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 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** (augmented with schematic symbols), with axioms of **Set Theory**. While the LISA library is built on top of Set Theory axxioms, 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*:

$$\begin{aligned} \mathcal{L}_{Term} := \ & \text{ConstantTermLabel}(\text{Id}, \text{Arity}) && (1.2) \\ | \ & \text{SchematicTermLabel}(\text{Id}, \text{Arity}) && (1.3) \end{aligned}$$

A label can be either *constant* or *schematic*, and is made of an identifier (a pair of a string and an Integer, for example x_1) 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 called a *constant*, and a schematic label of arity 0 a *variable*. We define the abbreviation

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

Constant labels represent a fixed function symbol in some language, for example 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. [1]

Example 1 (Terms)**.** The following are typical examples of terms labels:

$$\begin{aligned}
\emptyset &:= \mathrm{ConstantTermLabel}(\text{"}\emptyset\text{"}, 0) \\
7 &:= \mathrm{ConstantTermLabel}(\text{"7"}, 0) \\
x &:= \mathrm{SchematicTermLabel}(\text{"}x\text{"}, 0) \\
+ &:= \mathrm{ConstantTermLabel}(\text{"+"}, 2) \\
f &:= \mathrm{SchematicTermLabel}(\text{"}f\text{"}, 1)
\end{aligned}$$

The following are examples of Terms:

$$\begin{aligned}
&\emptyset() := \emptyset(\mathrm{Nil}) \\
&7() := 7(\mathrm{Nil}) \\
&x() := x(\mathrm{Nil}) \\
&+(7(), x()) \\
&f(x())
\end{aligned}$$

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

$$\begin{aligned}
\mathcal{F} := \ & \mathcal{L}_{Predicate}(\mathrm{List}[\mathcal{T}]) && (1.4)\\
| \ & \mathcal{L}_{Connector}(\mathrm{List}[\mathcal{F}]) && (1.5)\\
| \ & \mathcal{L}_{Binder}(\mathrm{Var}(\mathrm{Id}), \mathcal{F}) \ , && (1.6)
\end{aligned}$$

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

$$\begin{aligned}
\mathcal{L}_{Predicate} := \ & \mathrm{ConstantPredicateLabel}(\mathrm{Id}, \mathrm{Arity}) && (1.7)\\
| \ & \mathrm{SchematicPredicateLabel}(\mathrm{Id}, \mathrm{Arity}) \ , && (1.8)
\end{aligned}$$

[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 **??** Section **??** the explanation of why our inclusion of schematic function symbols doesn't fundamentally move us out of First Order Logic.

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

$$\begin{aligned} \mathcal{L}_{Connector} := & \text{ ConstantConnectorLabel(Id, Arity)} && (1.9)\\ & \mid \text{SchematicConnectorLabel(Id, Arity)} \ . && (1.10) \end{aligned}$$

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

$$\begin{aligned} \mathcal{L}_{BinderLabel} := & \forall && (1.11)\\ & \mid \exists && (1.12)\\ & \mid \exists! && (1.13) \end{aligned}$$

Connectors and predicates, like terms, can exist in either constant or schematic forms. Note that connectors and predicates vary only in the type of arguments they takeso that connectors and predicates of arity 0 are essentially the same thing. Hence, LISA, does not permit connectors of arity 0 and suggests the use of 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, so we introduce a special notation for them

$$\text{FormulaVar}(X) \equiv \text{SchematicPredicateLabel}(X, 0) \ .$$

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

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

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

$$(\forall) \mid (\exists) \mid (\exists!) \ .$$

We also introduce a special constant predicate symbol, equality:

$$\text{Equality}(=, 2) \ .$$

Example 2 (Formula)**.** The following are typical examples of formula labels:

$$\begin{aligned}
\text{True} &:= \text{ConstantPredicateLabel}(\text{"True"}, 0)\\
\text{False} &:= \text{ConstantPredicateLabel}(\text{"False"}, 0)\\
X &:= \text{SchematicPredicateLabel}(\text{"}X\text{"}, 0)\\
= &:= \text{ConstantPredicateLabel}(\text{" = "}, 2)\\
\in &:= \text{ConstantPredicateLabel}(\text{"}\in\text{"}, 2)\\
P &:= \text{SchematicPredicateLabel}(\text{"}P\text{"}, 1)\\
\neg &:= \text{ConstantConnectorLabel}(\text{"}\neg\text{"}, 1)\\
\wedge &:= \text{ConstantConnectorLabel}(\text{"}\wedge\text{"}, -1)\\
\vee &:= \text{ConstantConnectorLabel}(\text{"}\vee\text{"}, -1)\\
\to &:= \text{ConstantConnectorLabel}(\text{"}\to\text{"}, 2)\\
\leftrightarrow &:= \text{ConstantConnectorLabel}(\text{"}\leftrightarrow\text{"}, 2)\\
c &:= \text{SchematicConnectorLabel}(\text{"}c\text{"}, 3)
\end{aligned}$$

Note that in the case of ConstantConnectorLabel, the list is exhaustive: $\neg, \wedge, \vee, \to$ and $\leftrightarrow$ are the only logical connectors accepted by LISA. The following are examples of Formulas:

$$\begin{aligned}
&\text{True}() := \text{True}(\text{Nil})\\
&X() := X(\text{Nil})\\
&P(x(), 7())\\
&\quad = (+(7(), x()), +(x(), 7()))\\
&\forall(x, =(x(), x()))\\
&\neg(\exists(x, \in(x(), \emptyset)))
\end{aligned}$$

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 its identifier only, preceded by an apostrophe (`) if we need to precise that a 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 emphasize that a variable is possibly free in a term or formula:

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

Convention Throughout this document, 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 non-0 arity. We precede those with an apostrophe, such as $`f$ to denote schematic symbols. 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 formulas, X, Y, Z to denote formula variables. We use capital letters like P, Q, R to denote predicate symbols, preceding them similarly with an apostrophe for schematic predicates. Schematic connectors are rarer, but when they appear, we use for example $`c$. Sets or sequences of formulas are denoted with capital greek letters Π, Σ, Γ, Δ, etc.

1.1.2 Substitution

On top of basic building blocks 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 denoted by $t[r/x]$ and is computed by replacing all occurences of x by r.

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

$$(\phi \wedge \psi)[r/x] \equiv \phi[r/x] \wedge \psi[r/x] \ ,$$
$$P(t_1, t_2, \ldots, t_n)[r/x] \equiv P(t_1[r/x], t_2[r/x], \ldots, t_n[r/x]) \ ,$$

and for binders as

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

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

if $y \neq x$ and 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 a term and specified symbols indicating the "holes". A lambda term can be though of as a meta-function on terms. For example, for a functional term with two arguments, we write

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

This is similar to the representation of functions in lambda calculus 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:

$$\lambda xy.t_{x,y}$$

Similarly as how variables can be substituted by terms, schematic terms labels of arity greater than 0 can be substituted by such lambda terms. The substitution is defined in a manner similar to that of variable substitution with the base case

$$`f(s_1, s_2, \ldots, s_n)[\lambda y_1.y_2.\ldots.y_n.t/`f] \equiv t[s_1/y_1][s_2/y_2][\ldots][s_n/y_n] \ ,$$

where no y_i is free in any s_j. Otherwise, the lambda term is renamed to an alpha-equivalent term with fresh variable names.

Example 3 (Functional terms substitution in terms)**.**

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

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

Example 4 (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 bound variables may be needed.

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

Example 5 (Typical functional expressions)**.**

$$\begin{aligned} \text{LambdaTermTerm}(x, y)(x + y) &= \lambda x.y.x + y \\ \text{LambdaTermFormula}(x, y)(x = y) &= \lambda x.y.x = y \\ \text{LambdaFormulaFormula}(X, Y)(X \wedge Y) &= \lambda X.Y.X \wedge Y \end{aligned}$$

Note that in the last case, X and Y are FormulaVar. Substitution of functional formulas is completely analogous to (capture free!) substitution of functional terms. Note that there is no expression representing a function taking formulas as arguments and returning a term.

1.1.3 The Equivalence Checker

While proving theorems, trivial syntactical transformations such as $p \wedge q \equiv q \wedge p$ significantly increase the length of proofs, which is desirable neither to 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 transformations. For example, we would like the formulas $p \wedge q$ and $q \wedge p$ to 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 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 quadratic in the size of the formula, which means that it is not significantly slower than syntactic equality. It is based on an algorithm that decides the word problem for Ortholattices [1]. Ortholattices are a generalization of Boolean algebra where instead of the law of distributivity, the weaker absorption law (L9, Table 1.1) holds. In particular, every identity in the theory of ortholattices is also a theorem of propositional logic.

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

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 1.1: Laws of ortholattices, an algebraic theory with signature $(S, \wedge, \vee, 0, 1, \neg)$.

equivalent to $a \vee b = b$. If s and t are propositional formulas, we denote $s \leq_{OL} t$ if and only if $s \leq t$, is provable from the axioms of Table 1.1. We write $s \sim_{OL} t$ if both $s \leq_{OL} t$ and $s \geq_{OL} t$ hold. 1 is the main result we rely on.

Theorem 1 ([?])**.** *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.

LISA's Kernel contains an algorithm, called the $F(OL)^2$ Equivalence Checker which further extends OL inequality algorithm to first order logic formulas. It first expresses the formula using de Bruijn indices, then desugars $\exists.\phi$ into $\neg\forall.\neg\phi$. It then extends the OL algorithm with the rules in Table 1.2.

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

Table 1.2: 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.

In particular, the implementation in LISA also takes into account sym-

metry and reflexivity of equality as well as alpha-equivalence, by which we mean renaming of bound variables. It also expresses $\rightarrow$ and $\leftrightarrow$ in in term of $\vee$ and $\wedge$. A more detailed discussion of extension of ortholattices to first-order logic, proof of correctness and implementation details can be found in [1] and [2].

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 (Γ, Σ) of (possibly empty) sets of formulas, noted:

$$\Gamma \vdash \Sigma$$

The intended semantic of such a sequent is:

$$\bigwedge \Gamma \implies \bigvee \Sigma \,.$$

The sequent may also be written with the elements of the sets enumerated explicitly as

$$\gamma_1, \gamma_2, \ldots, \gamma_n \vdash \sigma_1, \sigma_2, \ldots, \sigma_m \ .$$

A sequent $\phi \vdash \psi$ is logically (but not conceptually) equivalent to a sequent $\vdash \phi \rightarrow \psi$. The distinction is similar to the distinction between meta-implication and inner implication in Isabelle [3], for example. Typically, a theorem or a lemma should have its various assumptions on the left-hand side of the sequent and a single conclusion on the right. During proofs however, there may be multiple elements on the right side. [2]

Sequents are manipulated in a proof using *deduction rules*. A deduction rule, also called a proof step, has zero or more prerequisite sequents (which we call *premises* of the rule) and one conclusion sequent. The basic deduction rules used in LISA are shown in Figure 1.1. This includes first rules of propositional logic, then rules for quantifiers, then equality rules. Moreover, we include equal-for-equal and equivalent-for-equivalent substitutions. While those substitution rules are deduced steps, and hence could technically be omitted, simulating them can sometimes take a high number of steps, so they are included as base steps for efficiency. Finally, the two rules Restate and Weakening leverage the $F(OL)^2$ algorithm.

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

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

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

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

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

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

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

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

$$\frac{\Gamma, \phi[t := 'x] \vdash \Delta}{\Gamma, \forall 'x.\phi \vdash \Delta}\ \text{LeftForall} \qquad \frac{\Gamma \vdash \phi, \Delta}{\Gamma \vdash \forall 'x.\phi, \Delta}\ \text{RightForall}$$

$$\frac{\Gamma, \phi \vdash \Delta}{\Gamma, \exists 'x.\phi \vdash \Delta}\ \text{LeftExists} \qquad \frac{\Gamma \vdash \phi[t := 'x], \Delta}{\Gamma \vdash \exists 'x.\phi, \Delta}\ \text{RightExists}$$

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

$$\frac{\Gamma \vdash \Delta}{\Gamma[\psi(\vec{v}) := 'p(\vec{v})] \vdash \Delta[\psi(\vec{v}) := 'p(\vec{v})]}\ \text{InstSchema}$$

$$\frac{\Gamma, \phi[s := 'f] \vdash \Delta}{\Gamma, s = t, \phi[t := 'f] \vdash \Delta}\ \text{LeftSubstEq} \qquad \frac{\Gamma \vdash \phi[s := 'f], \Delta}{\Gamma, s = t \vdash \phi[t := 'f], \Delta}\ \text{RightSubstEq}$$

$$\frac{\Gamma, \phi[a := 'p] \vdash \Delta}{\Gamma, a \leftrightarrow b, \phi[b := 'p] \vdash \Delta}\ \text{LeftSubstIff} \qquad \frac{\Gamma \vdash \phi[a := 'p], \Delta}{\Gamma, a \leftrightarrow b \vdash \phi[b := 'p], \Delta}\ \text{RightSubstIff}$$

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

$$\frac{\Gamma_1 \vdash \Delta_1}{\Gamma_2 \vdash \Delta_2}\ \text{Restate}\ \text{ if } (\bigwedge \Gamma_1 \to \bigvee \Delta_1) \sim_{F(OL)^2} (\bigwedge \Gamma_2 \to \bigvee \Delta_2)$$

$$\frac{\Gamma_1 \vdash \Delta_1}{\Gamma_2 \vdash \Delta_2}\ \text{Weakening}\ \text{ if } (\bigwedge \Gamma_1 \to \bigvee \Delta_1) \leq_{F(OL)^2} (\bigwedge \Gamma_2 \to \bigvee \Delta_2)$$

Figure 1.1: Deduction rules allowed by LISA's kernel. Different occurrences of the same symbols need not represent equal elements, but only elements with the same $\mathrm{F(OL)}^2$ normal form.

1.2.2 Proofs

Proof steps 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 1.3 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 refers to its 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 uses 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:

```
Proof(steps:List[ProofStep], imports:List[Sequent])
```

We call the bottommost 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 numbers smaller then their own 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 proof of Pierce's Law is shown in Figure 1.4.

1.2.3 Proof Checker

In LISA, a proof object has no guarantee to be correct. It is perfectly possible to write 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.
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. Note that for most cases 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. That way, `LeftAnd`, `RightOr`, `LeftIff` become instances of the `Weakening` rules, and `RightImplies` an instance of `RightAnd`.

`LeftNot`, `RightNot`, `LeftImplies`, `RightImplies`, `LeftRefl`, `RightRefl`, `LeftExistsOne`, `RightExistsOne` can be omitted altogether. This gives an intuition of how useful the equivalence checker is to cut proof length. It also combines 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 sequent semantics (5), 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 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 upon set of axioms, definitions, and previously proven theorems. Formally, theorems are developed 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 within it.

In LISA, a `theory` is a mutable object that starts as the pure theory of predicate logic: It has no known symbols and no axioms. Then we can introduce into it elements of Set Theory (symbols $\in$, $\emptyset$, $\bigcup$ 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 1.5 shows how to define and use new function and predicate symbols. To define a predicate on n variables, we must provide a formula along with n distinguished free variables. Then, this predicate can be freely used and at any time substituted by its definition. Functions are slightly more 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

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

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 lets 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 us to extend the theory without changing what is or isn't provable. For detailed explanation, see part **??**.

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 contain free schematic symbols.

Once a definition has been introduced, future theorems 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 1.6 shows the types of justification in a theory (Theorem, Axiom, Definition). Figure 1.7 shows how to introduce new justifications as well as symbols in the theory. Figure 1.8 shows how to obtain various types of information from the theory.

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 a 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 provides a set of utilities and a Domain Specific Language (DSL) 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

This feature is under active 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 made possible by some of Scala 3's features such as string interpolation, extension and implicits. The way to write a new theory file to mathematical development is:

```
1 object MyTheoryName extends lisa.Main {
2
3 }
```

and that's it! To write a theorem, the recommended syntax is:

```
1 object MyTheoryName extends lisa.Main {
2
3   THEOREM("theoremName") of "desired conclusion" PROOF {
4
5     ???: Proof
6
7   } using (listOfJustifications)
8   show
9 }
```

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

```
1   object MyTheoryName extends lisa.Main {
2
3     val myFunction =
4       DEFINE("symbol", x, y) as definition(x,y)
5     show
6   }
```

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 developed, they all use the same underlying `RunningTheory`. This makes it possible to use results proved previously by means of a simple `import` statement as one would import a regular object. Similarly, one should also import as usual automation and tactics developed alongside. It is expected in the medium term that `lisa.Main` will come with basic automation.

To check the result of a developed 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 its 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 1.9 shows a typical example of set theory development.

0	`Hypothesis`	$\phi \vdash \phi$
1	`Weakening`(0)	$\phi \vdash \phi, \psi$
2	`RightImplies`(1)	$\vdash \phi, (\phi \to \psi)$
3	`LeftImplies`$(2, 0)$	$(\phi \to \psi) \to \phi \vdash \phi$
4	`RightImplies`(3)	$\vdash ((\phi \to \psi) \to \phi) \to \phi$

(a) The proof of Pierce's Law as a sequence of steps using classical Sequent Calculus rules.

```
val PierceLawProof = SCProof(IndexedSeq(
  Hypothesis(ϕ ⊢ ϕ, ϕ),
  Weakening(ϕ ⊢ (ϕ, ϕ), 0),
  RightImplies(() ⊢ (ϕ, ϕ ⟹ ϕ), 1, ϕ, ϕ)
  LeftImplies((ϕ ⟹ ϕ) ⟹ ϕ ⊢ ϕ, 2, 0, ϕ ⟹ ϕ, ϕ),
  RightImplies(() ⊢ ((ϕ ⟹ ϕ) ⟹ ϕ) ⟹ ϕ, 3, (ϕ ⟹ ϕ) ⟹ ϕ, ϕ)
), Seq.empty /* no imports */ )
```

```
val PierceLawProof = SCProof(IndexedSeq(
    Hypothesis(
                φ ⊢ φ
      , φ),
    Weakening(
                φ ⊢ ( φ, ψ )
      , 0),
    RightImplies(
               () ⊢ ( φ, φ ⟹ ψ )
      , 1, φ, ψ)
    LeftImplies(
               ( φ ⟹ ψ ) ⟹ φ ⊢ φ
      , 2, 0, φ ⟹ ψ, φ),
    RightImplies(
               () ⊢ (( φ ⟹ ψ ) ⟹ φ) ⟹ φ
      , 3, ( φ ⟹ ψ ) ⟹ φ, φ)
), Seq.empty /* no imports */ )
```

(b) The proof from Figure 1.4 written for LISA's kernel. $\vdash$ and $\Longrightarrow$ are alternative, nicer constructors for sequents and formulas and are not part of the kernel. The second argument (empty here) is the sequence of proof imports.

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

Figure 1.3: A proof of Pierce's law in Sequent Calculus. The bottommost sequent (root) is the conclusion.

$$\begin{array}{ll}
0 & \texttt{Hypothesis} \quad \phi \vdash \phi \\
1 & \texttt{RightWeakening}(0) \quad \phi \vdash \phi, \psi \\
2 & \texttt{RightImplies}(1) \quad \vdash \phi, (\phi \to \psi) \\
3 & \texttt{Hypothesis} \quad \phi \vdash \phi \\
4 & \texttt{LeftImplies}(2, 3) \quad (\phi \to \psi) \to \phi \vdash \phi \\
5 & \texttt{RightImplies}(4) \quad \vdash ((\phi \to \psi) \to \phi) \to \phi
\end{array} \tag{1.14}$$

Figure 1.4: Linearization of the proof of Pierce's Law as represented in LISA.

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 1.5: Definitions in LISA.

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 1.6: The different types of justification in a `Theory` object.

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 1.7: The mutable interface of a `Theory` object.

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 1.8: The static interface of a `Theory` object.

```
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 1.9: Example of library development in LISA Kernel

Part II

Theory

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-Fraenkel Set Theory. It contains a set of initial predicate symbols and function symbols, as 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 of mathematics, plus the axiom of replacement of Fraenkel (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 its 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.

Bibliography

[1] Simon Guilloud, Mario Bucev, Dragana Milovancevic, and Viktor Kuncak. Formula Normalizations in Verification. In *35th International Conference on Computer Aided Verification*, Lecture Notes in Computer Science, pages –, Paris, 2023. Springer.

[2] Simon Guilloud, Sankalp Gambhir, and Viktor Kuncak. LISA – A Modern Proof System. In *14th Conference on Interactive Theorem Proving*, Leibniz International Proceedings in Informatics, page 0, Bialystok, 2023. Daghstuhl.

[3] Lawrence C. Paulson. Isabelle: The Next 700 Theorem Provers. *CoRR*, cs.LO/9301106, 1993.