LISA Reference Manual (WIP)

Simon Guilloud
Laboratory for Automated Reasoning and Analysis, EPFL

Introduction

This document aims to give a complete documentation on LISA. Tentatively, every chapter and section will explain a part or concept of LISA, and explains both its implementation and its theoretical foundations.

Part I

Reference Manual

Chapter 1

LISA's trusted code: The Kernel

LISA's kernel is the starting point of LISA, formalising the foundations of the whole theorem prover. It is the only trusted code base, meaning that if it is bug-free then no further erroneous or malicious code can violate the soundness property and prove invalid statements. Hence, the two main goals of the kernel are to be efficient and trustworthy.

LISA's foundations are based on very traditional (in the mathematical community) foundational theory of all mathematics: **First Order Logic**, expressed using **Sequent Calculus** (and augmented with schematic symbols), with axioms of **Set Theory**. Interestingly, while LISA is built with the goal of using Set Theory, the kernel is actually theory-agnostic and is sound to use with any other set of axioms. Hence, we defer Set Theory to chapter 2.

1.1 First Order Logic

1.1.1 Syntax

Definition 1 (Terms)**.** In LISA, the set of terms $\mathcal{T}$ is defined by the following grammar:

$$\mathcal{T} := \mathcal{L}_{Term}(\text{List}[\mathcal{T}]) \tag{1.1}$$

Where $\mathcal{L}_{Term}$ is the set of *term labels*:

$$\begin{aligned} \mathcal{L}_{Term} := &\,\text{ConstantTermLabel}(\text{Id}, \text{Arity}) \\ &\mid \text{SchematicTermLabel}(\text{Id}, \text{Arity}) \end{aligned} \tag{1.2}$$

A label can be either *constant* or *schematic*, and is made of an identifier (a string) and the arity of the label (an integer). A term is made of a term

label and a list of children, whose length must be equal to the arity of the label. A constant label of arity 0 is sometimes called just a constant, and a schematic label of arity 0 a variable. We define the shortcut

$$\mathrm{Var}(x) \equiv \mathrm{SchematicTermLabel}(x, 0)$$

As the definition states, we have to kind of function symbols: *Constant* ones and *Schematic*. Constant labels represent a fixed function symbol in some language, for exemple the addition "+" in peano arithmetic.

Schematic symbols on the other hand are uninterpreted: They can represent any possible term and hence can be substituted by any term. Their use will become clearer in the next section when we introduce the concept of deductions. Moreover, variables, which are schematic terms of arity 0, can be bound in formulas as we explain bellow. [1]

Definition 2 (Formulas)**.** The set of Formulas $\mathcal{F}$ is defined similarly:

$$\begin{aligned} \mathcal{F} :=& \mathcal{L}_{Predicate}(\mathrm{List}[\mathcal{T}]) \\ &| \mathcal{L}_{Connector}(\mathrm{List}[\mathcal{F}]) \\ &| \,\mathrm{Binder(Id)(Var(Id)}, \mathcal{F}) \end{aligned} \tag{1.3}$$

Where $\mathcal{L}_{Predicate}$ is the set of *predicate labels*:

$$\begin{aligned} \mathcal{L}_{Predicate} :=& \,\mathrm{ConstantPredicateLabel(Id, Arity)} \\ &| \,\mathrm{SchematicPredicateLabel(Id, Arity)} \end{aligned} \tag{1.4}$$

and $\mathcal{L}_{Connector}$ is the set of *predicate labels*:

$$\begin{aligned} \mathcal{L}_{Connector} :=& \,\mathrm{ConstantConnectorLabel(Id, Arity)} \\ &| \,\mathrm{SchematicConnectorLabel(Id, Arity)} \end{aligned} \tag{1.5}$$

A formula can be constructed from a list of term with a predicate label:

$$\leq(x, 7)$$

A formula can be constructed from a list of smaller formulas using a connector label:

$$\leq(x, 7) \land \geq(x, 5)$$

And finally a formula can be constructed from a variable and a smaller formula using a binder:

$$\exists x.\, (\leq(x, 7) \land \geq(x, 5))$$

[1] In a very traditional presentation of first order logic, we would only have variables, i.e. schematic terms of arity 0, and schematic terms of higher arity would only appear in second order logic. We defer to Part II Section 3.1 the explanation of why our inclusion of schematic function symbols doesn't fundamentally move us out of First Order Logic.

Connectors and predicate, like terms, can exist in either constant or schematic form. Note that Connectors and predicates vary only in the type of arguments they take and hence Connectors and Predicates of arity 0 are the same thing. Hence, in LISA, we don't permit connectors of arity 0 and ask to use predicates instead. A contrario to schematic terms of arity 0, schematic predicates of arity 0 can't be bound, but they still play a special role sometimes, and hence we introduce the special notation for them

$$\text{FormulaVar}(\text{Id}) = \text{SchematicPredicateLabel}(\text{Id}, 0)$$

Moreover in LISA, A contrario to constant predicate and term symbols, which can be freely created, there is only the following finite set of constant connector symbols in LISA:

$$\text{Neg}(\neg, 1) \mid \text{Implies}(\rightarrow, 2) \mid \text{Iff}(\leftrightarrow, 2) \mid \text{And}(\wedge, -1) \mid \text{Or}(\vee, -1)$$

Moreover, connectors (And and Or) are allowed to have an unrestricted arity, represented by the value -1. This means that a conjunction or disjunction can have any finite number of children. Similarly, there are only 3 binder labels.

$$\text{Forall}(\forall) \mid \text{Exists}(\exists) \mid \text{ExistsOne}(\exists!)$$

We also introduce a special constant predicate symbol, equality:

$$\text{Equal}(=)$$

In this document as well as in the code documentation, we often write terms and formula in a more conventional way, generally hiding the arity of labels and representing the label with it's identifier only, preceded by an interrogation mark ? if the symbol is schematic. When the arity is relevant, we write it with an superscript, for example:

$$f^3(x, y, z) \equiv \text{Fun}(f, 3)(\text{List}(\text{Var}(x), \text{Var}(y), \text{Var}(z)))$$

and

$$\forall x.\phi \equiv \text{Binder}(\forall, \text{Var}(x), \phi)$$

We also use other usual representations such as symbols in infix position, omitting parenthesis according to usual precedence rules, etc. FInally, note that we use subscript to emphasis that a variable is possibly free in a term or formula:

$$t_{x,y,z}, \phi_{x,y,z}$$

Convention Throughout this book and in the code base we adopt the following conventions. We use r, s, t, u to denote arbitrary terms, a, b, c to denote constant term symbols of arity 0 and f, g, h to denote term symbols of arity non-0. We preceede those with an interogation mark, sucha as $?f$ to

denote schematic symbols. Moreover, we also use x, y, z to denote variables (schematic terms of order 0).

For formulas, we use greek letters such as ϕ, ψ, τ to denote arbitrary formula, ν, μ to denote formula variables. We use capital letters like P, Q, R to denote predicate symbols, precedding them with an interrogation mark ? for schematic predicates. Schematic connectors are rarer, but when they appear, we preceede them by 2 interrogation marks, for example $??c$. Sets or sequences of formula are denoted with capital greek letters Π, Σ, Γ, Δ.

•

1.1.2 Substitution

On top of basic building of terms and formulas, there is one important type of operations: substitution of schematic symbols, which has to be implemented in a capture-avoiding way. We start with the subcase of variable substitution

Definition 3 (Capture-avoiding Substitution of variables)**.** Given a base term t, a variable x and another term r, the substitution of x by r inside t is noted $t[r/x]$ and simply consists in replacing all appearance of x by r.

Given a formula ϕ, the substitution of x by r inside ϕ is defined recursively the obvious way for connectors and predicates, and for binders:

$$(\forall y.\psi)[r/x] \equiv \forall y.(\psi[r/x])$$

if y does not appear in r and

$$(\forall y.\psi)[r/x] \equiv \forall z.(\psi[z/y][r/x])$$

with any fresh variable z (which is not free in r and ϕ) otherwise.

This definition of substitution is justified by the notion of alpha equivalence: Two formulas which are identical up to renaming of bound variables are considered equivalent. In practice, this means that the free variables inside r will never get caught when substituted.

We can now define "lambda terms"

Definition 4 (Lambda Terms)**.** A lambda term is a meta expression (meaning that it is not part of FOL itself) consisting in a term with "holes" that can be filled by other terms. This is represented with specified variables as arguments, similar to lambda calculus. For example, for a functional term with two arguments, we write

$$L = \text{Lambda}(\text{Var}(x), \text{Var}(y))(t_{x,y})$$

It comes with an instantiation operation: given terms r, s,

$$L(r, s) = t[r/x, s/y]$$

Those expressions are a generalization of terms, and would be part of our logic if we used Higher Order Logic rather than First Order Logic. For conciseness and familiarity, in this document and in code documentation, we represent those expressions as lambda expressions:

$$\lambda x.y.t$$

They are usefull because similarly as variables can be substituted by terms, schematic terms labels of arity greater than 0 can be substituted by such functional terms. As the definition of such substitution is rather convoluted to describe, we prefer to show examples and redirect the reader to the source code of LISA for a technical definition. [2]

Example 1 (Functional terms substitution in terms)**.**

Base term	Substitution			Result
$?f(0,3)$	$?f$	$\to$	$\lambda x.y.x+y$	$0+3$
$?f(0,3)$	$?f$	$\to$	$\lambda y.x.x-y$	$3-0$
$?f(0,3)$	$?f$	$\to$	$\lambda x.y.y+y-10$	$3+3-10$
$10\times ?g(x)$	$?g$	$\to$	$\lambda x.x^2$	$10\times x^2$
$10\times ?g(50)$	$?g$	$\to$	$\lambda x.?f(x+2,z)$	$10\times ?f(50+2,z)$
$?f(x,x+y)$	$?f$	$\to$	$\lambda x.y.\cos(x-y)*y$	$\cos(x-(x+y))*(x+y)$

Those definition extends to substitution of schematic terms inside formulas, with capture free substitution for bound variables. For example:

Example 2 (Functional terms substitution in formulas)**.**

Base formula	Substitution			Result
$?f(0,3)=?f(x,x)$	$?f$	$\to$	$\lambda x.y.x+y$	$0+3=x+x$
$\forall x.?f(0,3)=?f(x,x)$	$?f$	$\to$	$\lambda x.y.x+y$	$\forall x.0+3=x+x$
$\exists y.?f(y)\leq ?f(5)$	$?f$	$\to$	$\lambda x.x+y$	$\exists y_1.y_1+y\leq 5+y$

Note that if the lambda expression contains free variables (such as y in the last example), then appropriate alpha-renaming of variables may be needed.

We similarly define functional formulas, except than those can take either term arguments of formulas arguments. Specifically, we use LambdaTermTerm, LambdaTermFormula, LambdaFormulaFormula to indicate functional expression that take terms or formulas as arguments and return a term or formula.

Example 3 (Typical functional expressions)**.**

$\text{LambdaTermTerm}(Var(x),Var(y))(x+y)$	$=$	$\lambda x.y.x+y$
$\text{LambdaTermFormula}(Var(x),Var(y))(x=y)$	$=$	$\lambda x.y.x=y$
$\text{LambdaFormulaFormula}(\text{FormulaVar}(\nu),\text{FormulaVar}(\mu))$	$=$	$\lambda \nu.\mu.\nu\wedge\mu$

Not that in the last case, we use FormulaVar to represent the arguments of the lambda formula. Substitution of functional formulas is completely analog to (capture free!) substitution of functional terms.

[2] Note that in lambda calculus, this would simply be iterated beta-reduction.

1.1.3 The Equivalence Checker

While proving theorem, trivial syntactical transformations such as $p \wedge q \equiv q \wedge p$ significantly increase the length of proofs, which is desirable neither for the user nor the machine. Moreover, the proof checker will very often have to check whether two formulas that appear in different sequents are the same. Hence, instead of using pure syntactical equality, LISA implements a powerful equivalence checker able to detect a class of equivalence-preserving logical transformation. As an example, two formulas $p \wedge q$ and $q \wedge p$ would be naturally treated as equivalent.

For soundness, the relation decided by the algorithm should be contained in the $\iff$ "if and only if" relation of first order logic. It is well known that this this relationship is in general undecidable however, and even the $\iff$ relation for propositional logic is coNP-complete. For practicality, we need a relation that is efficiently computable

The decision procedure implemented in LISA takes time log-linear in the size of the formula, which means that is is only marginally slower than syntactic equality checking. It is based on an algorithm that decides the word-problem for Orthocomplemented Bisemilattices [1]. Informally, the theory of Orthocomplemented Bisemilattices is the same as that of Propositional Logic, but without the distributivity law. Figure 1.1 shows the axioms of this theory and the logical transformations LISA is able to automatically figure out. Moreover, the implementation in LISA also takes into account symmetry and reflexivity of equality as well as alpha-equivalence, by which we mean renaming of bound variables.

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:	$x \implies y = \neg x \vee y$		
L10:	$x \leftrightarrow y = (\neg x \vee y) \wedge (\neg y \vee x)$		
L11:	$\exists! x.P = \exists y.\forall x.(x = y) \leftrightarrow P$		

Table 1.1: Laws LISA's equivalence checker authomatically account for. LISA's equivalence-checking algorithm is complete (and log-linear time) with respect to laws L1-L11 and L1'-L8'.

1.2 Proofs in Sequent Calculus

1.2.1 Sequent Calculus

The deductive system used by LISA is an extended version of Gentzen's sequent Calculus.

Definition 5. A **sequent** is a pair of (possibly empty) sets of formula, noted:

$$\phi_1, \phi_2, ..., \phi_m \vdash \psi_1, \psi_2, ..., \psi_n$$

The intended semantic of such a sequent is:

$$(\phi_1 \land \phi_2 \land ... \land \phi_m) \implies (\psi_1 \lor \psi_2 \lor ... \lor \psi_n) \tag{1.6}$$

A sequent $\phi \vdash \psi$ is logically but not conceptually equivalent to a sequent $\vdash \phi \to \psi$. The distinction is similar to the distinction between meta-implication and inner implication in Isabelle, for example. Typically, a theorem or a lemma should have its various assumptions on the left handside of the sequent and its single conclusion on the right. During proofs however, there may be multiple elements on the right side. [3]

A deduction rule, also called a proof step, has (in general) between zero and two prerequisite sequents (which we call *premises* of the rule) and one conclusion sequent, and possibly take some arguments that describe how the deduction rule is applied. The basic deduction rules used in LISA are shown in Figure 1.1.

Since we work on first order logic with equality and accept axioms, there are also rules for equality reasoning, which include reflexivity of equality. Moreover, we include substitution of equal-for-equal and equivalent-for-equivalent in Figure 1.2. While those substitution rules are deduced steps, and hence could technically be omitted, simulating them can sometime take a high number of steps so they are included as base steps for efficiency.

There are also some special proof steps used to either organise proofs, shown in Figure 1.3.

[3] In a strict description of Sequent Calculus, this is in particular needed to make double negation elimination.

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

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

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

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

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

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

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

$$\frac{\Gamma, \phi[t/x] \vdash \Delta}{\Gamma, \forall x.\phi \vdash \Delta}\ \texttt{LeftForall} \qquad \frac{\Gamma \vdash \phi, \Delta}{\Gamma \vdash \forall x.\phi, \Delta}\ \texttt{RightForall}$$

$$\frac{\Gamma, \phi \vdash \Delta}{\Gamma, \exists x.\phi \vdash \Delta}\ \texttt{LeftExists} \qquad \frac{\Gamma \vdash \phi[t/x], \Delta}{\Gamma \vdash \exists x.\phi, \Delta}\ \texttt{RightExists}$$

$$\frac{\Gamma, \exists y \forall x.(x = y) \leftrightarrow \phi \vdash \Delta}{\Gamma, \exists! x.\phi \vdash \Delta}\ \texttt{LeftExistsOne} \qquad \frac{\Gamma \vdash \exists y \forall x.(x = y) \leftrightarrow \phi, \Delta}{\Gamma \vdash \exists! x.\phi, \Delta}\ \texttt{RightExistsOne}$$

$$\frac{\Gamma \vdash \Delta}{\Gamma, \Sigma \vdash \Delta}\ \texttt{LeftWeakening} \qquad \frac{\Gamma \vdash \Delta}{\Gamma \vdash \delta, \Delta}\ \texttt{RightWeakening}$$

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

Figure 1.1: Strict set of deduction rules for sequent calculus.

$$\frac{\Gamma, \phi[s/?f] \vdash \Delta}{\Gamma, s = t, \phi[t/?f] \vdash \Delta}\ \texttt{LeftSubstEq} \qquad \frac{\Gamma \vdash \phi[s/?f], \Delta}{\Gamma, s = t \vdash \phi[t/?f], \Delta}\ \texttt{RightSubstEq}$$

$$\frac{\Gamma, \phi[a/?p] \vdash \Delta}{\Gamma, a \leftrightarrow b, \phi[b/?p] \vdash \Delta}\ \texttt{LeftSubstIff} \qquad \frac{\Gamma \vdash \phi[a/?p], \Delta}{\Gamma, a \leftrightarrow b \vdash \phi[b/?p], \Delta}\ \texttt{RightSubstIff}$$

$$\frac{\Gamma \vdash \Delta}{\Gamma[\psi(\vec{v})/?p] \vdash \Delta[\psi(\vec{v})/?p]}\ \texttt{InstPredSchema} \qquad \frac{\Gamma \vdash \Delta}{\Gamma[r(\vec{v})/?f] \vdash \Delta[r(\vec{v})/?f]}\ \texttt{InstFunSchema}$$

Figure 1.2: Additional deduction rules for substitution and instantiation

$$\frac{\Gamma \vdash \Delta}{\Gamma \vdash \Delta}\ \texttt{Rewrite} \qquad \frac{}{\Gamma \vdash \Gamma}\ \texttt{RewriteTrue}$$

$$\frac{\cdots \quad \cdots \quad \cdots}{\Gamma \vdash \Delta}\ \texttt{Subproof}$$

Figure 1.3: Bonus and structural proof steps. Rewrite allows to deduce a sequent equivalent from a previous sequent by OCBSL laws and sequent interpretation. Subproof hide a part of a proof tree inside a single proof step.

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\phi \vdash \phi}\,\texttt{Hypothesis}}{\phi \vdash \phi, \psi}\,\texttt{RightWeakening}}{\vdash \phi, (\phi \to \psi)}\,\texttt{RightImplies} \qquad \dfrac{}{\phi \vdash \phi}\,\texttt{Hypothesis}}{\dfrac{(\phi \to \psi) \to \phi \vdash \phi}{\vdash ((\phi \to \psi) \to \phi) \to \phi}\,\texttt{RightImplies}}\,\texttt{LeftImplies}$$

Figure 1.4: A proof of Pierce's law in sequent calculus. The bottom most sequent (root) is the conclusion.

1.2.2 Proofs

Proof step can be composed into a Directed Acyclic Graph. The root of the proof shows the conclusive statement, and the leaves are assumptions or tautologies (instances of the`Hypothesis` rule). Figure 1.4 shows an example of a proof tree for Pierce's Law in strict Sequent Calculus.

In the Kernel, proof steps are organised linearly, in a list, to form actual proofs. Each proof step refer to it's premises using numbers, which indicate the place of the premise in the proof. Moreover, proofs are conditional: They can carry an explicit set of assumed sequents, named "imports", which give some starting points to the proof. Typically, these imports will contain previously proven theorems, definitions or axioms (More on that in section 1.3). For a proof step to refer to an imported sequent, one use negative integers. -1 corresponds to the first sequent of the import list of the proof, -2 to the second, etc.

Formally, a proof is a pair made of a list of proof steps and a list of sequents:

$$\text{Proof}(\text{steps}{:}\,\text{List}\,[\text{ProofStep}],\ \text{imports}{:}\text{List}\,[\text{Sequent}])$$

We call the bottom sequent of the last proof step of the proof the "conclusion" of the proof. For the proof to be the linearization of a rooted directed acyclic graph, we require that proof steps must only refer to number smaller then their own number in the proof. Indeed, using topological sorting, it is always possible to order the nodes of a directed acyclic graph such that for any node, its predecessors appear earlier in the list. The linearization of our Pierce's law proof is shown in Figure 1.5.

1.2.3 Proof Checker

In LISA, a proof object has no guarantee to be correct. It is perfectly possible to wright a wrong proof. LISA contains a *proof checking* function, which given a proof will verify if it is correct. To be correct, a proof must satisfy the following conditions:

1. No proof step must refer to itself or a posterior proof step as a premise.

$$
\begin{array}{ll}
0 & \texttt{Hypothesis} \quad \phi \vdash \phi \\
1 & \texttt{RightWeakening}(0) \quad \phi \vdash \phi, \psi \\
2 & \texttt{RightImplies}(1) \quad \vdash \phi, (\phi \to \psi) \\
3 & \texttt{Hypothesis} \quad \phi \vdash \phi \\
4 & \texttt{LeftImplies}(2, 3) \quad (\phi \to \psi) \to \phi \vdash \phi \\
5 & \texttt{RightImplies}(4) \quad \vdash ((\phi \to \psi) \to \phi) \to \phi
\end{array}
\tag{1.7}
$$

Figure 1.5: Linearization of the proof of Pierce's Law as represented in LISA.

2. Every proof step must be correctly constructed, with the bottom sequent correctly following from the premises by the type of the proof step and its arguments.

Given some proof p, the proof checker will verify these points. For most proof steps, this typically involve verifying that the premises and the conclusion match according to a transformation specific to the deduction rule. Not that for most case where there is an intuitive symmetry in arguments, such as `RightAnd` or `LeftSubstIff` for example, permutations of those arguments don't matter.

Hence, most of the proof checker's work consists in verifying that some formulas, or subformulas thereof, are identical. This is where the equivalence checker comes into play. By checking equivalence rather than strict syntactic equality, a lot of steps become redundant and can be merged in a single step. That way, `LeftAnd`, `RightOr`, `LeftIff` become instances of the `Weakening rule`, and `RightImplies` an instance of `RightAnd`.

`LeftNot`, `RightNot`, `LeftImplies`, `RightImplies`, `LeftRefl`, `RightRefl`, `LeftExistsOne`, `RightExistsOne` can be omitted altogether. This gives an intuition of how usefull the equivalence checker is to cut proof length. It also combine very well with substitution steps.

While most proof steps are oblivious to formula transformations allowed by the equivalence checker, they don't allow transformations of the whole sequent: To easily rearrange sequents according to the semantic of 1.6, one should use the `Rewrite` step. The proof checking function will output a *judgement*:

SCValidProof(proof: SCProof)

or

SCInvalidProof(proof: SCProof, path: Seq[Int], message: String)

SCInvalidProof indicates an erroneous proof. The second argument point to the faulty proofstep (through subproofs), and the third argument is an error message hinting towards the reason why the step is faulty.

1.3 Theorems and Theories

In mathematics as a discipline, theorems don't exist in isolation. They depend on some agreed uppon set of axioms, definitions, and previously proven theorems. Formally, Theorems are developped within theories. A theory is defined by a language, which contains the symbols allowed in the theory, and by a set of axioms, which are assumed to hold true.

In LISA, a theory is a mutable object that starts by being the pure theory of predicate logic: It has no known symbol and no axiom. Then we can introduce into it symbols of set theory($\in$, $\emptyset$, $\bigcup$ and set theory axioms, see Chapter 2) or of any other theory.

To conduct a proof inside a Theory, using its axioms, the proof should be normally constructed and the needed axioms specified in the imports of the proof. Then, the proof can be given to the Theory to check, along with *justifications* for all imports of the proof. A justification is either an axiom, a previously proven theorem, or a definition. The Theory object will check that every import of the proof is properly justified by an axiom introduced in the theory, i.e. that the proof is in fact not conditional in the theory. Then it will pass the proof to the proof checker. If the proof is correct, it will return a Theorem encapsulating the sequent. This sequent will be allowed to be used in all further proofs exactly like an axiom.

1.3.1 Definitions

The user can also introduce definitions in the Theory. LISA's kernel allows to define two kinds of objects: Function (or Term) symbols and Predicate symbols. It is important to remember that in the context of Set Theory, function symbols are not the usual mathematical functions and predicate symbols are not the usual mathematical relations. Indeed, on one hand a function symbol defines an operation on all possible sets, but on the other hand it is impossible to use the symbol alone, without applying it to arguments, or to quantify over function symbol. Actual mathematical functions on the other hand, are proper sets which contains the graph of a function on some domain. Their domain must be restricted to a proper set, and it is possible to quantify over such set-like functions or to use them without applications. These set-like functions are represented by constant symbols. For example "f is derivable" cannot be stated about a function symbol. We will come back to this in Chapter 2, but for now let us remember that (non-constant) function symbols are suitable for intersection $\bigcap$ between sets but not for, say, the Riemann ζ function.

Figure 1.6 shows how to define and use new function and predicate symbols. To define a predicate on n variables, we must provide any formula with n distinguished free variables. Then, this predicate can be freely used and at any time substituted by its definition. Functions are slightly more

A definition in LISA is one of those two kinds of objects:

```
PredicateDefinition(
                label: ConstantPredicateLabel,
                expression: LambdaTermFormula
)
```

Corresponding to "let $p^n(\vec{x}) := \phi_{\vec{x}}$"

```
FunctionDefinition(
                label: ConstantFunctionLabel,
                out: VariableLabel,
                expression: LambdaTermFormula
)
```

Corresponding to "let $f(\vec{x})$ be the unique element s.t. $\phi[f(\vec{x})/y]$"

Figure 1.6: Definitions in LISA

complicated: To define a function f, one must first prove a statement of the form

$$\exists! y.\phi_{y,x_1,...,x_k}$$

Then we obtain for free the property that

$$\forall y.(f(x_1,...,x_k) = y) \leftrightarrow \phi_{y,x_1,...,x_k}$$

from which we can deduce in particular $\phi[f(x_1,...,x_k)/y]$ The special case where $n = 0$ defines constant symbols. The special case where ϕ is of the form $y = t$, with possibly the x's free in t let us recover a more simple definition *by alias*, i.e. where f is simply a shortcut for a more complex term t. This mechanism is typically called *Extension by Definition*, and allows to extend the theory without changing what is or isn't provable. For detailed explanation, see part II chapter chapt:definitions.

The Theory object is responsible of keeping track of all symbols which have been defined so that it can detect and refuse conflicting definitions. As a general rule, definitions should have a unique identifier and can't contains free schematic symbols.

Once a definition has been introduce, future theorem can refer to those definitional axioms by importing the corresponding sequents in their proof and providing justification for those imports when the proof is verified, just like with axioms.

Figure 1.7 shows the types on Justification in a theory (Theorem, Axiom, Definition). Figure 1.8 shows how to introduce new such justifications as well as symbols in the theory. Figure 1.9 shows how to obtain various informations from the theory.

Justifications:

Explanation	Data Type

A proven theorem	`Theorem(`
	`  name: String,`
	`  proposition: Sequent`
	`  )`

An axiom of the theory	`Axiom(`
	`  name: String,`
	`  ax: Formula`
	`  )`

A predicate definition	`PredicateDefinition(`
	`  label: ConstantPredicateLabel,`
	`  expression: LambdaTermFormula`
	`  )`

A function definition	`FunctionDefinition(`
	`  label: ConstantFunctionLabel,`
	`  out: VariableLabel,`
	`  expression: LambdaTermFormula`
	`  )`

Figure 1.7: The different type of justification in a Theory object.

Setters/Constructors:

Explanation	Function
Add a new theorem to the theory	`makeTheorem( name: String, statement: Sequent, proof: SCProof, justs: Seq[Justification] )`
Add a new axiom to the theory	`addAxiom( name: String, f: Formula )`
Make a new predicate definition	`makePredicateDefinition( label: ConstantPredicateLabel, expression: LambdaTermFormula )`
Make a new function definition	`makeFunctionDefinition( proof: SCProof, justifications: Seq[Justification], label: ConstantFunctionLabel, out: VariableLabel, expression: LambdaTermFormula )`
Add a new symbol without definition	`addSymbol(s: ConstantLabel)`
Add all symbols of a formula without definition	`makeFormulaBelongToTheory(phi: Formula)`
Add all symbols of a sequent without definition	`makeSequentBelongToTheory(s: Sequent)`

Figure 1.8: The mutable interface of a Theory object.

Getters:

Explanation	Function

Check if all symbols in a formula, term or sequent belong to the theory.	`belongsToTheory(phi: Formula)`
	`belongsToTheory(t: Term)`
	`belongsToTheory(s: Sequent)`

Return the list of symbols and definitions in the theory	`language()`
Check if a label is a symbol of the theory	`isSymbol(label: ConstantLabel)`
Check if a label is *not* already a symbol of the theory	`isAvailable(label: ConstantLabel)`
Return the list of axioms in the theory	`axiomsList()`
Check if a formula is an axiom of the theory	`isAxiom(f: Formula)`
Return the Axiom matching the given name or formula, if it exists	`getAxiom(f: Formula)` `getAxiom(name: String)`
Return the Definition of a given Label, if defined	`getDefinition(label: ConstantLabel)`
Return the Theorem object with the given name, if it is one.	`getTheorem(name: String)`

Figure 1.9: The static interface of a Theory object.

1.4 Kernel Supplements and Utilities

The Kernel itself is a logical core, whose main purpose is to attest correctness of mathematical developments and proofs. In particular, it is not intended to use directly to formalise large library, but rather as either a foundation for LISA's user interface and automation, or as a tool to write and verify formal proofs produced by other programs. Nonetheless, LISA's kernel comes with a set of utilities and features that make the kernel more usable. LISA also provides a set of utilities and a DSL[4] to ease and organise the writing of proofs. This is especially directed to people who want to build understanding and intuition regarding formal proofs in sequent calculus.

1.4.1 Printer and Parser

Under development

1.4.2 Writing theory files

LISA provides a canonical way of writing and organizing Kernel proofs by mean of a set of utilities and a DSL[5] made possible by some of scala 3 features such as string interpolation, extension and implicits. This is especially directed to people who want to build understanding and intuition regarding formal proofs in sequent calculus. The way to write a new theory file to mathematical development is:

```
object MyTheoryName extends lisa.Main {

}
```

And that's it! To write a theorem, the recommended syntax is (for example):

```
object MyTheoryName extends lisa.Main {

  THEOREM("theoremName") of "desired_conclusion" PROOF {

    ???: Proof

  } using (listOfJustifications)
  show
}
```

show is optional and prints the last proven theorem. We can similarily make definition:

[4]Domain Specific Language
[5]Domain Specific Language

```
object MyTheoryName extends lisa.Main {

  val myFunction =
    DEFINE("symbol", x, y) as definition(x,y)
  show
}
```

This works for definitions of function and predicate symbols with a direct definition. for indirect definitions (via ∃!), use the following:

```
object MyTheoryName extends lisa.Main {

  val testdef =
    DEFINE("symbol", x, y) asThe z suchThat {
        ???:Formula
     } PROOF {
        ???:Proof
    } using (listOfJustifications)
    show
}
```

It is important to note that when multiple such files are developped, they all use the same underlying RunningTHeory. This makes it possible to use results proved previously by simple mean of an **import** statement as one would import a regular object.

To check the result of a developped file, and verify that the proofs contain no error, it is possible to run such a library object. All imported theory objects will be run through as well, but only the result of the selected one will be printed.

It is possible to refer to a theorem or axiom that has been previously proven or added using it's name. the syntax is thm"theoremName" or ax"axiomName". This makes it possible to write, for example, thm"theoremName".show and ... using (ax"comprehensionSchema"). Figure 1.10 shows a typical example of set theory development.

```
object MyTheoryName extends lisa.Main {
  THEOREM("russelParadox") of
    ∀x. (x∈?y)↔ ¬(x∈x)⊢ PROOF {
      val y = VariableLabel("y")
      val x = VariableLabel("x")
      val s0 = RewriteTrue(in(y, y) <=> !in(y, y) |-())
      val s1 = LeftForall(
        forall(x, in(x, y) <=> !in(x, x)) |- (),
        0, in(x, y) <=> !in(x, x), x, y
      )
      Proof(s0, s1)
    } using ()
  thm"russelParadox".show


  THEOREM("unorderedPair_symmetry") of
    "⊢∀y, x. {x, y} = {y, x}" PROOF {
      ...
    } using (ax"extensionalityAxiom", ax"pairAxiom")
  show


  val oPair =
    DEFINE("", x, y) as pair(pair(x, y), pair(x, x))




}
```

Figure 1.10: Example of library development in LISA Kernel

Chapter 2

Set Theory

LISA is based on set theory. More specifically, it is based on ZF with (still not decided) an axiom of choice, of global choice, or Tarski's universes.

ZF Set Theory stands for Zermelo-Frankel Set Theory. It contains a set of initial predicate symbols and function symbols, shown in Figure 2.1. It also contains the 7 axioms of Zermelo (Figure 2.2), which are technically sufficient to formalize a large portion initial portion of mathematics, plus the axiom of replacement of Frankel(Figure 2.3), which is needed to formalize more complex mathematical theories. In a more typical mathematical introduction to Set Theory, ZF would naturally only contain the set membership symbol $\in$. Axioms defining the other symbols would then only express the existence of functions or predicates with those properties, from which we could get the same symbols using extensions by definitions.

In a very traditional sense, an axiomatization is any possibly infinite semi-recursive set of axioms. Hence, in it's full generality, Axioms should be any function producing possibly infinitely many formulas. This is however not a convenient definition. In practice, all infinite axiomatizations are schematic, meaning that they are expressable using schematic variables. Axioms Z8 (comprehension schema) and ZF1 (replacement schema) are such examples of axiom schema, and motivates the use of schematic variables in LISA.

	Math symbol	LISA Kernel
Set Membership predicate	$\in$	in(s,t)
Subset predicate	$\subset$	subset(s,t)
Empty Set constant	$\emptyset$	emptyset()
Unordered Pair constant	$(\cdot,\cdot)$	pair(s,t)
Power Set function	$\mathcal{P}$	powerSet(s)
Set Union/Flatten function	$\bigcup$	union(x)

Figure 2.1: The basic symbols of ZF

Z1 (empty set). $\forall x.x \notin \emptyset$

Z2 (extensionality). $\forall x, y.(\forall z.z \in x \iff z \in y) \iff (x = y)$

Z3 (extensionality). $\forall x, y.x \subset y \iff \forall z.z \in x iff z \in y$

Z4 (pair). $\forall x, y, z.(z \in (x, y)) \iff ((x \in y) \vee (y \in z))$

Z5 (union). $\forall x, z.(x \in \mathrm{U}(z)) \iff (\exists y.(x \in y) \wedge (y \in z))$

Z6 (power). $\forall x, y.(x \in \mathcal{P}(y)) \iff (x \subset y)$

Z7 (foundation). $\forall x.(x \neq \emptyset) \implies (\exists y.(y \in x) \wedge (\forall z.z \in x))$

Z8 (comprehension schema). $\forall z, \vec{v}.\exists y.\forall x.(x \in y) \iff ((x \in z) \wedge \phi(x, z, \vec{v}))$

Figure 2.2: Axioms for Zermelo set theory.

ZF1 (replacement schema).

$$\forall a.(\forall x.(x \in a) \implies !\exists y.\phi(a, \vec{v}, x, y)) \implies$$

$$(\exists b.\forall x.(x \in a) \implies (\exists y.(y \in b) \wedge \phi(a, \vec{v}, x, y)))$$

Figure 2.3: Axioms for Zermelo-Fraenkel set theory.

Part II

Selected Theoretical Topics

Chapter 3

Set Theory and Mathematical Logic

3.1 First Order Logic with Schematic Variables

3.2 Extension by Definition

An extension by definition is the formal way of introducing new symbols in a mathematical theory. Theories can be extended into new ones by adding new symbols and new axioms to it. We're interested in a special kind of extension, called *conservative extension*.

Definition 6 (Conservative Extension)**.** A theory $\mathcal{T}_2$ is a conservative extension over a theory $\mathcal{T}_1$ if:

- $\mathcal{T}_1 \subset \mathcal{T}_2$
- For any formula ϕ in the language of $\mathcal{T}_1$, if $\mathcal{T}_2 \vdash \phi$ then $\mathcal{T}_1 \vdash \phi$

An extension by definition is a special kind of extension obtained by adding a new symbol and an axiom defining that symbol to a theory. If done properly, it should be a conservative extension.

Definition 7 (Extension by Definition)**.** A theory $\mathcal{T}_2$ is an extension by definition of a theory $\mathcal{T}_1$ if:

- $\mathcal{L}(\mathcal{T}_2) = \mathcal{L}(\mathcal{T}_2) \cup \{S\}$, where S is a single new function or predicate symbol, and
- $\mathcal{T}_2$ contains all the axioms of $\mathcal{T}_1$, and one more of the following form:
  - If S is a predicate symbol, then the axiom is of the form $\phi_{x_1,...,x_k} \iff S(x_1, ..., x_k)$, where ϕ is any formula with free variables among $x_1, ..., x_k$.

– If S is a function symbol, then the axiom is of the form $\phi_{y,x_1,...,x_k} \iff y = S(x_1, ..., x_k)$, where ϕ is any formula with free variables among $y, x_1, ..., x_k$. Moreover, in that case we require that

$$\exists! y.\phi_{y,x_1,...,x_k}$$

is a theorem of $\mathcal{T}_1$

We also say that a theory $\mathcal{T}_k$ is an extension by definition of a theory $\mathcal{T}_1$ if there exists a chain $\mathcal{T}_1$, $\mathcal{T}_2$, ... , $\mathcal{T}_k$ of extensions by definitions.

For function definition, it is common in logic textbooks to only require the existence of y and not its uniqueness. The axiom one would then obtain would only be $\phi[f(x_1, ..., x_n)/y]$ This also leads to conservative extension, but it turns out not to be enough in the presence of axiom schemas (axioms containing schematic symbols).

Lemma 1. *In ZF, an extension by definition without uniqueness doesn't necessarily yields a conservative extension if the use of the new symbol is allowed in axiom schemas.*

Proof. In ZF, consider the formula $\phi_c := \forall x.\exists y.(x \neq \emptyset) \implies y \in x$ expressing that nonempty sets contain an element, which is provable in ZFC.

Use this formula to introduce a new unary function symbol choice such that $\text{choice}(x) \in x$. Using it within the axiom schema of replacement we can obtain for any A

$$\{(x, \text{choice}(x)) \mid x \in A\}$$

which is a choice function for any set A. Hence using the new symbol we can prove the axiom of choice, which is well known to be independent of ZF, so the extension is not conservative. □

Note that this example wouldn't work if the definition required uniqueness on top of existence. For the definition with uniqueness, there is a stronger result than only conservativity.

Definition 8. A theory $\mathcal{T}_2$ is a fully conservative extension over a theory $\mathcal{T}_1$ if:

- It is conservative
- For any formula ϕ_2 with free variables $x_1, ..., x_k$ in the language of $\mathcal{T}_2$, there exists a formula ϕ_1 in the language of $\mathcal{T}_1$ with free variables among $x_1, ..., x_k$ such that

$$\mathcal{T}_2 \vdash \forall x_1 ... x_k.(\phi_1 \leftrightarrow \phi_2)$$

Theorem 1. *An extension by definition with uniqueness is fully conservative.*

The proof is done by induction on the height of the formula and isn't difficult, but fairly tedious.

Theorem 2. *If an extension $\mathcal{T}_2$ of a theory $\mathcal{T}_1$ with axiom schemas is fully conservative, then for any instance of the axiom schemas of an axiom schemas α containing a new symbol, $\Gamma \vdash \alpha$ where Γ contains no axiom schema instantiated with new symbols.*

Proof. Suppose

$$\alpha = \alpha_0[\phi/?p]$$

Where ϕ has free variables among $x_1, ..., x_n$ and contains a defined function symbol f. By the previous theorem, there exists ψ such that

$$\vdash \forall A, w_1, ..., w_n, x.\phi \leftrightarrow \psi$$

or equivalently, as in a formula and its universal closure are deducible from each other,

$$\vdash \phi \leftrightarrow \psi$$

which reduces to

$$\alpha_0[\psi/?p] \vdash \alpha$$

Since $\alpha_0[\psi/?p]$ is an axiom of $\mathcal{T}_1$, we reach the conclusion. □

Bibliography

[1] Simon Guilloud and Viktor Kuncak. Equivalence checking for orthocomplemented bisemilattices in log-linear time. In Dana Fisman and Grigore Rosu, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 28th International Conference, TACAS 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2-7, 2022, Proceedings, Part II*, volume 13244 of *Lecture Notes in Computer Science*, pages 196–214. Springer, 2022.