practice one usually doesn't write $\emptyset \subset \pi$
- Soft types carry information both for humans and for automation

Mike Gordon. *Merging HOL with set theory.* Tech. rep. University of Cambridge, Computer Laboratory, 1994

LISA/Stainless

- Starting from Stainless, a program verifier for Scala
- Build foundations for more trustable program verification
- With more granular user feedback and interaction

LISA/Stainless

```
def plusOne(x: Int): Int = {
    x + 1
}

```

LISA/Stainless

```
def plusOne(x: Int): Int = {
    require(x >= 0)
    x + 1
}

```

LISA/Stainless

```scala
def plusOne(x: Int): Int = {
    require(x >= 0)
    x + 1
} ensuring(res => res >= 1)

```

LISA/Stainless

```
def plusOne(x: Int): Int = {
    require(x >= 0)
    x + 1
} ensuring(res => res >= 1)

//$> stainless myFile.scala
//$>  ...  counterexample

```

LISA/Stainless

SMT-based automation works quite well, till it doesn’t!

LISA/Stainless

- Horn-clause driven verification backend

LISA/Stainless

- Horn-clause driven verification backend
- Goal: Proof-producing program verification

LISA/Stainless

- Horn-clause driven verification backend
- Goal: Proof-producing program verification
- Integrate with the Eldarica Horn solver

LISA/Stainless

- Horn-clause driven verification backend
- Goal: Proof-producing program verification
- Integrate with the Eldarica Horn solver
- Augment to reconstruct LISA proofs

LISA/Stainless

- Horn-clause driven verification backend
- Goal: Proof-producing program verification
- Integrate with the Eldarica Horn solver
- Augment to reconstruct LISA proofs
- Use proofs for feedback with higher granularity and readability

LISA/Stainless

- Horn-clause driven verification backend
- Goal: Proof-producing program verification
- Integrate with the Eldarica Horn solver
- Augment to reconstruct LISA proofs
- Use proofs for feedback with higher granularity and readability
- $\implies$ program verification grounded in set-theory

LISA/Stainless

- Horn-clause driven verification backend
- Goal: Proof-producing program verification
- Integrate with the Eldarica Horn solver
- Augment to reconstruct LISA proofs
- Use proofs for feedback with higher granularity and readability
- $\implies$ program verification grounded in set-theory

LISA/Stainless

- Horn-clause driven verification backend
- Goal: Proof-producing program verification
- Integrate with the Eldarica Horn solver
- Augment to reconstruct LISA proofs
- Use proofs for feedback with higher granularity and readability
- $\implies$ program verification grounded in set-theory

Benefits outside of program verification too!

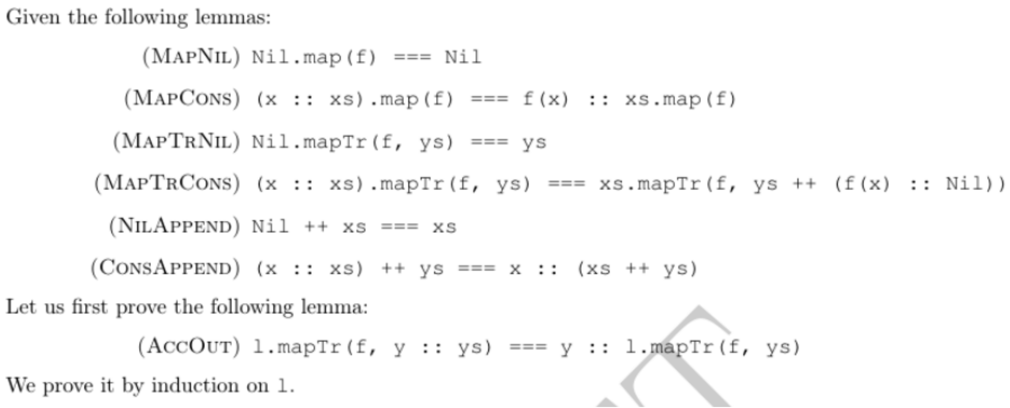
LISA/Education

- Goal: introducing more formal proofs to undergraduate students

LISA/Education

- Goal: introducing more formal proofs to undergraduate students
- Turns out we already have most of the ingredients

Proofs for Functional Programs



Given the following lemmas:

(MapNil) `Nil.map(f) === Nil`

(MapCons) `(x :: xs).map(f) === f(x) :: xs.map(f)`

(MapTrNil) `Nil.mapTr(f, ys) === ys`

(MapTrCons) `(x :: xs).mapTr(f, ys) === xs.mapTr(f, ys ++ (f(x) :: Nil))`

(NilAppend) `Nil ++ xs === xs`

(ConsAppend) `(x :: xs) ++ ys === x :: (xs ++ ys)`

Let us first prove the following lemma:

(AccOut) `l.mapTr(f, y :: ys) === y :: l.mapTr(f, ys)`

We prove it by induction on `l`.

Question 8 *Induction step:* `l` is `x :: xs`. Therefore, we need to prove:

```
(x :: xs).map(f) === (x :: xs).mapTr(f, Nil)
```

We name the inductions hypothesis IH.
Starting from the left hand-side (`(x :: xs).map(f)`), what exact sequence of lemmas should we apply to get the right hand-side (`(x :: xs).mapTr(f, Nil)`)?

- ☐ MapCons, NilAppend, AccOut, IH, MapTrCons
- ☐ MapCons, NilAppend, IH, AccOut, MapTrCons
- ☐ MapTrCons, IH, AccOut, NilAppend, MapCons
- ☐ MapTrCons, NilAppend, IH, IH, MapCons
- ☐ MapCons, IH, NilAppend, AccOut, MapTrCons
- ☐ MapCons, IH, NilAppend, MapTrCons, IH
- ☐ MapCons, IH, IH, NilAppend, MapTrCons
- ☐ MapCons, AccOut, IH, NilAppend, MapTrCons
- ☐ MapTrCons, AccOut, NilAppend, IH, MapCons
- ☐ MapCons, IH, NilAppend, MapTrCons, AccOut
- ☐ MapCons, NilAppend, AccOut, MapTrCons, AccOut
- ☐ MapCons, NilAppend, AccOut, MapTrCons, IH
- ☐ MapTrCons, IH, NilAppend, AccOut, MapCons
- ☐ MapCons, IH, AccOut, NilAppend, MapTrCons
- ☐ MapCons, NilAppend, IH, AccOut, MapTrCons
- ☐ MapCons, NilAppend, AccOut, AccOut, MapTrCons

Path to Integrated Program Proofs

Using LISA's DSL and Scala extensions, we can have a similar formal syntax:

```
val mapTrEq = Theorem(
    (x :: xs).map(f) ≡ (x  :: xs).mapTr(f, Nil)
) {
    ...
}

```

Path to Integrated Program Proofs

Using LISA's DSL and Scala extensions, we can have a similar formal syntax:

```
val mapTrEq = Theorem(
    (x :: xs).map(f) ≡ (x  :: xs).mapTr(f, Nil)
) {
    ...
}

```

- Since LISA is a Scala library, it integrates with students' existing IDE
- The syntax is intuitive enough, as it corresponds to actual functional programs

LISA — Summary

- Proof Assistant in Scala
- Small kernel based on schematic FOL
- Proof and Tactic interface with LISA’s DSL
- Mathematical library based on TG set theory

Future plans:

- Embedding of HOL
- Integration with Horn-clause based program verification
- Proofs for undergraduate functional programming

References

[1] Simon Guilloud, Mario Bucev, Dragana Milovančević, and Viktor Kunčak. “Formula normalizations in verification.” In: *International Conference on Computer Aided Verification*. Springer. 2023, pp. 398–422.

[2] Mike Gordon. *Merging HOL with set theory*. Tech. rep. University of Cambridge, Computer Laboratory, 1994.

```
val myTheorem = Theorem(P ∧ Q ⊢ Q ∧ P) {
    assume(P ∧ Q)
    have(Q ∧ P) by Restate
}

```

```scala
val myTheorem = Theorem(P ∧ Q ⊢ Q ∧ P) {
    assume(P ∧ Q)
    have(Q ∧ P) by Restate
}

```

Just Scala syntax!

```scala
have(
    ConnectorFormula(And, Seq(Q, P))
)
.by(using proof)(Restate)

```